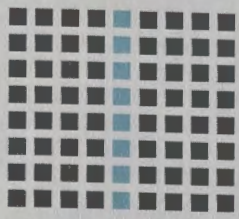


Jorge Cardoso

JAVA PARA TELEMÓVEIS MIDP 2.0

FEUP edições

Java para Telemóveis
MIDP 2.0

Jorge Cardoso

Director da Colecção Documentos Técnicos : Professor Luís Andrade Ferreira
Projecto gráfico : Incomum
Capa : adaptação de ilustração de Sara Coutinho
Impressão e acabamentos : Gráfica Maiadouro
1.ª edição : 2007
Depósito legal n.º 260 856/07
ISBN : 978 972 752 091 6

À Sara

```
while (true);
```

Índice

Lista de Figuras

Lista de Exemplos

Prefácio

A programação em Java para dispositivos móveis, nomeadamente telemóveis, é muito diferente da programação em Java para computadores de secretária. As limitações ao nível da capacidade gráfica, de processamento e de comunicação destes dispositivos implicam uma nova abordagem em termos das plataformas de desenvolvimento. O Java Platform, Micro Edition – Java ME (anteriormente designado de *Java 2 Platform Micro Edition*) – é a proposta da Sun para esta nova abordagem.

Pretende-se, neste livro, iniciar o programador de linguagem Java a este novo ambiente de programação. Mais propriamente, o livro focar-se-á num subconjunto da tecnologia Java ME – as aplicações MIDP, uma vez que é esta a tecnologia presente nos telemóveis Java.

Porque é que eu quero ler este livro?

A plataforma Java ME, mais concretamente o perfil MIDP, introduz uma forma completamente nova de desenvolver aplicações. Temos de ter em conta que os dispositivos para os quais estamos a programar não são computadores de secretária com memória e capacidade de armazenamento e processamento infinitos (OK, não são infinitos, mas muitas vezes nem sequer temos de nos preocupar com isso), mas dispositivos pequenos em que todos os bytes e bytecodes contam!

Este livro descreve o perfil MIDP, versão 2.0, em detalhe e completamente.

Ao longo do livro assumo que o leitor tem um nível de conhecimento médio de programação em Java (Java SE ou Java EE).

Se não sabe programar em Java, recomendo que leia primeiro um livro sobre introdução à linguagem.

Alguns comentários tipográficos e linguísticos

Normalmente, os livros de programação em português usam exemplos de código em que se descartam os caracteres acentuados substituindo-os por caracteres sem acentos. Isto é justificável e, nalguns casos, é mesmo obrigatório já que algumas linguagens de programação apenas permitem o uso de caracteres ASCII Standard (ou seja, não permitem caracteres acentuados). No caso da linguagem Java isso não é verdade. O Java utiliza o conjunto de

caracteres Unicode o que nos permite utilizar praticamente qualquer caracter. Assim, optei por, nos meus exemplos de código deste livro, utilizar o português correcto nos nomes das variáveis, métodos, comentários, etc. À primeira vista parece uma opção sem consequências, mas possui algumas dignas de destaque

- É mais agradável à vista. É minha opinião de que o código se lê de uma forma mais agradável e sem quebrar o fio de leitura do resto do texto. Isto é especialmente verdade no caso dos comentários no próprio código.
- É preciso mais cuidado ao copiar os exemplos para os testar ou modificar porque é mais fácil nos esquecermos de um acento. O compilador não perdoa!

Por outro lado fui obrigado a manter os nomes dos ficheiros sem caracteres acentuados, uma vez que os compiladores Java podem ter problemas com esse tipo de nomes de ficheiros.

Tentei, ao longo deste livro, utilizar ao máximo palavras e expressões portuguesas. Isto não é fácil num livro sobre programação, que implica a utilização de termos informáticos. De qualquer forma, a minha prioridade foi sempre a compreensão fácil dos conceitos, pelo que, nalguns casos, preferi utilizar as expressões estrangeiras por serem as que os programadores estão mais habituados.

Os exemplos de código são apresentados utilizando as convenções de código da linguagem de programação Java [Sun99], nomeadamente no que diz respeito à indentação e formatação do código.

Este livro foi escrito utilizando a linguagem tipográfica LaTeX [Lam94].

O jogo do nome

Durante o evento JavaOne de 2005, que marcou o 10.º aniversário da linguagem Java, a Sun apresentou uma nova forma de designar as tecnologias Java. A partir desse momento, o "2" nos nomes desapareceu e o "0" nos números das versões também (excepto no caso do *Java 2 Platform Micro Edition*, que não tem número de versão):

- *Java 2 Platform Micro Edition* passou a ser designado por *Java Platform, Micro Edition*;
- *Java 2 Platform, Standard Edition 5.0* passou a ser designado por *Java Platform, Standard Edition 5*; e
- *Java 2 Platform, Enterprise Edition 5.0* passou a ser designado por *Java Platform, Enterprise Edition 5*.

Os acrónimos passaram a ser Java ME, Java SE e Java EE (o "J2" foi expandido para "Java").

Este livro estava já a ser terminado na altura do evento JavaOne, mas achei que seria uma boa prenda, pelos 10 anos da linguagem Java, modificar os nomes utilizados de forma a reflectir a nova nomenclatura!

Outros recursos

Este livro é acompanhado por um sítio Web em `http://livromidp.jorgecardoso.org`. Aqui podem encontrar o código-fonte de todos os exemplos dados ao longo deste livro, assim como outras informações relacionadas com este tópico.

Criei também um fórum, em português, sobre MIDP e Java ME em geral que pode ser encontrado em `http://midpforum.jorgecardoso.org`.

O portal Java.pt (`http://java.pt`) é outro lugar onde podem encontrar um fórum de discussão, assim como artigos e tutoriais sobre Java ME.

Se quiserem conhecer outros livros sobre programação em Java para telemóveis aqui fica uma pequena lista (apenas um está escrito em português):

- *Aplicações Móveis com J2ME* de Luís Miguens e Pedro Remelhe [MR05]. Apesar do título, este livro trata apenas de aplicações MIDP e não descreve a API de áudio. Alguns tópicos são abordados muito ligeiramente, como a API de jogos e comunicações. Apesar de tudo, tem a vantagem de ter sido o primeiro livro em português sobre o assunto!
- *Wireless Java* do Jonathan Knudsen [Knu03]. Este livro e o seu autor são referências na área. A primeira edição tratava do MIDP 1.0, a segunda foi aumentada para descrever as novas API introduzidas na versão MIDP 2.0.
- *Learning Wireless Java* do Qusay Mahmoud [Mah02]. Este livro trata apenas do MIDP 1.0.

Estrutura do livro

O livro está estruturado nos seguintes capítulos:

O Capítulo 1, "História do Java ME", faz uma breve descrição da história da linguagem Java e da plataforma Java ME.

O Capítulo 2, "A CLDC em Detalhe", descreve em detalhe a configuração CLDC que serve de base ao MIDP. Neste capítulo, descrevo os dispositivos-alvo, o modelo de segurança e as classes que compõem a CLDC.

O Capítulo 3, "O MIDP em Detalhe", aborda o perfil MIDP, descrevendo o tipo de API, o ciclo de vida das aplicações MIDP, a forma como as aplicações são distribuídas e as principais diferenças entre as versões 1.0 e 2.0.

O Capítulo 4, "O Clássico... (Olá Mundo!)", apresenta a primeira MIDlet e descreve os passos necessários para a compilar e testar. Neste capítulo descrevo também o ambiente de desenvolvimento *J2ME Wireless Toolkit*.

No Capítulo 5, "A Interface com o Utilizador – API de Alto Nível", descrevo os componentes que fazem parte da API de alto nível relacionada com a interface com o utilizador.

O Capítulo 6, "A Interface com o Utilizador – API de Baixo-Nível", é a continuação do anterior mas, neste caso, descrevo os componentes que fazem parte da API de baixo nível.

O Capítulo 7, "Armazenamento Persistente", descreve a forma como se armazenam dados em MIDP.

O Capítulo 8, "*Threads*", aborda algumas questões relacionadas com *threads*, que, embora não totalmente específicas do MIDP, são importantes para o desenvolvimento de aplicações MIDP.

O Capítulo 9, "Comunicações", descreve como realizar comunicações entre as aplicações MIDP e servidores externos.

O Capítulo 10, "*Push Registry*", descreve como as MIDlets podem ser lançadas em resposta a ligações vindas do exterior ou através de temporizadores.

O Capítulo 11, "Áudio", apresenta a API para geração e reprodução de áudio.

O Capítulo 12, "Jogos – API e Técnicas Básicas", trata da API de jogos introduzida no MIDP 2.0.

No final do livro encontra-se ainda um glossário com a definição de alguns termos e expressões muito utilizadas e uma lista dos atributos das MIDlets para consulta rápida.

Agradecimentos

Este livro, como qualquer outro, não é uma obra de uma pessoa isolada. Várias pessoas contribuíram, de uma forma ou de outra, para o resultado final aqui apresentado.

Tenho de agradecer em especial ao Nuno Ramos, o "Crivo Número Um", pelos comentários, sugestões e ideias oferecidos à medida que cada capítulo era terminado.

Ao meu irmão Jaime pelas correcções à primeira versão.

O meu obrigado também ao Paulo Moreira, que leu uma das primeiras versões do livro e com quem troquei algumas impressões por email.

Ao Miguel Gonçalves, pelos conhecimentos musicais ;)

Finalmente, à Sara, por tudo... e mais alguma coisa... :)

PARTE I

Introdução à Tecnologia *Java Platform, Micro Edition*

CAPÍTULO 1
História do Java ME

Este primeiro capítulo pretende dar uma pequena visão histórica sobre as origens da linguagem Java e do Java ME.

1.1. Nasce uma Linguagem de Programação

A linguagem Java, cujo nome original era *Oak*[1], nasceu de um projecto da *Sun Microsystems* – o projecto *Green* [Gos] – que começou em Dezembro de 1990.

O objectivo do projecto Green era perceber qual seria a principal tendência da indústria da computação da época e o que a Sun poderia fazer para se manter a par. A conclusão dos estudos levados a cabo no âmbito deste projecto foi a de que a tendência seria a convergência dos computadores com a electrónica de consumo – pequenos sistemas embebidos, portáteis e distribuídos.

No âmbito deste projecto foi desenvolvido um protótipo de um dispositivo chamado *Star7*. O Star7 era um *Personal Digital Assistant* (PDA) com comunicação sem fios, um monitor LCD táctil de 5 polegadas, interfaces PCMCIA, etc., que corria uma versão do sistema operativo Unix em menos de 1 megabyte. O Star7 foi demonstrado em Setembro de 1992.

Inicialmente, a intenção era usar C++ para o novo dispositivo, mas uma sucessão de problemas levou ao desenvolvimento de uma nova linguagem – assim nasceu o Java, uma linguagem para pequenos dispositivos móveis.

O objectivo era construir uma linguagem simples, orientada a objectos, distribuída, interpretada, robusta, segura, independente da arquitectura, portátil, com alto desempenho, multi-*threaded* e dinâmica [Gos95].

O projecto Star7 acabou por morrer alguns anos depois mas, nessa mesma altura, com a massificação da Web, percebeu-se que existia um ambiente ideal para a aplicação dos conceitos de multiplataforma que estiveram na base do Java. Foi assim que apareceram as *applets* – pequenos programas que eram descarregados dos servidores e executados localmente. Foi o aparecimento das *applets* que trouxe visibilidade à linguagem Java, principalmente depois do anúncio, em 1995, na conferência *Sun World*, de que o *browser* Netscape iria suportar a tecnologia das *applets*.

1. Aparentemente o nome foi alterado para Java devido a um problema de registo de marcas.

As *applets* nunca "pegaram" como se esperava inicialmente. Na minha opinião, porque a maior parte dos profissionais que trabalham nos conteúdos para a Web não são programadores e por isso não têm o conhecimento necessário para o desenvolvimento de *applets*. Para além disso foram surgindo tecnologias, como o *Macromedia Flash*, mais fáceis de utilizar e que permitiam fazer muito do que se fazia com as *applets*.

A linguagem, no entanto, encantou muitos programadores pela sua facilidade de programação. O facto de ter uma sintaxe muito próxima do C/C++ contribuiu para isso, uma vez que os programadores de C/C++ sentiram que estavam num ambiente conhecido ao utilizar o Java.

Houve uma grande evolução desde a altura em que foi concebida e, neste momento, é mais do que uma linguagem de programação. Podemos dividir a tecnologia Java em três áreas principais[2]:

Java SE O *Java Platform, Standard Edition* é um ambiente de desenvolvimento de aplicações em *desktops* e servidores. Serve de base ao *Java Platform, Enterprise Edition*.

Java EE O *Java Platform, Enterprise Edition* é orientado para o desenvolvimento de aplicações empresariais, define o *standard* para desenvolvimento de aplicações de várias camadas e baseadas em componentes.

Java ME O *Java Platform, Micro Edition* é um ambiente de desenvolvimento orientado para a electrónica de consumo.

1.2. Java ME – o Filho Pródigo Regressa a Casa

Podemos pensar no Java ME como um regresso às origens por parte do Java, uma vez que inicialmente foi desenvolvido a pensar exactamente no tipo de dispositivos a que se destina agora a tecnologia Java ME. Podemos dizer que, finalmente, o Java está no seu "habitat natural".

O Java ME nasceu da necessidade de adaptar a tecnologia Java existente aos dispositivos móveis com sérias limitações de recursos, quando comparados com os PC de secretária.

É um ambiente Java direccionado a uma vasta gama de dispositivos, que vai desde *Smart Cards* até *Set-top Boxes*, passando por telemóveis e PDA. Consiste num conjunto de especificações organizadas em camadas que permitem abranger um largo leque de dispositivos e tecnologias.

Basicamente, a arquitectura do Java ME está dividida em três camadas, como se pode ver na Figura 1.1: *máquina virtual*, *configurações* e *perfis*.

2 Na verdade, podemos enumerar mais áreas, como o *Java Card* ou *Java Web Services*, mas considero que estas são as que merecem mais destaque.

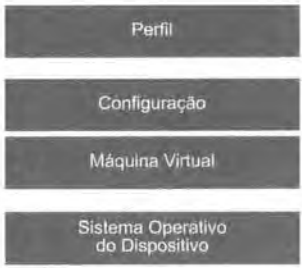


Figura 1.1 Arquitectura de um ambiente Java ME.

A máquina virtual está colocada directamente acima do sistema operativo do dispositivo. É a máquina virtual que define quais são as limitações dos programas que podem executar nos dispositivos.

A configuração é um conjunto de bibliotecas básicas que estão disponíveis para o programador. Uma configuração é definida para uma classe horizontal de dispositivos, i.e., uma gama de dispositivos, com diferentes aplicações, mas que partilham algumas características. Por exemplo, a classe dos dispositivos de informação com comunicação *wireless* abrange vários tipos de dispositivos, desde telemóveis, PDA, *pagers*, etc. A justificação para a criação de configurações é que, apesar dos dispositivos serem muito diversos na forma e na funcionalidade, têm, frequentemente, processadores e quantidades de memória muito similares. É a camada da configuração que define o nível de funcionalidades e serviços que têm de ser oferecidos pela máquina virtual; por isso, a máquina virtual e a configuração estão fortemente interligadas.

Finalmente, o perfil define um conjunto de bibliotecas específicas para uma classe vertical de dispositivos. Seguindo o exemplo anterior, poderíamos ter um perfil para telemóveis, outro para PDA, etc. As bibliotecas definidas por um perfil são mais específicas do que as definidas pela configuração uma vez que a gama de dispositivos-alvo também é menor. Um perfil é sempre especificado para uma determinada configuração, mas uma configuração pode suportar vários perfis.

Para além das configurações e dos perfis, existe ainda um outro tipo de bibliotecas chamadas *pacotes opcionais*. Os pacotes opcionais são bibliotecas de programação específicas a uma determinada tecnologia e que aumentam as capacidades do ambiente Java, caso estejam implementadas no dispositivo. Estes pacotes são chamados opcionais porque, mesmo que não estejam implementados num determinado dispositivo, esse dispositivo continua a ser considerado um dispositivo Java ME.

Neste momento existem apenas duas configurações definidas:

Connected Limited Device Configuration – **CLDC** Esta configuração é utilizada em dispositivos muito limitados, i.e., telemóveis, *pagers*, PDA, com capacidade de ligação à rede.

Connected Device Configuration – **CDC** Usada em dispositivos com maior capacidade (de memória e processamento) e também com conectividade a uma rede. Exemplos de dispositivos que podem ser incluídos nesta categoria são: *set-top boxes* para televisores, alguns PDA, sistemas de navegação para automóveis, etc.

A Figura 1.2 mostra as várias tecnologias Java e os dispositivos ou mercados-alvo. É de salientar que a linha que divide as várias tecnologias, principalmente no que diz respeito à CDC e à CLDC, não é exactamente bem definida. Alguns dispositivos poderiam perfeitamente ser colocados em mais do que uma categoria.

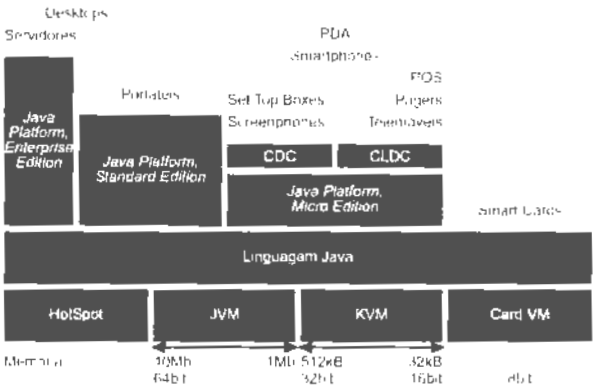


Figura 1.2 As tecnologias Java e os mercados-alvo.

1.3. *Connected, Limited Device Configuration*

A CLDC é orientada para dispositivos muito limitados a nível de processamento e memória, i.e., telemóveis, PDA, *pagers*, etc.

A palavra *Limited* no nome da configuração não se refere à conexão à rede, mas sim ao próprio dispositivo. Alguns autores afirmam que a CLDC é orientada para dispositivos com conexão limitada à rede e chegam mesmo a referir que um melhor nome seria *Limited*

Connection Device Configuration[3]. Tal ideia é, do meu ponto de vista, errada. Se os engenheiros da Sun se quisessem referir a dispositivos desse tipo teriam nomeado a configuração de forma mais apropriada. Além do mais, se dúvidas restassem, bastava ler o título da especificação CLDC 1.0 [Sun00]: "*Connected, Limited Device Configuration*"[4]. A vírgula não deixa margem para dúvidas; esta configuração destina-se a dispositivos limitados a vários níveis (processamento, memória, ecrã), ligados de alguma forma a uma rede (o que não significa que a ligação não tenha, ela própria, algumas limitações).

As características gerais de um dispositivo CLDC[5] são:

- Pelo menos 128 kilobytes de memória não volátil[6] disponíveis para a máquina virtual e as bibliotecas CLDC.
- Pelo menos 32 kilobytes de memória volátil disponíveis para o *runtime* da máquina virtual (por exemplo, para o *heap*).
- Um processador de 16 bits ou de 32 bits.
- Baixo consumo de energia, operando frequentemente através de bateria.
- Conectividade a algum tipo de rede, frequentemente sem fios, de forma intermitente e com largura de banda limitada.

Neste momento existem dois perfis definidos para a CLDC: o perfil *Mobile Information Device Profile* – MIDP (já com duas versões) e o perfil *Information Module Profile* – IMP.

O perfil IMP é mais recente do que o MIDP e é, basicamente, um subconjunto deste último. O IMP é destinado a dispositivos com uma interface com o utilizador mais limitada do que os dispositivos MIDP. A principal diferença do IMP relativamente ao MIDP é não ter API relativas à interface gráfica com o utilizador.

A máquina virtual correspondente a esta configuração é a chamada KVM – uma máquina virtual compacta desenhada especificamente para dispositivos pequenos e com recursos limitados. O objectivo principal era criar uma máquina virtual Java "completa" o mais pequena possível que mantivesse os aspectos centrais da linguagem de programação Java, mas que corresse num dispositivo com apenas algumas centenas de kilobytes de memória. Aliás, o K em KVM significa "kilo" – uma alusão à dimensão da máquina virtual.

3. Configuração para Dispositivos com Conexão Limitada.
4. Configuração para Dispositivos Limitados, Conectados.
5. Requisitos para a versão CLDC 1.0. No caso da CLDC 1.1, a única diferença é o aumento de memória não volátil para 160 kB.
6. Memória não volátil é memória cujo conteúdo se mantém mesmo que o utilizador desligue o dispositivo.

1.4. *Connected Device Configuration*

A CDC é orientada para dispositivos relativamente potentes que estão ligados a uma rede, ainda que de forma intermitente. Estes dispositivos têm geralmente as seguintes características:

- Processador de 32 bits.
- Disponibilidade para o ambiente Java de pelo menos 2Mb de memória.
- Conectividade a algum tipo de rede, frequentemente sem fios.
- Interface com o utilizador com alguma sofisticação.

Os seguintes perfis estão disponíveis para a CDC neste momento:

***Foundation Profile* (FP)** O FP estende as API da CDC com serviços que praticamente todas as aplicações baseadas na CDC necessitam. Este perfil não inclui API relacionadas com a interface com o utilizador uma vez que nem todas as aplicações dela precisam.

***Personal Basis Profile* (PBP)** O Personal Basis Profile adiciona suporte para os componentes *lightweight* AWT (*Abstract Windowing Toolkit*)[7] e o modelo de aplicação *xlet*[8].

***Personal Profile* (PP)** Contém todas as API do PBP e adiciona suporte para compatibilidade total com o AWT e o modelo de aplicação *applet*.

A máquina virtual correspondente à CDC é a CVM e, basicamente, suporta todas as funcionalidades da máquina virtual Java 2 Versão 1.3. O nome – CVM – significava originalmente Compact Virtual Machine, mas como o Compact poderia ser confundido com o K de KVM, neste momento o C não tem nenhum significado especial[9].

7. O AWT é o conjunto de classes básicas para a construção de interfaces gráficas com o utilizador. O AWT permite construir objectos gráficos como janelas, caixas de diálogo, botões, etc.
8. O modelo *xlet* é semelhante ao das *applets* – são aplicações descarregadas dinamicamente e executadas sob condições de segurança.
9. Encontrei esta explicação num artigo no portal Java da Sun.

CAPÍTULO 2
A CLDC em Detalhe

Como referido no Capítulo 1, a CLDC é uma configuração da plataforma Java ME. Como tal, especifica um subconjunto da linguagem de programação Java, um subconjunto das funcionalidades da máquina virtual Java da configuração, características de segurança e comunicação e também as bibliotecas nucleares da plataforma. Este capítulo descreve a configuração CLDC em detalhe.

2.1.A Especificação CLDC

As configurações são definidas para classes horizontais de dispositivos, isto é, para dispositivos com funções e usos diferentes mas que partilham algumas características, e.g., processadores semelhantes, quantidade de memória na mesma ordem de grandeza, etc. Isto significa que uma determinada configuração Java ME determina quais as características mínimas da máquina virtual, quais as bibliotecas básicas e quais os atributos da linguagem Java que o programador terá ao seu dispor.

A CLDC é descrita num documento – a especificação CLDC – também conhecido por JSR-30[1], para o qual contribuíram, para além da Sun Microsystems, vários parceiros industriais: America Online, Bull, Ericsson, Matsushita, Mitsubishi, Motorola, Nokia, NTT DoCoMo, Oracle, Palm Computing, RIM, Samsung, Sharp, Siemens, Sony e Symbian, para citar apenas os envolvidos na versão 1.0.

A especificação abrange áreas como: definição dos dispositivos-alvo, modelo de segurança, características da linguagem Java e características da máquina virtual Java. Ficam fora do âmbito da CLDC as seguintes áreas: interface com o utilizador, gestão do ciclo de vida da aplicação, gestão de eventos e modelo de alto nível da aplicação. Estas áreas são tratadas pelos perfis implementados em cima da CLDC.

A primeira versão da CLDC, a versão 1.0, foi terminada no ano 2000; a versão 1.1 saiu em 2003.

1. *Java Specification Request* 30. A CLDC 1.1 é descrita no JSR-139.

2.2. Dispositivos-Alvo

A CLDC foi pensada para dispositivos pequenos e limitados ao nível de processamento, memória e interface com o utilizador. Em termos gerais, os dispositivos-alvo têm as seguintes características mínimas:

- 128 kilobytes de memória não volátil disponíveis para a máquina virtual e as bibliotecas CLDC.
- 32 kilobytes de memória volátil disponíveis para o *runtime* da máquina virtual (por exemplo, para o *heap*).
- Um processador de 16 bits ou de 32 bits.
- Baixo consumo de energia, frequentemente operando através de bateria.
- Conectividade a algum tipo de rede, frequentemente sem fios, de forma intermitente e com largura de banda limitada.
- Um sistema operativo ou *kernel* para gerir o hardware e fornecer pelo menos uma entidade escalonável para executar a máquina virtual Java. O sistema operativo não necessita, no entanto, de suportar espaços de endereçamento diferentes nem garantir escalonamento em tempo real.

Podem incluir-se neste grupo dispositivos como PDA, *pagers*, terminais *Point of Sale* (POS) e alguns electrodomésticos. No entanto, os dispositivos mais conhecidos por implementarem esta configuração são os telemóveis.

2.3. Modelo de Segurança

Uma área de grande importância no Java ME é a segurança. Uma vez que uma das grandes vantagens da plataforma é a possibilidade de descarregar e instalar aplicações dinamicamente nos dispositivos dos clientes, é necessário garantir que as aplicações descarregadas não possam, voluntária ou involuntariamente, avariar ou corromper o dispositivo ou os dados dos utilizadores.

A CLDC define um modelo de segurança a dois níveis: segurança de baixo nível da máquina virtual e segurança ao nível da aplicação.

A segurança de baixo nível diz respeito à forma como os programas Java são executados e garante que uma aplicação Java não pode danificar o dispositivo em que está a executar. Numa máquina virtual *standard*, isto é feito pelo verificador dos ficheiros de classes (*classfile verifier*), que garante que os *bytecodes* não têm referências para posições inválidas de memória ou para posições de memória que não pertencem à memória de objectos do Java.

Na máquina virtual da CLDC o processo é semelhante, mas a verificação é feita de forma diferente.

O verificador Java SE necessita de mais memória e poder de processamento do que alguns dispositivos conseguem oferecer, por isso teve de ser encontrada outra forma de verificação. Basicamente, na CLDC, o processo de verificação foi dividido em duas fases: pré-verificação no computador de desenvolvimento da aplicação e verificação no dispositivo no momento de execução.

A pré-verificação altera os ficheiros de classes modificando algumas instruções e adicionando atributos que serão utilizados pelo verificador do dispositivo. Desta forma a verificação no dispositivo é feita de forma muito mais eficiente e rápida e garante que a classe é um programa Java válido.

A segurança de baixo nível é muito limitada. Nada garante no que diz respeito ao acesso a recursos externos como ficheiros, rede, portas de comunicação, etc. No Java SE estes aspectos são tratados pelo gestor de segurança. No entanto, implementar um gestor de segurança num dispositivo CLDC seria impraticável, pelo que teve de ser encontrado outro método. A solução foi o chamado modelo de segurança "caixa de areia" (*sandbox model*). O uso deste modelo significa que a aplicação Java executa num ambiente fechado e que só consegue aceder às funções definidas pela configuração, perfis e pelas classes dos fabricantes.

2.4. Diferenças para um Ambiente Java "Normal"

O objectivo geral de uma máquina virtual que esteja conforme à especificação CLDC é aproximar-se tanto quanto possível de um ambiente Java "normal", i.e., um ambiente que está conforme à Especificação da Linguagem Java (*Java Language Specification* [GJS96]) e à Especificação da Máquina Virtual Java (*Java Virtual Machine Specification* [LY97]), dentro das limitações dos dispositivos-alvo da CLDC. Assim, é mais prático descrever a especificação CLDC referindo as diferenças para esse ambiente Java.

Em termos das características da máquina virtual as principais diferenças são:

Não suporta números de vírgula flutuante A versão 1.0 da CLDC não tem suporte para números de vírgula flutuante. Esta opção foi tomada na altura porque a maioria dos dispositivos-alvo não tinham suporte para vírgula flutuante por *hardware* e o custo da implementação de vírgula flutuante por *software* foi considerado muito alto. Isto significa que a máquina virtual não permite o uso de literais, operações, tipos ou valores de vírgula flutuante, i.e., `float x = 0.1f;` não é permitido. A versão 1.1 da CLDC, no entanto, reintroduz o suporte para números de vírgula flutuante. Assim, é preciso tomar em atenção o dispositivo para o qual estamos a programar caso a nossa aplicação

necessite de cálculos com este tipo de números. No caso de estarmos a programar para a CLDC 1.0 e necessitarmos de efectuar cálculos com casas decimais existem algumas bibliotecas que implementam essas operações por software. Um exemplo é a biblioteca `MathFP` [jT03], que usa números de vírgula fixa.

Não suporta finalização de objectos As bibliotecas da configuração CLDC não incluem o método `finalize()` da classe `Object` pelo que a máquina virtual não necessita de suportar finalização de objectos. As aplicações desenvolvidas não podem requerer esta funcionalidade.

Limitações ao nível do tratamento de erros e excepções A máquina virtual da CLDC suporta excepções, mas não excepções assíncronas[2]. As classes de erro incluídas na CLDC também são mais limitadas. O conjunto de classes de erros e excepções da CLDC é apresentado mais à frente.

***Não suporta o Java Native Interface* (JNI)**[3] A máquina virtual não implementa o JNI. O JNI foi excluído por razões de segurança e por limitações de memória. Uma vez que o modelo de segurança utilizado é o modelo "caixa de areia", que obriga a que o conjunto de funções nativas seja fechado, o JNI não poderia ser suportado porque estaria a quebrar o modelo de segurança. Por outro lado, a implementação completa do JNI era demasiado dispendiosa dadas as limitações de memória dos dispositivos CLDC.

Não suporta reflexão (*reflection*) A máquina virtual não suporta reflexão, *i.e.*, não permite a um programa inspeccionar o número e o conteúdo de classes, métodos, campos, *threads*, pilhas de execução e outras estruturas da máquina virtual. Como consequência desta limitação a máquina virtual não suporta RMI (*Remote Method Invocation*), JVMDI (*Debugger Interface*), JVMPI (*Profiler Interface*) e outras características avançadas que necessitam de reflexão.

Não suporta *class loaders* definidos pelo utilizador Esta restrição foi imposta por questões de segurança. O carregamento de classes é feito pelo *class loader* da máquina virtual e não pode ser substituído ou reconfigurado.

Não suporta grupos de *threads* nem *daemon threads*[4] A máquina virtual suporta *threading* mas não suporta grupos de *threads* nem *daemon threads*. O suporte para grupos de *threads* tem de ser feito a nível da aplicação, caso estas necessitem desta funcionalidade.

2. Excepções assíncronas são excepções em que o ponto do programa em que ocorrem não é conhecido. Um exemplo de excepção assíncrona é a `InternalError` causada por um erro na máquina virtual. Para mais informação ver [GJS96].

3. O JNI é um mecanismo que permite que aplicações Java invoquem métodos implementados em código nativo.

4. Há dois tipos de *threads* em Java: *user threads* e *daemon threads*. A diferença é a seguinte: se o sistema determina que não existem mais *user threads* a executar, *i.e.*, existem apenas *daemon threads*, o programa é terminado. De resto, estes dois tipos de *threads* são tratados da mesma forma.

Não suporta referências fracas (*weak references*)[5] A máquina virtual CLDC 1.0 não suporta referências fracas. Um subconjunto da API relativa às referências fracas foi introduzido na versão 1.1 da CLDC.

2.5. Classes Derivadas do Java SE

Grande parte das bibliotecas da CLDC são subconjuntos das bibliotecas do Java SE. Desta forma consegue-se, pelo menos parcialmente, retrocompatibilidade e portabilidade das aplicações. Mas, mais importante ainda, do meu ponto de vista, consegue-se criar um ambiente familiar para os programadores Java habituados a programar em Java SE ou Java EE. As classes que são derivadas do Java SE encontram-se nos pacotes `java.lang.*`, `java.util.*` e `java.io.*`.

Estas classes não são, no entanto, completamente iguais às da versão Java SE. Muitos métodos foram removidos de forma a tornar a implementação mais pequena.

As classes derivadas do Java SE estão listadas na Tabela 1 (excepto as classes de excepções e de erros).

Tabela 1 Classes CLDC derivadas do Java SE

java.lang	java.io	java.util
Object	InputStream	Vector
Class	OutputStream	Stack
Runtime	ByteArrayInputStream	Hashtable
System	ByteArrayOutputStream	*Enumeration*
Thread	*DataInput*	Calendar
Runnable	*DataOutput*	Date
String	DataInputStream	TimeZone
StringBuffer	DataOutputStream	Random
Throwable	Reader	
Boolean	Writer	
Byte	InputStreamReader	
Short	OutputStreamWriter	
Integer	PrintStream	
Long		
Character		
Enumeration		
Math		

5. Referências fracas são uma forma de um programa Java manter uma referência para um objecto sem que esta referência impeça o *garbage collector* de o reclamar.

2.6. Classes Específicas

Para além das classes derivadas do Java SE existem também algumas específicas da CLDC. Estas classes fazem todas parte do pacote `javax.microedition.io` e constituem o chamado *Generic Connection Framework* (GCF). O GCF foi a forma encontrada para o suporte de vários tipos de protocolos de comunicação em dispositivos com as características dos dispositivos-alvo da CLDC. A ideia é, em vez de usar abstracções totalmente diferentes para as diversas formas de comunicação, como acontece no Java SE, usar uma série de abstracções relacionadas entre si ao nível da programação da aplicação. Basicamente, todas as conexões são criadas usando apenas um método estático numa classe chamada `Connector`. Este método irá devolver um objecto que implementa uma das interfaces genéricas consoante o URI[6] passado como parâmetro. Por exemplo[7]:

```
Connector.open("http://www.jorgecardoso.org");
```

irá devolver um objecto que implementa uma interface HTTP, enquanto que

```
Connector.open("socket://172.20.85.211:2005");
```

irá devolver um objecto que implementa uma interface do tipo socket.

As classes específicas da CLDC são as seguintes:

Classes do *Generic Connection Framework*

`javax.microedition.Connection` (interface)
`javax.microedition.ContentConnection` (interface)
`javax.microedition.Datagram` (interface)
`javax.microedition.DatagramConnection` (interface)
`javax.microedition.InputConnection` (interface)
`javax.microedition.OutputConnection` (interface)
`javax.microedition.StreamConnection` (interface)
`javax.microedition.StreamConnectionNotifier` (interface)
`javax.microedition.Connector`
`javax.microedition.ConnectionNotFoundException`

6. *Uniform Resource Indicator* – RFC2396 (`http://www.ietf.org/rfc/rfc2396.txt`).
7. Estes exemplos são apenas ilustrativos. A CLDC não inclui nenhuma implementação de protocolos; a implementação fica a cargo dos perfis.

CAPÍTULO 3
O MIDP em Detalhe

Um perfil Java ME estende a API de uma configuração acrescentando-lhe funcionalidades que fazem sentido numa determinada gama vertical de dispositivos, por exemplo, telemóveis. Neste capítulo descrevo o perfil MIDP – o perfil utilizado na maioria dos telemóveis Java.

3.1. Visão Geral

O MIDP[1] é um dos perfis definidos para a CLDC e está agora na versão 2.0, embora ainda existam dispositivos que implementam apenas a versão 1.0.

A arquitectura geral de um dispositivo MIDP é a indicada na Figura 3.1. Todos os dispositivos conformes à especificação MIDP implementam as classes da configuração CLDC e as classes do perfil MIDP. Estas classes tentam fornecer uma base comum de desenvolvimento de aplicações, mas obviamente não cobrem todas as funcionalidades de dispositivos particulares. Para colmatar estas restrições, alguns fabricantes incluem bibliotecas Java próprias de forma que as aplicações desenvolvidas possam tirar partido de funcionalidades particulares dos seus dispositivos. As aplicações desenvolvidas usando estas bibliotecas são específicas de um determinado dispositivo (ou dispositivos de um fabricante), não podendo ser, por isso, executadas noutros dispositivos. Por outro lado as aplicações escritas para MIDP em geral podem correr em qualquer dispositivo que siga a especificação MIDP. As aplicações nativas são aplicações que usam o sistema operativo nativo directamente, i.e., não são aplicações Java e são, tipicamente, instaladas pelo fabricante e não podem ser removidas ou actualizadas.

1. Não sei se há outra forma, mas eu costumo pronunciar MIDP como "midepé".

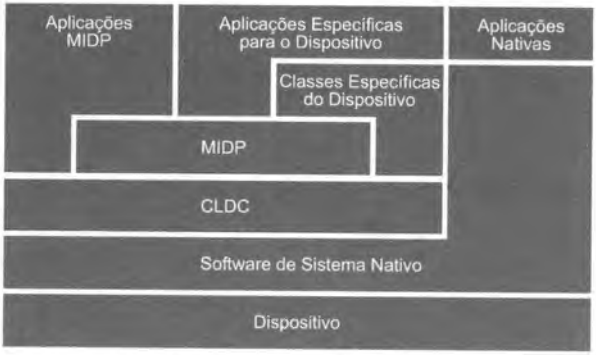


Figura 3.1 Arquitectura de um dispositivo MIDP.

O perfil MIDP é descrito, à semelhança da configuração CLDC, num documento produzido por um grupo de trabalho composto por pessoas de várias entidades – a especificação MIDP (ou JSR-118[2]). Este documento descreve, entre outras coisas, os dispositivos-alvo, o ciclo de vida das aplicações, as várias API fornecidas pelo perfil e alguns aspectos de segurança.

3.2. Dispositivos MID

O perfil MIDP foi desenvolvido para dispositivos de informação móveis (*Mobile Information Device*) – MID. Segundo a especificação MIDP 2.0 [Mic02], um dispositivo MID deve ter as seguintes características mínimas de hardware:

Ecrã

- Tamanho: 96x54 píxeis.
- Profundidade de cor: 1 bit.
- Forma do píxel (*aspect ratio*): aproximadamente 1:1.

Entrada

- Um ou mais dos mecanismos de entrada seguintes: "teclado de uma mão"[3], "teclado de duas mãos"[4] ou ecrã táctil.

2. O JSR-118 corresponde à versão 2.0 do MIDP que é a versão à qual vou dar mais atenção. O MIDP 1.0 é descrito no JSR-37.
3. "Teclado de uma mão" é um termo usado para descrever os teclados de telefone ITU-T.
4. "Teclado de duas mãos" é um termo usado para descrever os teclados QWERTY de computador.

Memória[5]

- 256 kilobytes de memória não volátil para os componentes MIDP.
- 8 kilobytes de memória não volátil para dados persistentes criados pelas aplicações.
- 128 kilobytes de memória volátil para o *runtime* do Java (*i.e.*, *heap* do Java).

Comunicação

- Comunicação sem fios, nos dois sentidos, possivelmente intermitente e com largura de banda limitada.

Em relação ao software, o MIDP assume apenas funcionalidades mínimas uma vez que, ao nível dos dispositivos MID, há uma grande variedade de arquitecturas de software de sistema. Os requisitos mínimos são os seguintes:

- Um mecanismo para ler e escrever em memória não volátil.
- Acesso de leitura e escrita ao sistema de comunicação sem fios do dispositivo.
- Um mecanismo que forneça uma base de tempo para ser usado nas API de temporizadores.
- Capacidade mínima para escrever gráficos do tipo mapa de bits no ecrã.
- Um mecanismo para capturar *input* do utilizador.
- Um mecanismo para gerir o ciclo de vida das aplicações do dispositivo.

Para além dos requisitos previamente mencionados, a especificação MIDP 2.0 define outros que não podem ser facilmente incluídos nas categorias anteriores. Segundo a especificação, os dispositivos MIDP 2.0 são obrigados a:

- Suportar MIDlets e MIDlet Suites (descritas mais à frente) em ambas as versões MIDP 1.0 e MIDP 2.0. Ou seja, devem ser retrocompatíveis com a versão anterior do MIDP.
- Implementar a especificação OTA (descrita mais à frente).
- Fornecer indicações visuais ao utilizador relativamente ao uso da rede.
- Suportar o acesso a servidores HTTP 1.1 e conexões HTTP seguras (HTTPS).
- Suportar transparência nas imagens PNG.
- Suportar geração de tons sonoros (biblioteca multimédia).
- Suportar SP-MIDI (*Scalable Polyphony MIDI*)[6].

[5] Estes requisitos de memória são para os componentes MIDP apenas. Os requisitos para a CLDC e outros requisitos de memória do sistema não são considerados.

[6] SP-MIDI é um formato MIDI (*Musical Instrument Digital Interface*) adaptado a dispositivos móveis.

3.3. Bibliotecas do MIDP

O MIDP oferece ao programador algumas bibliotecas básicas para o desenvolvimento de aplicações. Estas bibliotecas abrangem as seguintes áreas:

Bibliotecas nucleares O perfil MIDP adiciona algumas classes e funcionalidades aos pacotes `java.lang` e `java.util` da CLDC.

Ciclo de vida da aplicação Define o ciclo de vida da aplicação e a forma como é controlada pelo dispositivo.

Interface gráfica Funções para construir interfaces gráficas e para obter o *input* do utilizador.

Armazenamento persistente Apesar de não existir o conceito de ficheiro em MIDP, existem, no entanto, funções para ler e escrever dados num sistema de armazenamento persistente.

Comunicação O MIDP disponibiliza funções para aceder ao sistema de comunicação do dispositivo, *i.e.*, possibilita o uso de conexões HTTP, datagramas, acesso a portas-série, etc.

Multimédia Funções multimédia como geração de tons sonoros, notas MIDI (*Musical Instrument Digital Interface*), reprodução de ficheiros áudio e vídeo, etc.

Segurança Biblioteca relacionada com certificados usados em ligações seguras.

A Tabela 2 mostra os pacotes Java definidos pelo perfil MIDP.

Tabela 2 Pacotes da API MIDP 2.0

Funções	Pacote
Bibliotecas nucleares	`java.lang`
	`java.util`
Ciclo de vida da aplicação	`javax.microedition.midlet`
Interface gráfica	`javax.microedition.lcdui`
	`javax.microedition.lcdui.game`
Armazenamento persistente	`javax.microedition.rms`
Comunicação	`javax.microedition.io`
Multimédia	`javax.microedition.media`
Segurança	`javax.microedition.pki`

3.4. MIDlets e MIDlet Suites

As aplicações escritas para o perfil MIDP são designadas por *MIDlets*. Uma MIDlet consiste numa classe que estende a classe `javax.microedition.midlet.MIDlet` e nas outras classes necessárias à aplicação. As MIDlets são distribuídas em ficheiros JAR, à semelhança do que já se fazia no Java SE (embora não fosse obrigatório, *i.e.*, as aplicações podiam ser distribuídas como um conjunto de ficheiros de classes). Várias MIDlets podem ser empacotadas num ficheiro JAR; isto é designado por *MIDlet Suite*. Neste caso, as várias MIDlets podem partilhar classes e recursos contidos no ficheiro JAR.

3.4.1. Ciclo de vida das MIDlets

Uma MIDlet tem um ciclo de vida semelhante às applets. Há três estados possíveis no ciclo de vida de uma MIDlet:

Inactiva A instância da MIDlet foi construída e está inactiva.

Activa A MIDlet está activa.

Destruída A MIDlet foi terminada.

O ciclo de vida das MIDlets é gerido pelo *gestor de aplicações* (ou *application management software*) – AMS do dispositivo. O AMS é, basicamente, um programa (ou conjunto de programas) que gere a instalação, execução e remoção das aplicações Java no dispositivo.

A Figura 3.2 mostra o diagrama de estados de uma MIDlet e as transições entre os possíveis estados. Imediatamente depois de ser construída, a MIDlet encontra-se no estado inactivo (*paused*). De seguida o AMS invoca o método `startApp()` e a MIDlet passa para o estado activo. É neste estado que o utilizador pode interagir com a aplicação. Do estado activo, a MIDlet pode passar para o estado inactivo por imposição do AMS ou por opção da própria MIDlet. O AMS pode terminar a aplicação, por opção do utilizador, por exemplo, ou a MIDlet pode terminar por ela própria, passando para o estado destruída.

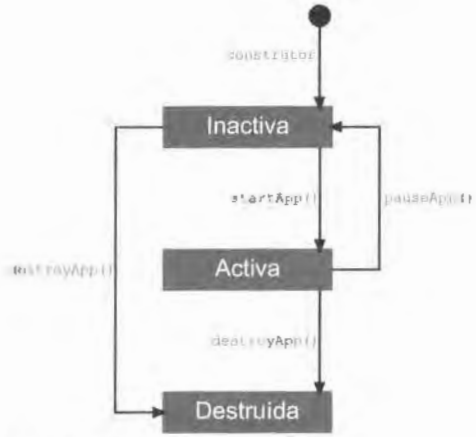


Figura 3.2 Ciclo de vida de uma MIDlet.

3.4.2. Atributos das MIDlets

O ficheiro JAR contém um ficheiro de manifesto que descreve a MIDlet Suite e cada uma das MIDlets presentes. Esse manifesto define uma série de atributos que são usados pelo AMS do dispositivo para identificar e instalar a MIDlet Suite. Para além do ficheiro de manifesto, incluído no ficheiro JAR, existe um outro ficheiro com informação sobre a MIDlet Suite: o descritor da aplicação (*Java Application Descriptor*), ou ficheiro JAD. Este ficheiro acompanha o JAR mas não está colocado no seu interior. A lista de atributos predefinidos que descrevem a MIDlet Suite é a seguinte[7]:

`MIDlet-Name`[mjad] O nome da MIDlet Suite.

`MIDlet-Version`[mjad] O número de versão da MIDlet Suite. O formato do número é maior.menor.micro, e.g., 1.0.1. O número de versão é usado pelo AMS para determinar se é um *upgrade* ou instalação e também para informar o utilizador.

`MIDlet-Vendor`[mjad] O nome da organização que fornece a MIDlet Suite.

`MIDlet-Icon`[mjad] O nome de um ficheiro PNG usado como ícone da MIDlet Suite.

`MIDlet-Description`[mjad] A descrição da MIDlet Suite.

`MIDlet-Info-URL`[mjad] Um URL que aponta para mais informação que descreve a MIDlet Suite.

7. Existem mais atributos específicos de algumas API e outros relativos a segurança. Esses atributos serão introduzidos à medida que for descrevendo essas API.

MIDlet-<n>[jad] O nome, ícone e classe principal da MIDlet número <*n*> no ficheiro JAR, separados por vírgulas. <*n*> tem de começar em 1 e devem ser usados números consecutivos.

MIDlet-Jar-URL[jad] O URL de onde o ficheiro JAR pode ser descarregado.

MIDlet-Jar-Size[jad] O número de bytes ocupados pelo ficheiro JAR.

MIDlet-Data-Size[mf,jad] O número mínimo de bytes de armazenamento persistente que a MIDlet necessita. O valor por omissão é zero.

MicroEdition-Profile[mf] O perfil Java ME necessário. Por exemplo, "MIDP-2.0".

MicroEdition-Configuration[mf] A configuração Java ME necessária. Por exemplo, "CLDC-1.0".

MIDlet-Install-Notify[jad] O URL para o qual um pedido HTTP POST é enviado para relatar o resultado da instalação. O URL não pode exceder os 256 caracteres.

MIDlet-Delete-Notify[jad] O URL para o qual um pedido HTTP POST é enviado para relatar que a MIDlet Suite foi apagada do dispositivo. O URL não pode exceder os 256 caracteres.

MIDlet-Delete-Confirm[jad] A mensagem que o sistema mostra ao utilizador para confirmar a eliminação da MIDlet Suite.

Os atributos sinalizados com [mf] podem ser utilizados no ficheiro de manifesto e os sinalizados com [jad] podem ser utilizados no descritor da aplicação. Alguns atributos são obrigatórios no manifesto e outros no descritor da aplicação. Os atributos obrigatórios do manifesto são:

- `MIDlet-Name`
- `MIDlet-Version`
- `MIDlet-Vendor`
- `MIDlet-<n>`
- `MicroEdition-Profile`
- `MicroEdition-Configuration`

Os atributos obrigatórios do descritor da aplicação são:

- `MIDlet-Name`
- `MIDlet-Version`
- `MIDlet-Vendor`
- `MIDlet-Jar-URL`
- `MIDlet-Jar-Size`

Como se pode verificar, alguns atributos estão repetidos no manifesto e no descritor da aplicação. Esses atributos devem coincidir, caso contrário o AMS não deve permitir a instalação do ficheiro JAR.

Para além dos atributos predefinidos, o programador pode incluir, no descritor da aplicação, atributos específicos da MIDlet Suite. Os nomes desses atributos não podem, no entanto, começar por "MIDlet" uma vez que esses nomes estão reservados.

Os atributos cujo nome começa por "MIDlet" e os definidos pelo programador podem ser acedidos pela MIDlet através do método `getAppProperty(nome-Atributo)`, da classe `MIDlet`.

O objectivo principal do ficheiro JAD é permitir ao dispositivo decidir se tem capacidade para executar a aplicação antes de a descarregar. Se a MIDlet Suite for demasiado grande para a capacidade de memória do dispositivo ou se a MIDlet necessitar de uma versão MIDP mais recente do que a presente no dispositivo não faz sentido descarregar o ficheiro JAR, por exemplo.

3.5. Distribuição *Over The Air*

Uma das grandes vantagens do uso do Java em telemóveis é o facto de os utilizadores poderem instalar novas aplicações nos seus dispositivos de forma relativamente simples.

A especificação MIDP define uma forma normalizada de distribuição e instalação de MIDlets – o chamado *Over The Air User Initiated Provisioning* – ou simplesmente OTA.

De uma forma geral, a especificação OTA define que os dispositivos devem fornecer uma forma para o utilizador descobrir MIDlets que possam ser instaladas. Isto pode ser feito através do *browser* residente no dispositivo ou através de uma aplicação própria. Através desta aplicação o utilizador pode escolher a MIDlet que pretende instalar e iniciar o processo de instalação. A transferência dos ficheiros é feita via HTTP 1.1.

O processo típico para distribuir uma MIDlet é o seguinte:

1. Colocar o ficheiro JAR e o ficheiro JAD num servidor Web.
2. Colocar um *link* para o JAD numa página HTML.

A instalação é feita acedendo à página HTML com o *browser* do dispositivo e activando o *link*. O gestor de aplicações irá descarregar o ficheiro JAD e perguntar ao utilizador se pretende instalar a MIDlet. Em caso afirmativo, o ficheiro JAR será descarregado (o URL do JAR está definido num atributo do JAD), se o gestor de aplicações determinar que o dispositivo tem capacidade para instalar e executar a MIDlet, dada a descrição feita no JAD.

A instalação pode ser bem ou mal-sucedida e o fornecedor da MIDlet pode ter interesse em conhecer o relatório de estado da instalação. Se for este o caso, o serviço pode especificar URL no descritor da aplicação que devem ser usados pela aplicação de instalação para reportar o estado final da instalação (atributos `MIDlet-Install-Notify` e `MIDlet-Delete-Notify`).

3.6. Segurança

Para além dos mecanismos de segurança implementados pela CLDC, o perfil MIDP define ainda mais um esquema de segurança.

Este esquema baseia-se na noção de operações sensíveis e permissões relativas a essas operações. Um exemplo de operação sensível é o estabelecimento de uma ligação à rede.

Quando uma aplicação tenta executar uma operação sensível, a implementação MIDP verifica se a permissão necessária existe. Se existir, a operação é efectuada, caso contrário é gerada uma excepção.

3.6.1. Domínio de protecção

Uma aplicação possui determinada permissão consoante o *domínio de protecção* em que se encontra.

Um domínio de protecção é caracterizado por dois factores:

- conjunto de permissões concedidas;
- critério pelo qual uma aplicação entra no domínio de protecção.

O conjunto de permissões concedidas define permissões de dois tipos: permissões automáticas (*allowed permissions*) e permissões de utilizador (*user permissions*). As permissões automáticas são concedidas automaticamente à aplicação, ou seja, uma operação sensível que necessite de uma permissão automática é imediatamente executada. As permissões de utilizador são concedidas apenas após consulta do utilizador, que pode negar ou conceder a permissão. Neste último caso, a execução da operação sensível é adiada até a implementação perguntar ao utilizador se pretende dar autorização.

As permissões de utilizador podem ser de três tipos consoante o período de tempo que a implementação recorda a resposta do utilizador:

Oneshot A implementação não guarda a resposta do utilizador. De cada vez que a permissão for necessária, o utilizador terá de ser consultado.

Session A resposta do utilizador é guardada durante o resto da execução da MIDlet. Se o utilizador deu permissão para executar aquela operação, então ela irá ser permitida até a aplicação terminar, sem ser necessário voltar a consultar o utilizador.

Blanket A resposta é guardada até a MIDlet Suite ser desinstalada.

Regra geral, as implementações actuais definem apenas dois domínios de protecção:

untrusted Nenhuma permissão é concedida automaticamente. Todas as operações sensíveis têm de ser autorizadas pelo utilizador.

trusted Todas as permissões são concedidas automaticamente. Para uma MIDlet ser associada a este domínio de protecção é necessário que o código tenha sido assinado digitalmente através de um certificado emitido por uma autoridade de certificação reconhecida pelo dispositivo.

No entanto, a especificação não impede que outros existam apenas obriga a que o domínio *untrusted* esteja definido.

3.6.2. Operações sensíveis

As aplicações que necessitam de executar operações sensíveis devem indicar esse facto através de atributos do descritor da aplicação (e no manifesto), enumerando as permissões necessárias.

As permissões existentes são:

- `javax.microedition.io.Connector.http`
- `javax.microedition.io.Connector.socket`
- `javax.microedition.io.Connector.https`
- `javax.microedition.io.Connector.ssl`
- `javax.microedition.io.Connector.datagram`
- `javax.microedition.io.Connector.serversocket`
- `javax.microedition.io.Connector.datagramreceiver`
- `javax.microedition.io.Connector.comm`
- `javax.microedition.io.PushRegistry`

Os nomes são muito parecidos com os nomes das classes, mas não são exactamente iguais.

Os atributos usados para definir as permissões necessárias pela aplicação são:

`MIDlet-Permissions`[mf,jad] Lista de permissões necessárias separadas por vírgula. Neste atributo devem ser listadas as permissões, sem as quais, a MIDlet não pode funcionar.

`MIDlet-Permissions-opt`[mfjad] Lista de permissões opcionais separadas por vírgula. Neste atributo devem ser listadas as permissões que não são absolutamente necessárias para o funcionamento da aplicação.

Estes atributos devem aparecer tanto no descritor da aplicação como no manifesto e devem conter exactamente as mesmas permissões.

Tal como a maioria dos atributos, estes são também meramente informativos e não são obrigatórios. A vantagem de os utilizar é permitir ao dispositivo determinar de antemão (antes de descarregar e instalar a MIDlet) que permissões a aplicação necessita. Se o dispositivo estiver prestes a instalar a aplicação num domínio de protecção que não permite alguma das permissões necessárias, não faz sentido instalá-la.

3.7. Pacotes Opcionais

Um pacote opcional é um conjunto de bibliotecas que, ao contrário de um perfil, não define um ambiente aplicacional completo. Estas bibliotecas estendem o ambiente de execução de forma a suportar capacidades que não são suficientemente universais para fazerem parte de um perfil e são sempre usadas em conjunção com uma determinada configuração, mas podem estar disponíveis para vários perfis.

À semelhança das configurações e dos perfis, é o fornecedor dos dispositivos que decide que pacotes opcionais irão estar disponíveis ao programador.

Existem vários pacotes opcionais em fase de definição através do *Java Community Process*, dos quais destaco os seguintes:

JSR-120: WMA – *Wireless Messaging API* O WMA estende o *Generic Connection Framework* permitindo às aplicações enviar e receber mensagens usando o serviço *Short Message Service* (SMS).

JSR-135: MMAPI – *Mobile Media API* O MMAPI permite a uma aplicação capturar e reproduzir conteúdo multimédia. Algumas classes deste pacote opcional foram mesmo incluídas na versão 2.0 do MIDP.

JSR-172: WSA – *Web Services* Permite que os dispositivos MIDP sejam clientes de *Web Services*.

JSR-179: *Location API* Permite escrever aplicações que fornecem serviços baseados na localização do utilizador.

JSR-184: M3G – *Mobile 3D Graphics API* Suporte para gráficos tridimensionais em MIDP.

3.8. Diferenças entre o MIDP 2.0 e o MIDP 1.0

A versão 2.0 do perfil MIDP é, como se esperava, pouco mais do que uma versão aumentada da versão 1.0 relativamente às bibliotecas disponibilizadas. As principais diferenças são:

- o mecanismo de distribuição OTA era apenas uma recomendação no MIDP 1.0. Na versão 2.0 passou a ser obrigatório;
- foram adicionadas bibliotecas multimédia para geração e reprodução de áudio e para reprodução de vídeo. No MIDP 1.0 era impossível gerar ou reproduzir som ou vídeo;
- a API de interface gráfica foi estendida com funções direccionadas para jogos;
- o esquema de segurança foi aumentado no MIDP 2.0. O acesso a funções sensíveis é protegido através de permissões definidas por domínios de protecção. O MIDP 1.0 não endereçava a questão da segurança (para além da já definida pela CLDC – o modelo "caixa de areia").

CAPÍTULO 4
O Clássico... (Olá Mundo!)

Agora que conhecemos a tecnologia com que iremos trabalhar, vamos ver um primeiro exemplo prático – o famoso "Olá Mundo". Neste capítulo apresento o ambiente de desenvolvimento da Sun – o Java ME Wireless Toolkit e descrevo o programa "Olá Mundo", indicando como organizar os ficheiros e como compilar e executar a aplicação.

4.1. As Ferramentas

Existem várias ferramentas que nos permitem desenvolver aplicações para o MIDP. A Sun tem, por exemplo, o Sun ONE Studio, Mobile Edition, mas existem muitos outros como o CodeWarrior da Metrowerks, o WebSphere Studio Device Developer da IBM, o NetBeans com o Mobility Pack, etc.

De qualquer modo, para quem está a iniciar-se no mundo da programação para dispositivos móveis, recomendo uma ferramenta muito simples da Sun: o *Java ME Wireless Toolkit* – WTK. O WTK é uma aplicação gráfica que permite criar, compilar e executar projectos MIDP. Esta ferramenta não possui, no entanto, nenhum editor de texto pelo que a escrita do código tem de ser feita noutro programa. O WTK é grátis e pode ser descarregado a partir do sítio Web da Sun, em `http://java.sun.com/products/j2mewtoolkit/`. A última versão, à data de escrita, é a versão 2.5 e está disponível para as plataformas Linux e Windows.

Eu vou usar o WTK ao longo deste livro, mas os exemplos são facilmente adaptados para qualquer outro ambiente de desenvolvimento que esteja a ser usado.

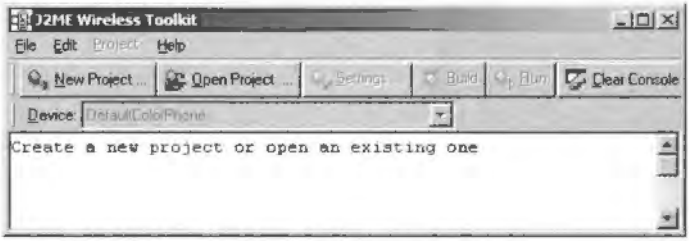


Figura 4.1 Janela principal do *Java ME Wireless Toolkit*.

A melhor maneira de aprender a trabalhar com o WTK é mesmo instalá-lo e experimentá-lo. Para se instalar o WTK é necessário ter o Java SDK, StandardEdition, versão 1.4.2. O Toolkit vem com várias aplicações, mas a principal é o KToolbar que nos dá acesso a todas as outras. A janela principal do WTK é apresentada na Figura 4.1. Esta ferramenta vem com documentação e exemplos de MIDlets que podem ser executadas imediatamente; por isso vamos experimentar abrir e executar um dos exemplos. Vamos abrir o projecto "Demos", clicando no botão [Open Project] do WTK. Depois de aberto o projecto, basta compilá-lo ([Build]) e finalmente executá-lo ([Run]). Quando se executa a aplicação é lançado o emulador do WTK. Existem vários *skins* para o emulador que podem ser escolhidos na caixa de selecção *Device* do WTK. A Figura 4.2 mostra o ecrã inicial do emulador com a lista das várias MIDlets que compõem a MIDlet Suite do projecto "Demos". Podemos usar o teclado do emulador como de um telemóvel se tratasse para escolher e executar as MIDlets. Para terminar o emulador basta clicar no botão [Fechar] da janela ou no botão de [Power] do próprio emulador. Podemos ver o código do projecto acedendo ao directório `[WTK]\apps\Demos\src`, em que `[WTK]` é o directório onde foi instalado o Toolkit (no meu caso `C:\WTK22`). A estrutura de directórios dos projectos do WTK é descrita mais à frente.

O WTK automatiza grande parte das tarefas relacionadas com a criação de MIDlets (excepto a programação propriamente dita!), como a criação automática dos descritores da aplicação, dos manifestos e do próprio JAR.

Figura 4.2 As várias MIDlets do projecto "Demos" no emulador do WTK.

4.2. O Código

A nossa primeira MIDlet é o famigerado programa "Olá Mundo":

Exemplo 4.1: `OlaMundo` – A primeira MIDlet

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.*;
public final class OlaMundo extends MIDlet {
    /**
     * Caixa de texto para a mensagem "Olá Mundo".
     */
    TextBox textBox;

    public OlaMundo() {
        textBox = new TextBox("A Primeira MIDlet", "Olá Mundo!", 255,
            TextField.ANY);
        System.out.println("MIDlet criada.");
    }

    public void startApp() {
        /* colocar a caixa de texto no ecrã. */
        Display.getDisplay(this).setCurrent(textBox);

        System.out.println("MIDlet iniciada.");
    }

    public void pauseApp() {
        System.out.println("MIDlet pausada.");
    }

    public void destroyApp(boolean unconditional) {
        System.out.println("MIDlet destruída.");
    }
}
```

Vamos então fazer uma breve descrição do código. As primeiras linhas são as importações de classes necessárias.

`javax.microedition.midlet.MIDlet`

para a classe `MIDlet`, que todas as MIDlets devem estender e

`javax.microedition.lcdui.*`

para as classes de interface com o utilizador.

Nesta MIDlet as únicas classes de interface com o utilizador usadas são a classe `TextBox` e a classe `TextField`. Estas classes serão analisadas no Capítulo 5.

Todos os métodos definidos são invocados pelo AMS e não pela MIDlet.

O que acontece quando o utilizador decide executar esta MIDlet é o seguinte:

1. O AMS inicializa a classe `OlaMundo` provocando a chamada do construtor. A caixa de texto é, por sua vez, criada e uma mensagem de texto escrita na consola (apenas no emulador).

2. Depois da classe ter sido criada, o método `startApp()` é invocado. Isto irá fazer com que a caixa de texto então criada seja exibida no ecrã do telemóvel e uma mensagem escrita na consola.
3. Se houver necessidade de suspender a execução da aplicação, devido a uma chamada recebida no telemóvel, por exemplo, o AMS irá invocar o método `pauseApp()`. No nosso caso não é necessário fazer nada. Normalmente, as MIDlets devem libertar recursos partilhados quando este método é invocado.
4. Quando o utilizador decidir terminar, o método `destroyApp()` será invocado e a MIDlet termina. Este método deve ser utilizado para a operação de finalização da aplicação, *i.e.*, guardar registos, terminar conexões, etc.

Nas próximas secções vamos ver como construir a nossa MIDlet usando dois processos diferentes: utilizando o WTK ou fazendo tudo "à mão" através da linha de comando. Mas vamos primeiro à maneira fácil.

4.3. A Maneira Fácil

Vamos então construir a nossa primeira MIDlet usando o WTK. A primeira coisa a fazer é criar um novo projecto no WTK. Para isso basta arrancar o KToolbar e clicar [New Project]. Na janela que se abre – Figura 4.3 – introduzimos o nome do projecto e o nome da classe principal da MIDlet, no nosso caso será `OlaMundo`. Quando clicamos em [Create Project], abre-se automaticamente a janela de configuração do projecto onde podemos definir os atributos da MIDlet referidos na Secção 3.4.2. Esta janela pode ser chamada a qualquer momento clicando no botão [Settings]. Neste primeiro exemplo não vamos precisar de modificar nada nesta janela.

Figura 4.3 Criação de um novo projecto no KToolbar.

Agora que temos o projecto criado basta-nos inserir o código para a nossa MIDlet. Para tal, temos de criar o ficheiro `OlaMundo.java` no directório `[WTK]\apps\OlaMundo\src` e introduzir o código do exemplo.

Depois de fazermos isso só temos de compilar e executar o projecto para vermos a nossa primeira MIDlet a funcionar! A compilação através do KToolbar faz automaticamente a pré-verificação das classes, pelo que não temos de nos preocupar com este passo no desenvolvimento das aplicações. A criação dos ficheiros de manifesto e de descrição da aplicação é também feita automaticamente pelo WTK. Podem ver esses ficheiros no directório `[WTK]\apps\OlaMundo\bin`.

A estrutura de directórios dos projectos WTK é a seguinte:

bin Contém os ficheiros de manifesto, JAD e JAR.

res Directório onde se colocam os recursos externos da MIDlet, como ficheiros de texto, imagens, ícones, etc.

src O código-fonte.

lib Bibliotecas extra de que a nossa MIDlet necessite.

4.4. A Maneira Difícil

Podemos usar o WTK para desenvolver as nossas MIDlets sem termos de nos preocupar com a compilação, pré-verificação e empacotamento das classes e sem nos preocuparmos com a criação dos ficheiros de manifesto e de descrição da aplicação. No entanto, se quisermos ter um conhecimento mais profundo de como as coisas se passam internamente, podemos fazer as coisas "à mão", i.e., usando as ferramentas de linha de comando. Conhecer o funcionamento deste processo mais aprofundadamente pode revelar-se importante quando as coisas correm mal com o WTK e temos de resolver o problema de outra forma, ou para automatizar as tarefas. Para além disso, resulta muito bem para impressionar os amigos!

Os passos seguintes serão executados no directório do projecto "OlaMundo" criado anteriormente.

4.4.1. Compilar

Para compilar a nossa MIDlet temos de "dizer" ao compilador que queremos usar as classes da configuração CLDC e do perfil MIDP ao invés das classes Java SE. O compilador utilizado é o compilador do Java SE (o WTK não vem com nenhum compilador específico para MIDP). Para tal usamos a opção "`-bootclasspath`" (as quebras de linha são apenas para tornar o comando mais legível):

```
C:\WTK22\apps\OlaMundo>javac
-bootclasspath ..\..\lib\cldcapi10.jar;..\..\lib\midpapi20.
    jar
```

```
-d tmpclasses
src\OlaMundo.java
```

A opção "`-d tmpclasses`" faz com que o compilador coloque as classes no directório `tmpclasses`, tal como o KToolbar faz automaticamente.

Se quisermos usar as classes dos pacotes opcionais é necessário colocar os nomes dos ficheiros JAR correspondentes na opção "`-bootclasspath`".

4.4.2. Pré-verificar

Como foi referido anteriormente, o processo de desenvolvimento de aplicações MIDP implica mais uma operação: a pré-verificação das classes. Isto é feito através do comando `preverify` do WTK:

```
C:\WTK22\apps\OlaMundo>..\..\bin\preverify.exe
-classpath ..\..\lib\cldcapi10.jar;..\..\lib\midpapi20.
    jar
-d classes tmpclasses
```

As classes pré-verificadas irão ficar no directório `classes`.

4.4.3. Manifesto e descritor da aplicação

As MIDlets necessitam de um ficheiro de manifesto com a descrição da MIDlet dentro do JAR. Os atributos seguintes são obrigatórios e devem ficar no ficheiro `manifest.mf`:

```
MIDlet-Name: Ola Mundo
MIDlet-Version: 1.0.0
MIDlet-Vendor: jorgecardoso.org
MIDlet-1: Ola Mundo, OlaMundo.png, OlaMundo
MicroEdition-Profile: MIDP-2.0
MicroEdition-Configuration: CLDC-1.0
```

Todos os atributos devem estar numa linha terminada por um *linefeed*, mesmo o último, ou seja, o ficheiro de manifesto deve ter a última linha em branco. Este ficheiro será depois empacotado no ficheiro JAR juntamente com as classes da MIDlet.

O descritor da aplicação repete alguma da informação do manifesto e contém mais alguma que interessa no caso de a distribuição da MIDlet ser feita através da Web. Devemos colocar no ficheiro `OlaMundo.jad` a seguinte informação:

```
MIDlet-1: Ola Mundo, OlaMundo.png, OlaMundo
MIDlet-Name: Ola Mundo
```

```
MIDlet-Version: 1.0.0
MIDlet-Vendor: jorgecardoso.org
MIDlet-Jar-URL: OlaMundo.jar
MIDlet-Jar-Size: 100
```

O último atributo, "`MIDlet-Jar-Size`", terá de ser actualizado depois de empacotarmos a MIDlet, pois só nessa altura saberemos o tamanho do ficheiro JAR.

4.4.4. Empacotar

Depois de termos as classes já pré-verificadas e o ficheiro de manifesto é necessário empacotá-los num ficheiro JAR:

```
C:\WTK22\apps\OlaMundo>jar
cfm OlaMundo.jar manifest.mf -C classes . -C res .
```

A opção "`-C res .`" não é necessária neste caso particular uma vez que a nossa MIDlet não utiliza recursos externos.

Depois de criarmos o JAR devemos actualizar o valor do atributo "`MIDlet-Jar-Size`" do descritor da aplicação com o tamanho, em bytes, do ficheiro.

4.4.5. Emular

Finalmente, podemos executar a nossa aplicação no emulador do WTK:

```
C:\WTK22\apps\OlaMundo>..\..\bin\emulator.exe
-Xdescriptor:OlaMundo.jad
```

Alternativamente, podemos executar a aplicação "Run MIDP Application" e seleccionar o ficheiro JAD criado anteriormente, ou, no caso do sistema Windows, podemos fazer duplo clique sobre o ficheiro JAD (que, neste caso, deve estar no mesmo directório que o JAR).

4.4.6. Executar num telemóvel real

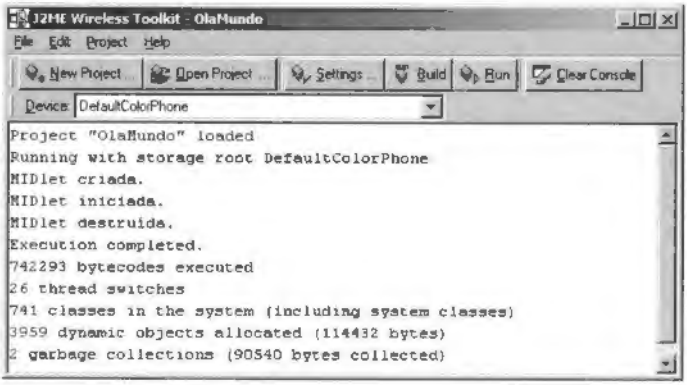
Se quisermos ver o resultado num telemóvel real, temos várias hipóteses, mas a mais prática, caso o telemóvel o permita, é transferir o ficheiro JAR via Bluetooth ou infravermelhos para o telemóvel. O processo de instalação difere de telemóvel para telemóvel. Nalguns a instalação começa automaticamente, noutros é necessário o utilizador indicar que pretende instalar a aplicação. Para instalar e executar a aplicação é necessário um telemóvel com MIDP 1.0 ou MIDP 2.0.

4.5. O Resultado

Seja qual for o método usado para criar a MIDlet, o resultado, quando se executa a MIDlet no emulador ou num telemóvel real, é o mesmo. A Figura 4.4(a) mostra o ecrã inicial da nossa MIDlet. Podemos ver a caixa de texto com a nossa mensagem na qual podemos inserir texto com o teclado do telemóvel.

A Figura 4.4(b) mostra a janela do KToolbar com saída da nossa MIDlet.

(a) Ecrã inicial



(b) A consola do emulador

Figura 4.4 MIDlet "Olá Mundo".

Podemos ver a ordem pela qual os nossos métodos foram invocados através das mensagens escritas por cada um deles. As instruções de saída utilizadas neste exemplo apenas funcionam no emulador, os telemóveis não possuem consolas de saída de texto. No final da execução o emulador imprime alguma informação relativa à execução da MIDlet, tal como memória utilizada, *bytecodes* executados, etc.

PARTE II

Programando com MIDP

CAPÍTULO 5
A Interface com o Utilizador – API de Alto Nível

A maior diferença que os programadores Java irão sentir ao programar para telemóveis é a forma como se trabalha com as API da interface com o utilizador. O modelo de programação é muito diferente do modelo utilizado no Java SE uma vez que não existe o conceito de janelas, botões, menus, etc. Isto porque o AWT seria demasiado pesado para implementar em dispositivos tão limitados como os telemóveis. Este capítulo apresenta a API de alto nível de interface com o utilizador, que permite utilizar componente típicos de interfaces gráficas, tais como campos de texto, campos de datas, listas, formulários, etc.

5.1. Introdução

O modelo de programação da interface com o utilizador da API MIDP baseia-se em ecrãs. Num dado momento apenas pode estar visível um ecrã que pode ter (e na maioria dos casos tem) *comandos* associados. Os comandos aparecem como opções ao utilizador e permitem-lhe navegar entre ecrãs.

Ao longo do livro vou usar a palavra "ecrã" com dois significados distintos. O primeiro para designar uma instância da classe `Screen`. O segundo, para designar o próprio ecrã do telemóvel. A distinção entre um e outro será feita pelo contexto.

Para tornar mais claro o funcionamento da interface vamos observar a Figura 5.1. A aplicação representada na figura é composta por três ecrãs. O primeiro ecrã é uma caixa de texto que permite a introdução de um número. Este ecrã tem dois comandos ou botões virtuais: [Sair] – para terminar a aplicação e [Seguinte] – para passar ao segundo ecrã. O utilizador actua sobre os comandos através do teclado do telemóvel (normalmente os telemóveis possuem botões junto ao ecrã que são usados para navegação). O segundo ecrã é muito semelhante ao primeiro e possui também dois comandos mas, neste caso, um para passar ao ecrã anterior e outro para ver o resultado do cálculo (terceiro ecrã). O terceiro e último ecrã é um formulário com vários campos não editáveis, *i.e.*, apenas para apresentação de resultados. Este ecrã apenas possibilita a passagem ao ecrã inicial da aplicação.

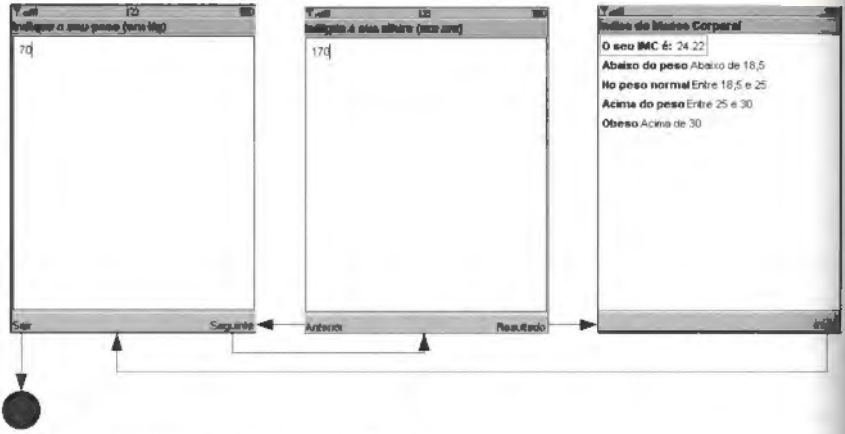


Figura 5.1 Navegação entre ecrãs de uma MIDlet.

5.2. Alto Nível e Baixo Nível

A API MIDP relacionada com a interface com o utilizador divide-se logicamente em duas API: a de alto nível e a de baixo nível.

A API de alto nível foi desenhada para fornecer um alto de nível de abstracção de forma a maximizar a portabilidade entre dispositivos. Por outro lado, esta API disponibiliza muito pouco controlo sobre a aparência da aplicação (*i.e.*, sobre o *look and feel*). As aplicações que usam a API de alto nível não controlam a aparência dos componentes gráficos, a navegação é encapsulada pela implementação e não podem aceder a dispositivos de entrada concretos (isto significa, por exemplo, que a aplicação não consegue saber se uma tecla concreta foi pressionada). A grande vantagem desta API é a garantia de portabilidade da aplicação entre dispositivos.

A API de baixo nível fornece muito pouca abstracção. As aplicações que usam esta API são responsáveis por tudo o que é desenhado no ecrã, pela detecção de eventos como o pressionar de uma determinada tecla, etc. As aplicações que usam a API de baixo nível podem não ser portáveis uma vez que esta API fornece meios para aceder a detalhes específicos de determinados dispositivos. Por exemplo, a API de baixo nível permite tratar eventos de ponteiro de ecrã; no entanto, apenas alguns dispositivos permitem interacção através deste meio. Isto não significa, contudo, que uma aplicação que usa a API de baixo nível não seja portável entre dispositivos. Usando ainda o exemplo anterior, é possível a uma MIDlet verificar se o dispositivo gera eventos de ponteiro de ecrã e, assim, adaptar-se ao dispositivo

Significa, isso sim, que o programador tem de se preocupar com a portabilidade e escrever código que permita à aplicação adaptar-se ao dispositivo concreto em que vai ser executada (ou então informar o utilizador que a aplicação necessita de um dispositivo com determinadas características para executar correctamente).

As principais classes da API de interface com o utilizador estão ilustradas na Figura 5.2. Todas as classes que representam algo que pode ser exibido no ecrã do telemóvel derivam da classe `Displayable`. As classes `Screen` e derivadas representam a API de alto nível. A classe `Canvas` representa a base da API de baixo nível. É através da implementação de subclasses de `Canvas` que o programador tem acesso ao objecto `Graphics`, que lhe permite desenhar directamente no ecrã, e a eventos de baixo nível, como o pressionar de uma determinada tecla, etc.

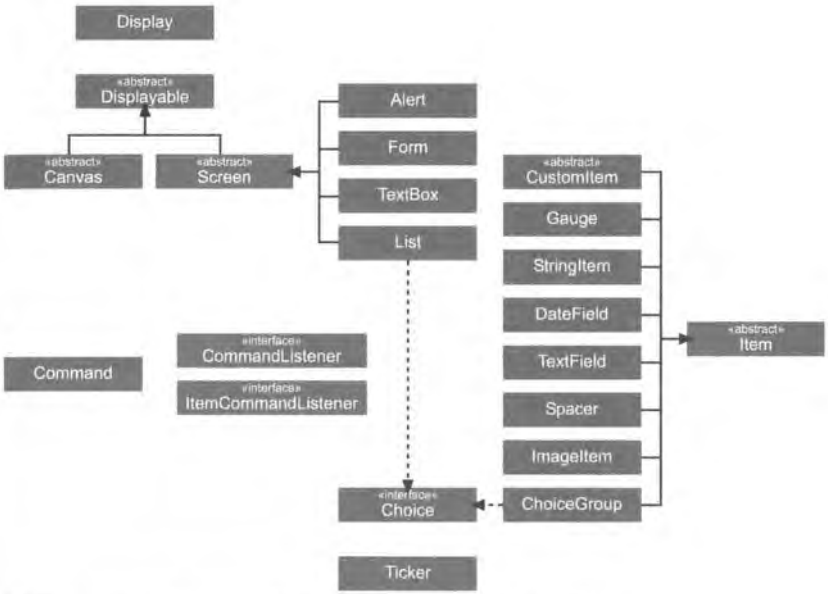


Figura 5.2 Diagrama de classes parcial da API de interface com o utilizador.

5.3. Um Ecrã Simples – Caixa de Texto

Um dos ecrãs mais básicos em MIDP é uma caixa de texto, representada pela classe `TextBox`. Vamos começar por construir então uma MIDlet com uma caixa de texto. Primeiro vamos criar o esqueleto da aplicação:

Exemplo 5.1: `MIDletCaixaTexto` – Esqueleto de uma MIDlet

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.*;

public final class MIDletCaixaTexto extends MIDlet {

    public MIDletCaixaTexto() {
    }

    public void startApp() {
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }
}
```

Para podermos modificar o conteúdo do ecrã do telemóvel precisamos de ter acesso ao objecto que gere esse conteúdo – o objecto `Display`. Tipicamente, obtemo-lo durante a inicialização da MIDlet uma vez que esse objecto, garantidamente, não muda durante a execução da aplicação:

```
public final class MIDletCaixaTexto extends MIDlet {

    private Display display;

    public MIDletCaixaTexto() {
       display = Display.getDisplay(this);
    }
```

A classe `Display` representa o gestor do ecrã e dos dispositivos de entrada do telemóvel. Existe uma instância desta classe por MIDlet, que pode ser obtida com o método estático `getDisplay(MIDlet m)`. Os objectos de interface que são mostrados no ecrã do telemóvel estão contidos num objecto do tipo `Displayable`. Em qualquer momento a aplicação pode ter no máximo um objecto `Displayable` exibido e através do qual se faz a interacção com o utilizador. Este objecto é o chamado `Displayable` corrente e é ele que recebe os eventos gerados pelos dispositivos de entrada. A aplicação define o ecrã corrente através do método `Display.setCurrent(Displayable d)`.

O `Display` também nos permite obter alguma informação sobre o dispositivo e efectuar algumas operações de baixo nível sobre o mesmo: é possível saber se o ecrã do dispositivo suporta cores, quantas suporta e quantos níveis de transparência. Dependendo da implementação, pode também ser possível fazer o dispositivo vibrar e piscar a luz de fundo do ecrã.

Agora podemos criar a nossa caixa de texto e afixá-la no ecrã:

```
public final class MIDletCaixaTexto extends MIDlet {
```

```
private Display display;
private TextBox ecrãTexto;

public MIDletCaixaTexto() {
   String titulo = "Um ecrã muito simples…";
   String mensagem = "O utilizador pode editar este ecrã";

   ecrãTexto = new TextBox(titulo, mensagem, 255, TextField.ANY);
   display = Display.getDisplay(this);
}

public void startApp() {
   display.setCurrent(ecrãTexto);
}
```

O construtor da `TextBox` é o seguinte:

```
TextBox(String title, String text, int maxSize, int
    constraints)
```

title O título do ecrã. Aparece geralmente no topo do ecrã do telemóvel.

text O texto inicial da caixa de texto. Este texto pode ser modificado dinamicamente e também pode ser editado pelo utilizador (aliás, é usado geralmente para obter dados do utilizador).

maxSize O número máximo de caracteres aceite pela caixa de texto.

constraints Um valor que define várias *flags* que restringem o tipo de dados aceite ou afectam o comportamento da caixa de texto. Estes valores estão definidos na classe `javax.microedition.lcdui.TextField`. As *flags* que restringem o tipo de dados são:

ANY O utilizador pode inserir qualquer texto.

EMAILADDR O utilizador pode introduzir um endereço de correio electrónico.

NUMERIC O utilizador pode introduzir um número inteiro.

PHONENUMBER O utilizador pode introduzir um número de telefone. Os caracteres aceites podem variar de acordo com o dispositivo e com a rede telefónica.

URL O utilizador pode introduzir um URL.

DECIMAL O utilizador pode introduzir um número decimal.

As *flags* seguintes alteram o comportamento da caixa de texto:

SENSITIVE Indica que os dados introduzidos são dados sensíveis e não devem ser guardados em tabelas de predição de texto, por exemplo.

PASSWORD Indica que o texto introduzido é confidencial e deve ser obscurecido sempre que possível. Os campos do tipo PASSWORD são tratados de forma semelhante aos do tipo SENSITIVE e não são armazenados em tabelas de predição de texto ou similares.

UNEDITABLE O utilizador não pode editar o texto.

NON_PREDICTIVE Indica que a implementação não deve usar mecanismos de predição de texto. Se, pelo contrário, esta *flag* não for usada, a implementação pode (mas não é obrigada a) usar mecanismos de predição.

INITIAL_CAPS_WORD Indica que a primeira letra de cada palavra deve ser maiúscula.

INITIAL_CAPS_SENTENCE Indica que a primeira letra de cada frase deve ser maiúscula.

Estas constantes podem ser combinadas usando o operador OR (|), e.g.,

```
TextBox("Caixa de Email Privada", "", 100,
TextField.EMAILADDR|TextField.PASSWORD)
```

para esconder a entrada de um endereço de correio electrónico.

5.4. Comandos e Eventos de Alto Nível

O exemplo anterior não é muito útil porque não tem nenhum tipo de interacção com o utilizador. Então, como obtemos *input* do utilizador? Na API de alto nível, através de comandos.

Os comandos são representados pela classe `Command`. Esta classe encapsula informação semântica sobre uma acção e não informação sobre o comportamento activado pelo comando. Traduzindo para português, um `Command` representa um "comando" e não a acção concreta que esse "comando" implica. Existem comandos EXIT – para sair da aplicação, CANCEL – para dar uma resposta negativa a um diálogo, etc, mas a acção despoletada pela selecção destes comandos fica sempre a cargo do programador. A definição de um `Command` serve apenas para a implementação escolher a melhor forma de apresentar o comando ao utilizador. A acção a executar quando um comando é activado é definida num `CommandListener` associado ao ecrã (`Displayable`). A forma de apresentação dos comandos fica completamente a cargo da implementação MIDP e depende de vários factores: da semântica associada ao comando (*i.e.*, o tipo de comando – EXIT, CANCEL, BACK, etc., e prioridade do comando), do número total de comandos a exibir, do espaço no ecrã do telemóvel, etc.

De qualquer forma, o melhor é vermos um exemplo de utilização de comandos:

Exemplo 5.2: `MIDletCaixaTextoV1` – Uso de comandos

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.*;

/**
 * MIDletCaixaTextoV1
```

```
 * Exemplifica o uso de comandos.
 */
public final class MIDletCaixaTextoV1 extends MIDlet
    implements CommandListener {

    /**
     * O gestor do ecrã.
     */
    private Display display;

    /**
     * O nosso ecrã caixa de texto.
     */
    private TextBox ecrãTexto;

    /**
     * Os comandos para terminar a aplicação, limpar a caixa
     * de texto e preenchê-la com texto predefinido.
     */
    private Command cmdSair, cmdLimpar, cmdPreencher;

    /**
     * Texto predefinido da caixa de texto.
     */
    private String mensagem = "Uma MIDlet para exemplificar o uso de
    comandos.";

    public MIDletCaixaTextoV1() {
        String titulo = "Um ecrã muito simples…";

        display = Display.getDisplay(this);
        ecrãTexto = new TextBox(titulo, mensagem, 255, TextField.ANY);

        /* vamos construir os comandos */
        cmdSair = new Command("Sair", Command.EXIT, 0);
        cmdLimpar = new Command("Limpar", "Limpar a caixa de texto",
            Command.SCREEN, 0);
        cmdPreencher = new Command("Preencher",
            "Preencher a caixa de texto", Command.SCREEN, 1);

        /* adicionar os comandos ao ecrã caixa de texto */
        ecrãTexto.addCommand(cmdSair);
        ecrãTexto.addCommand(cmdLimpar);
        ecrãTexto.addCommand(cmdPreencher);

        /* definir o CommandListener, neste caso será a própria MIDlet */
        ecrãTexto.setCommandListener(this);
    }

    public void startApp() {
        display.setCurrent(ecrãTexto);
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }

    /**
     * O método definido pelo CommandListener.
     */
    public void commandAction(Command c, Displayable d) {
```

```
            if (c == cmdSair) {
                notifyDestroyed();
            } else if (c == cmdLimpar) {
                /* limpa a caixa de texto */
                ecräTexto.setString(null);
            } else if (c == cmdPreencher) {
                ecräTexto.setString(mensagem);
            }
        }
    }
```

O exemplo anterior mostra os dois construtores da classe `Command`:

```
Command(String shortLabel, int commandType, int priority)
```

e

```
Command(String shortLabel, String longLabel,
    int commandType, int priority)
```

Os parâmetros são os seguintes:

label O "nome" do comando. É este texto que a aplicação pede que seja mostrado ao utilizador para representar o comando. Este texto pode, no entanto, ser substituído por outro mais apropriado ao tipo de comando no dispositivo actual, excepto se o tipo de comando for "`Command.SCREEN`".

commandType O tipo de comando representa o significado do comando. Há que notar que a implementação MIDP não define nenhuma acção associada a estes tipos de comandos. É perfeitamente possível programar um comando do tipo EXIT para avançar para o ecrã anterior, por exemplo. Os tipos de comandos servem apenas como pistas para a implementação os mapear da melhor forma no dispositivo em questão. Alguns dispositivos poderão ter, por exemplo, um botão específico para navegar entre ecrãs, outro para sair da aplicação... Os tipos possíveis são:

BACK Um comando de navegação que retorna o utilizador ao ecrã logicamente anterior.

CANCEL Um comando que representa uma resposta negativa a um diálogo implementado pelo ecrã actual.

EXIT Um comando usado para terminar a aplicação.

HELP Define um comando para um pedido de ajuda *online*.

ITEM Informa que o comando é específico aos itens do `Screen` ou aos elementos de uma `Choice`. Normalmente, significa que o comando se relaciona com o item seleccionado e a implementação MIDP pode usar esta pista para fornecer menus sensíveis ao contexto.

OK Representa uma resposta positiva a um diálogo implementado pelo ecrã actual.

SCREEN Define um comando específico da aplicação.

STOP Um comando para interromper um processo em curso.

priority A prioridade de um comando é mais uma pista para a implementação decidir como apresentar os comandos ao utilizador. A prioridade define a importância relativa entre comandos. Os comandos prioritários ficarão posicionados, tipicamente, de modo que o utilizador os consiga aceder directamente (*i.e.*, pressionando apenas um botão), enquanto que os comandos com menos prioridade poderão ficar agrupados num menu. Obviamente que o resultado final depende do número total de comandos e do dispositivo, *i.e.*, pode não haver espaço no ecrã para mostrar todos os comandos com alta prioridade. Valores mais altos indicam prioridades mais baixas.

longLabel Enquanto que o nome (curto) é usado normalmente para apresentar o comando como um *soft button*[1], o nome longo é usado precisamente nos casos em que o comando tem de ser exibido como um elemento num menu e, por isso, há mais espaço no ecrã. No entanto, mesmo os nomes longos deverão ser relativamente curtos (poucas palavras).

Para os comandos serem vistos pelo utilizador é necessário que estejam associados a um `Displayable`. Para isso usamos o método `addCommand()` do ecrã ao qual queremos associar o comando.

As acções executadas quando o utilizador acciona um comando são definidas no `CommandListener` do ecrã. Este objecto apenas define um método:

```
void commandAction(Command c, Displayable d)
```

que é invocado quando existe algum evento relacionado com um comando do ecrã actual. Para este objecto receber os eventos é necessário, obviamente, que a aplicação tenha definido o `CommandListener` do ecrã através de `setCommandListener()`. Este mecanismo de eventos é muito semelhante ao utilizado no Java SE; a principal diferença reside no facto de se ter adoptado uma versão *unicast* do modelo, *i.e.*, um ecrã apenas pode ter um `CommandListener` definido num determinado momento (aliás, basta reparar no nome do método que define o listener: `setCommandListener()` e não `addCommandListener()`).

No exemplo anterior, o `CommandListener` verifica qual o comando accionado comparando o parâmetro "c" com os vários comandos definidos pela aplicação. Se tivéssemos uma aplicação mais complexa, com vários ecrãs e com comandos partilhados entre ecrãs,

1. Um botão que aparece no ecrã mas está associado a um botão físico do telemóvel, normalmente posicionado directamente por baixo do ecrã e do texto que representa o comando.

poderíamos usar o `Displayable` (parâmetro "d") para determinar qual a acção a despoletar. Neste exemplo apenas temos três acções: terminar a aplicação – invocando o método `notifyDestroyed()` da `MIDlet`, limpar a caixa de texto ou preencher a caixa de texto com a mensagem predefinida:

```
public void commandAction(Command c, Displayable d) {
    if (c == cmdSair) {
       destroyApp(true);
       notifyDestroyed();
    } else if (c == cmdLimpar) {
       /* limpa a caixa de texto */
       ecrãTexto.setString(null);
    } else if (c == cmdPreencher) {
       ecrãTexto.setString(mensagem);
    }
}
```

5.5. Listas

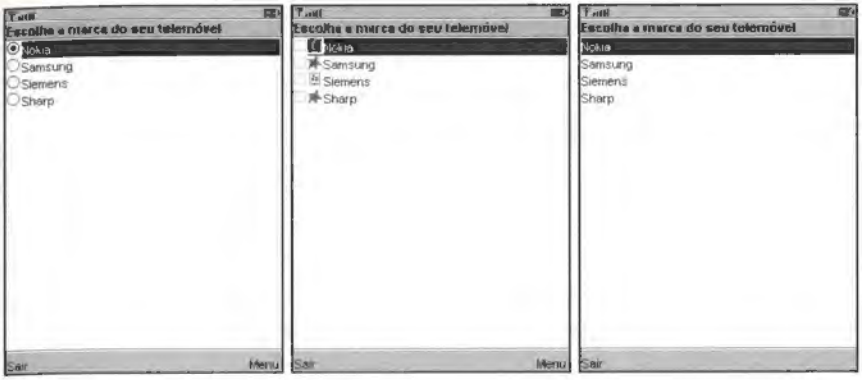
As listas são usadas para apresentar ao utilizador um ecrã com um conjunto de opções. Existem três tipos de listas em MIDP, relativamente à forma como as entradas são seleccionadas: *Exclusiva*, *Múltipla* e *Implícita*. Nas listas exclusivas apenas uma das entradas pode ser seleccionada pelo utilizador; estas listas são geralmente apresentadas como um conjunto de *radio buttons*. Nas listas múltiplas, o utilizador pode seleccionar várias opções ao mesmo tempo; normalmente são apresentadas como um conjunto de *check boxes*. As listas implícitas são um tipo especial de listas exclusivas na medida em que a aplicação é imediatamente informada quando o utilizador selecciona uma das entradas; estas listas são mais indicadas para apresentar menus de operações ao utilizador e consistem apenas num conjunto de itens de texto. A Figura 5.3 mostra os três tipos de listas.

O construtor da classe `List` tem duas variantes:

```
List(String title, int listType)
```

e

```
List(String title, int listType, String[] stringElements,
    Image[] imageElements)
```



(a) Exclusiva (b) Múltipla (c) Implícita

Figura 5.3 Os diferentes tipos de listas.

A primeira permite apenas definir o tipo de lista e o título do ecrã. A adição de elementos à lista, neste caso, tem de ser feita através dos métodos `append()`, `insert()` ou `set()`. Estes métodos permitem adicionar um elemento ao final da lista, inserir um elemento imediatamente antes de um outro e modificar um determinado elemento, respectivamente. Todos eles necessitam de uma `String`, que define o nome do elemento e, opcionalmente, de uma imagem[2] que ficará associada ao elemento.

O parâmetro "`listType`", define o tipo de lista (EXCLUSIVE, MULTIPLE ou IMPLICIT).

O segundo construtor, permite definir de uma só vez toda a estrutura da lista através do parâmetro "`stringElements`". Este parâmetro é um array com os nomes dos elementos que constituem a lista. O parâmetro "`imageElements`" é opcional (pode ser nulo) e define os ícones associados aos elementos da lista.

5.5.1. Selecção em listas exclusivas e múltiplas

As listas exclusivas e múltiplas permitem ao utilizador seleccionar opções, mas cabe à aplicação permitir ao utilizador confirmar uma selecção e decidir o que fazer com essa selecção. Para isso, é necessário que a lista tenha algum comando associado (um comando "Confirmar", por exemplo).

[2] As imagens devem ser do formato PNG e de um tamanho relativamente pequeno. O tamanho exacto das imagens dos elementos das listas depende da implementação MIDP, i.e., do telemóvel, no entanto, um tamanho 12x12 pixéis é geralmente utilizado. A criação de imagens é descrita mais à frente.

Nas listas exclusivas é possível determinar qual o elemento seleccionado através do método `List.getSelectedIndex()`. Este método devolve o índice do item seleccionado e com este índice podemos obter o nome do elemento através de `List.getString(índice)`.

Nas listas múltiplas, o método `getSelectedIndex()` não funciona. Nestes casos é necessário usar o método

`getSelectedFlags(boolean[] selectedArray_return)`

que retorna o número de elementos seleccionados e preenche o *array* passado como parâmetro com o estado de cada elemento (*true* – o elemento está seleccionado, *false* – o elemento não está seleccionado).

O exemplo seguinte ilustra um possível *CommandListener* para uma aplicação com duas listas – uma exclusiva e outra múltipla:

```
public void commandAction(Command c, Displayable d) {
    if (c == cmdConfirmar) {
       if (d == listaExclusiva) {
           List l = (List)d;
           String opção = l.getString(l.getSelectedIndex());
           System.out.println("Opção seleccionada da lista exclusiva:" +
               opção);

       } else if (d == listaMúltipla) {
           List l = (List)d;
           boolean selectedFlags[] = new boolean[4];
           int númeroOpções = l.getSelectedFlags(selectedFlags);

           System.out.println(númeroOpções +
               " opções seleccionadas da lista múltipla:");
           for (int i = 0; i < 4; i++) {
               if (selectedFlags[i]) {
                   System.out.println(l.getString(i));
               }
           }
       }
    }
}
```

5.5.2. Selecção em listas implícitas

A selecção em listas implícitas é ligeiramente diferente do que se passa com as listas de outros tipos. Neste caso não é necessário definir nenhum comando para a selecção de um item da lista. As listas implícitas têm o seu próprio comando – `List.SELECT_COMMAND`. Quando o utilizador pressiona o botão do telemóvel usado para efectuar selecções, o comando `SELECT_COMMAND` é invocado. Tudo o que a aplicação tem de fazer é comparar o comando invocado no `CommandListener`:

```
public void commandAction(Command c, Displayable d) {
    if (c == cmdSair) {
       notifyDestroyed();
```

```
        } else if (c == List.SELECT_COMMAND) {
            List l = (List)d;
            String opção = l.getString(l.getSelectedIndex());
            System.out.println("Opção seleccionada da lista implicita: " +
                opção);
        }
    }
```

5.6. Formulários

Os formulários são ecrãs que contêm uma mistura de elementos: imagens, caixas de texto (editáveis e não editáveis), campos de datas e horas, grupos de escolha, indicadores de nível e elementos personalizados. A implementação MIDP controla o posicionamento dos itens, a forma como o utilizador percorre os itens e o *scrolling*.

No que diz respeito ao posicionamento dos itens, os formulários estão organizados em linhas, todas com a mesma largura. Os itens são posicionados por ordem, primeiro horizontalmente (*i.e.*, numa linha) e depois verticalmente (*i.e.*, numa linha nova). Vários itens podem ser colocados numa mesma linha, desde que haja espaço e a aplicação não tenha dado indicações em contrário – é possível indicar que um item deve ficar isolado numa linha e o alinhamento dessa linha (esquerda, centro ou direita). As directivas de composição (*layout*) são descritas mais à frente.

O percorrer dos itens de um formulário e o *scrolling* não geram eventos visíveis para a aplicação. A aplicação apenas é informada quando o estado de um item é alterado através do método `itemStateChanged()` do *listener* definido para o formulário com o método `setItemStateListener()`.

Um formulário é representado pela classe `Form`, que tem os seguintes construtores:

```
Form(String title)
Form(String title, Item[] items)
```

O primeiro cria um formulário vazio enquanto que o segundo cria um formulário com o conjunto de itens especificado por "`items`". É possível inserir e remover itens de um formulário de uma forma muito semelhante à utilizada para inserir e remover elementos de uma lista.

Para a aplicação receber os eventos de alteração de estado dos itens (ou melhor, daqueles que permitem que o utilizador altere o seu estado), é necessário registar o *event listener*:

```
form.setItemStateListener(ItemStateListener listener)
```

e definir a acção a tomar quando o item é alterado:

```
public void itemStateChanged(Item item){
    /* Actuar de acordo com a alteração... */
}
```

O próprio `Form` pode, obviamente, ter comandos associados tal como qualquer outro ecrã. No entanto, este tipo de ecrã permite-nos também ter comandos sensíveis ao contexto, sendo que o "contexto", neste caso, é o item corrente. Os itens de um formulário são objectos `Item`. Estes objectos podem ter eles próprios comandos associados, que apenas estão visíveis para o utilizador quando o item em questão está seleccionado.

A MIDlet seguinte cria um formulário com um campo de texto (`TextField`) que permite ao utilizador editar o seu conteúdo e um campo (`StringItem`) que mostra o texto que o utilizador introduziu, de trás para a frente. O texto invertido é actualizado automaticamente sempre que o campo de texto é alterado pelo utilizador. Quando o utilizador selecciona o texto invertido é dada a possibilidade de passar o texto invertido para a caixa de texto original. A Figura 5.4 mostra o resultado da MIDlet. Reparem que o comando que permite fazer isto apenas está visível quando o texto invertido está seleccionado – é um comando associado ao item `StringItem`.

(a) TextField Seleccionado (b) StringItem Seleccionado

Figura 5.4 Comandos associados ao `Item`.

```
public class MIDletFormBasico extends MIDlet implements CommandListener,
    ItemCommandListener, ItemStateListener {

    private Display display;

    private Form form; //O nosso formulário.

    private TextField caixaTexto; //A caixa de texto.

    private StringItem textoInvertido; //O texto invertido.

    private Command cmdSair;
```

```
    //O comando para colocar o texto invertido na caixa de texto.
    private Command cmdTextoParaCaixa;

    public MIDletFormBasico() {
        display = Display.getDisplay(this);

        /* vamos construir o comando para sair da aplicação */
        cmdSair = new Command("Sair", Command.EXIT, 2);
        cmdTextoParaCaixa = new Command("Pra Caixa", "Texto para a caixa",
            Command.ITEM, 1);

        /* vamos construir o nosso formulário */
        form = new Form("Formulário Exemplo");
        /* vamos "ouvir" os eventos de itens alterados */
        form.setItemStateListener(this);

        /* construir os itens */
        caixaTexto = new TextField("Introduza texto", "", 100,
            TextField.ANY);
        textoInvertido = new StringItem("Texto Invertido", "");

        /* adicioná-los ao formulário */
        form.append(caixaTexto);
        form.append(textoInvertido);

        /* adicionar um comando ao StringItem */
        textoInvertido.addCommand(cmdTextoParaCaixa);
        textoInvertido.setItemCommandListener(this);

        /* adicionar um comando ao formulário */
        form.addCommand(cmdSair);
        form.setCommandListener(this);
    }

    public void commandAction(Command c, Displayable d) {
        if (c == cmdSair) {
            notifyDestroyed();
        }
    }

    /**
     * O método definido pelo ItemCommandListener.
     * Invocado quando um comando associado a um item
     * é activado.
     */
    public void commandAction(Command c, Item item) {
        caixaTexto.setString(textoInvertido.getText());
    }

    /**
     * O método definido pelo ItenstateListener.
     * Chamado quando algum item é alterado.
     */
    public void itemStateChanged(Item item){
        String texto = caixaTexto.getString();
        StringBuffer sb = new StringBuffer();

        /* inverter o texto */
        for (int i = texto.length() -1; i >= 0; i--) {
            sb.append(texto.charAt(i));
        }
        textoInvertido.setText(sb.toString());
    }
}
```

De seguida apresento os itens que podem ser usados num formulário, com excepção do item `CustomItem`, que será abordado no capítulo seguinte.

5.6.1. `TextField`

Um `TextField` é um campo que permite a introdução de texto por parte do utilizador. Estes campos de texto têm um tamanho máximo, que é o número máximo de caracteres que o utilizador pode introduzir (não é o tamanho da caixa de texto no ecrã). É possível restringir o tipo de dados introduzidos da mesma forma que se faz com o ecrã `TextBox`; aliás, se se recordam, as constantes utilizadas para definir essas restrições na `TextBox` são definidas na classe `TextField`. O construtor da `TextField` é:

```
TextField(String label, String text, int maxSize,
    int constraints)
```

em que "`label`" é o nome do item no formulário, "`text`" é o texto inicial do campo, "`maxSize`" é o tamanho máximo permitido em caracteres e "`constraints`" é um conjunto de *flags* que definem as restrições.

Há que ter em atenção que o tamanho máximo que a aplicação define pode ter de ser diminuído, caso não haja memória suficiente. A aplicação deve verificar o tamanho máximo realmente aplicado chamando o método `getMaxSize()`.

5.6.2. `ImageItem`

Este item permite afixar uma imagem num formulário. Os construtores deste item são:

```
ImageItem(String label, Image img, int layout,
    String altText)
ImageItem(String label, Image img, int layout,
    String altText, int appearanceMode)
```

em que "`label`" é o nome do item no formulário, "`img`" é um objecto do tipo `Image` que representa a imagem que queremos afixar, "`layout`" são as directivas de composição do item (descrito mais à frente) e "`altText`" é o texto que descreve a imagem e que pode ser usado caso não haja espaço suficiente para mostrar a imagem.

O objecto `Image` pode ser obtido de várias formas dependendo da origem da imagem. Caso esteja localizada num ficheiro dentro do pacote JAR (por exemplo, se colocarmos os ficheiros de imagem dentro do directório `res` do *Wireless Toolkit*), o objecto pode ser obtido através do método `Image.createImage(String name)`, e.g., `Image img = Image.createImage("/logo.png");`

Tipicamente, utilizamos imagens no formato PNG[3] uma vez que este formato é obrigatório na implementação MIDP. Outros formatos poderão ser suportados pelos dispositivos, mas nesse caso a aplicação deixa de ser portável.

Este item não pode ser modificado pelo utilizador pelo que não gera eventos do tipo `itemStateChanged`.

Este item tem, tal como o item `StringItem`, um atributo a que se dá o nome de *modo de aparência*. Este atributo pode ter um de três valores:

PLAIN Usado normalmente para exibir conteúdo textual ou gráfico, não interactivo.

HYPERLINK Usado quando queremos que o item seja exibido com o aspecto de uma hiperligação.

BUTTON Utilizado para fazer com que o item se assemelhe a um botão.

É preciso realçar que este atributo é uma indicação meramente visual. O facto de um item ser apresentado como uma hiperligação, por exemplo, não faz com que o *browser* do telemóvel abra uma página quando o utilizador selecciona o item. Cabe sempre à aplicação implementar os mecanismos apropriados, de forma que o comportamento se adeqúe ao aspecto do item.

5.6.3. `DateField`

O item `DateField` permite ao utilizador a introdução de datas e horas. É possível definir se queremos uma data, uma hora ou ambas através do atributo "`mode`", que pode ter um de três valores:

DATE Permite apenas a introdução de uma data (dia, mês, ano).

TIME Permite a introdução de uma hora (horas, minutos, segundos).

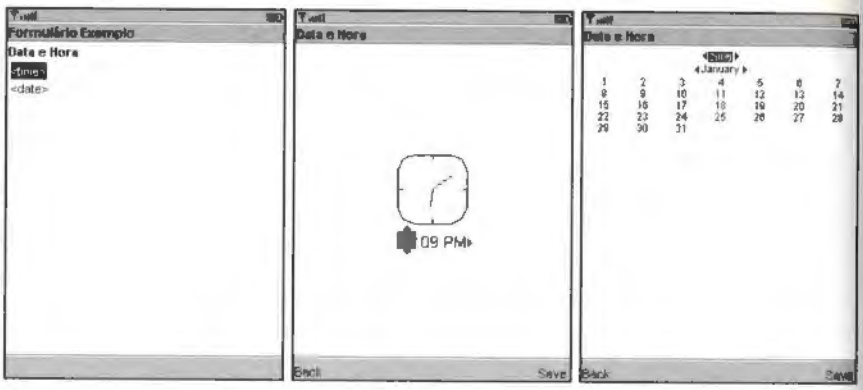
DATE_TIME Permite a introdução de uma data e hora.

Os construtores são:

```
DateField(String label, int mode)
DateField(String label, int mode, TimeZone timeZone)
```

Regra geral, as implementações MIDP fornecem interfaces para a introdução destes valores. A Figura 5.5 mostra um exemplo da introdução de datas e horas no emulador do WTK.

3. Lê-se ping.



(a) Campos no formulário (b) Introdução da hora (c) Introdução da data

Figura 5.5 O item `DateTime`.

5.6.4. `StringItem`

O item `StringItem` é dos mais básicos. Apenas permite mostrar um pequeno texto que não pode ser modificado pelo utilizador. À semelhança do `ImageItem`, tem vários modos de aparência: `PLAIN`, `HYPERLYNK` e `BUTTON`.

Os construtores da classe `StringItem` são:

```
StringItem(String label, String text)
StringItem(String label, String text, int appearanceMode)
```

5.6.5. `Gauge`

Um `Gauge` é um indicador de nível, *i.e.*, uma representação gráfica de um valor inteiro. O `Gauge` é definido através de um valor máximo e de um valor corrente, que pode ir de zero até ao valor máximo.

Um `Gauge` pode ser interactivo ou não interactivo. O tipo interactivo permite ao utilizador modificar o valor actual e pode ser utilizado, por exemplo, para implementar um controlo de volume de som. No caso do `Gauge` não interactivo, o seu valor apenas pode ser modificado pela aplicação. A Figura 5.6 mostra o aspecto de indicadores interactivos e não interactivos no emulador do WTK. A representação do valor actual no `Gauge` é apenas aproximada, isto é, se o valor máximo for, por exemplo, 100, isto não significa que o indicador irá ter resolução suficiente para diferenciar os valores 99 e 100. Será sempre dada ao utilizador a possibilidade de aumentar ou diminuir o valor em uma unidade, mas isso não significa que, visualmente, se consiga perceber essas variações.

Figura 5.6 O Gauge no emulador do WTK.

O valor actual do Gauge pode ser obtido através do método `getValue()` e pode ser alterado com `setValue()`. O construtor desta classe é:

```
Gauge(String label, boolean interactive, int maxValue,
    int initialValue)
```

O significado dos parâmetros é óbvio, pelo que não vale a pena explicá-lo.

Em MIDP, o conceito de Gauge vai um pouco mais além daquilo que descrevi acima. Mais concretamente, as versões não interactivas podem ter mais algumas variações. Um Gauge não interactivo pode ter um limite máximo definido ou indefinido. Quando o limite é definido, comporta-se como o descrito anteriormente. Quando o limite é indefinido, o Gauge funciona como um indicador de progresso. Para um limite indefinido usa-se a constante `Gauge.INDEFINITE` no parâmetro "`maxValue`" do construtor. Quando o limite é indefinido é possível passar apenas os seguintes valores no método `setValue()`:

INCREMENTAL_UPDATING Utilizado em processos dos quais não se sabe à partida quando acabam, mas em que se pode medir o progresso.
`setValue(Gauge.INCREMENTAL_UPDATING)` indica que se fez algum progresso e o indicador deve reflectir isso.

INCREMENTAL_IDLE Utilizado nas mesmas situações que o anterior para indicar que neste momento nada está a acontecer.

CONTINUOUS_RUNNING Usado em processos dos quais não se sabe quando acabam nem se consegue medir o progresso.

CONTINUOUS_IDLE Usado na situação anterior para indicar que neste momento o processo está parado.

Um exemplo de uma situação em que se pode utilizar um Gauge do tipo incremental é no *download* de um ficheiro do qual não se sabe o tamanho. Neste caso, de cada vez que a aplicação recebe, por exemplo, 1 kB de dados, pode-se invocar o método `setValue(Gauge.INCREMENTAL_UPDATING)` para dar a indicação de progresso.

Note-se que um Gauge incremental pode tornar-se num Gauge contínuo e vice-versa, apenas alterando o valor passado no método `setValue()`.

5.6.6. ChoiceGroup

O item ChoiceGroup é como uma lista que pode ser incluída num formulário. A principal diferença é que não existe o tipo IMPLICIT nos ChoiceGroup. Os ChoiceGroup podem ser EXCLUSIVE, MULTIPLE ou POPUP. Este último funciona como uma espécie de *combo box*.

Os construtores são em tudo semelhantes aos da List:

```
ChoiceGroup(String label, int choiceType)
ChoiceGroup(String label, int choiceType,
    String[] stringElements, Image[] imageElements)
```

A Figura 5.7 mostra um formulário com os três tipos de ChoiceGroup.

Figura 5.7 Os três tipos de ChoiceGroup.

5.6.7. Spacer

Um Spacer é um item especial cujo único propósito é ajudar a distribuir os outros itens no formulário. Basicamente, permite-nos definir um item com uma largura e altura mínimas:

```
Spacer(int minWidth, int minHeight)
```

Este item não pode ter comandos associados e a sua *label* deve ser sempre nula.

5.6.8. Composição dos formulários

A API MIDP dá pouco controlo ao programador relativamente ao aspecto dos formulários. Regra geral, os itens são posicionados da esquerda para a direita e de cima para baixo. Se o formulário tiver muitos itens, a implementação fornece um mecanismo de *scrolling* ao utilizador. Os formulários são organizados por linhas, de altura igual à altura do item mais alto.

O programador pode controlar, de forma limitada, o posicionamento horizontal e vertical dos itens através das directivas de composição. Estas directivas estão associadas ao próprio `item` e as mais importantes são:

LAYOUT_LEFT Para posicionar o item à esquerda do formulário.

LAYOUT_CENTER Para posicionar o item ao centro do formulário.

LAYOUT_RIGHT Para posicionar o item à direita do formulário.

LAYOUT_TOP Para posicionar o item alinhado verticalmente pelo topo da linha.

LAYOUT_BOTTOM Para posicionar o item alinhado verticalmente pela base da linha.

LAYOUT_VCENTER Para posicionar o item alinhado verticalmente pelo centro da linha.

LAYOUT_NEWLINE_BEFORE Para indicar que o item deve ser posicionado numa nova linha.

LAYOUT_NEWLINE_AFTER Para indicar que o próximo item a ser posicionado deve ficar numa nova linha.

LAYOUT_2 Indica que queremos composição *à la* MIDP 2.0. Se não indicarmos esta directiva a maior parte dos itens é posicionada automaticamente em linhas separadas (ao estilo do MIDP 1.0).

LAYOUT_DEFAULT Composição por omissão.

Estas directivas são indicadas através do método `setLayout()` da classe `Item`. Algumas delas são, obviamente, exclusivas, i.e., não faz sentido indicar `LAYOUT_LEFT | LAYOUT_CENTER` num mesmo item. Quando o alinhamento horizontal de um dado item é diferente do item anterior uma nova linha é criada automaticamente para esse item, mesmo que não tenha sido usada nenhuma directiva de nova linha.

O código seguinte daria como resultado o formulário da Figura 5.8:

```
/* imagem alinhada horizontalmente ao centro e verticalmente pela base */
itemImagemEsquerda = new ImageItem("Imagem Esquerda", imagens[1],
    Item.LAYOUT_DEFAULT, "ImgEsq");
itemImagemEsquerda.setLayout(Item.LAYOUT_CENTER | Item.LAYOUT_BOTTOM |
    Item.LAYOUT_2);
```

```
/* imagem alinhada horizontalmente ao centro */
itemImagemDireita = new ImageItem("Imagem Direita", imagens[0],
    Item.LAYOUT_DEFAULT, "ImgDir");
itemImagemDireita.setLayout(Item.LAYOUT_CENTER | Item.LAYOUT_NEWLINE_AFTER
    | Item.LAYOUT_2);

/* construir os StringItem*/
esquerdo = new StringItem("Layout", "Esquerda");
esquerdo.setLayout(Item.LAYOUT_LEFT | Item.LAYOUT_2);

centro = new StringItem("Layout", "Centro");
centro.setLayout(Item.LAYOUT_CENTER | Item.LAYOUT_2);

direito = new StringItem("Layout", "Direita");
direito.setLayout(Item.LAYOUT_RIGHT | Item.LAYOUT_2);

/* dois itens de data e hora na mesma linha */
itemData = new DateField("Data", DateField.DATE);
itemData.setLayout(Item.LAYOUT_LEFT | Item.LAYOUT_2);
itemHora = new DateField("Hora", DateField.TIME);
itemHora.setLayout(Item.LAYOUT_LEFT | Item.LAYOUT_2);
```

Figura 5.8 A composição dos formulários.

5.7. Alertas

Um alerta é um ecrã destinado a informar o utilizador acerca de erros e outras situações especiais. Tipicamente, um alerta permanece no ecrã durante um certo período de tempo e depois avança para o ecrã seguinte. Estes ecrãs podem mostrar texto, uma imagem ou emitir um som (aviso sonoro).

Os alertas são representados pela classe `Alert` que tem os seguintes construtores:

```
Alert(String title)
```

```
Alert(String title, String alertText, Image alertImage,
    AlertType alertType)
```

O parâmetro "`alertType`" é usado para indicar o tipo de alerta pretendido, de forma que a implementação possa tocar um som indicativo do tipo de alerta.

Os alertas podem ser colocados no ecrã do telemóvel de duas formas diferentes. A primeira é igual ao que temos vindo a utilizar até aqui, ou seja, recorrendo ao método `display.setCurrent(Displayable nextDisplayable)`. Neste caso, quando o tempo do alerta terminar, ou quando o utilizador responder ao alerta, o ecrã anterior torna a ficar visível. A alternativa é utilizar o método `display.setCurrent(Alert alert, Displayable nextDisplayable)`, que activa o alerta definido e passa para o ecrã representado por "`nextDisplayable`" quando o alerta terminar.

O tempo que o alerta permanece no ecrã é controlado através do método `setTimeout(int time)`, findo o qual o alerta desaparece. Por exemplo, o código seguinte mostra um alerta durante 3 segundos. No final desses 3 segundos, o ecrã que estava activo antes do alerta fica automaticamente visível.

```
alerta = new Alert("Alerta Vermelho!");
alerta.setTimeout(3000);
display.setCurrent(alerta);
```

Se quisermos que o alerta permaneça até o utilizador o terminar explicitamente, podemos passar o valor `FOREVER` como parâmetro ao método `setTimeout()`. Neste caso o alerta passa a ser *modal* e a implementação MIDP terá de fornecer um meio para o utilizador poder terminar o alerta (normalmente um *soft button*). Existem outras situações em que o alerta passa a ser modal, como no caso de o texto exibido ser demasiado extenso para ser apresentado sem *scrolling*. Nestes casos o alerta é transformado automaticamente em alerta modal, mesmo que a aplicação tenha definido um tempo finito para a sua exibição.

Os alertas podem, tal como qualquer outro ecrã, ter comandos e *listeners* associados. No entanto, a adição de comandos modifica o comportamento normal do alerta. Um `Alert` tem associado um comando predefinido – `DISMISS_COMMAND`. Se a aplicação adicionar um comando ao alerta, o comando `DISMISS_COMMAND` é automaticamente removido. Se a aplicação adicionar dois ou mais comandos ao alerta, este transforma-se num alerta modal, ou seja, é necessária a intervenção do utilizador para o terminar, mesmo que a aplicação tenha definido um *timeout* finito. Se não fizermos mais nada, estes comandos comportam-se da mesma forma, i.e., quando o utilizador seleccionar qualquer um deles o alerta desaparece. No entanto, se tiver sido adicionado apenas um comando, este comporta-se como o comando predefinido – o único efeito visível neste caso é o nome do comando no ecrã do telemóvel, que passa a ser o que o programador tiver definido.

Os alertas têm também predefinido um `CommandListener` que pode ser substituído por outro definido pela aplicação. Contudo, é necessário cuidado ao redefinir este *listener* uma vez que se o fizermos somos obrigados a implementar o mecanismo para avançar o ecrã, *i.e.*, se redefinirmos o *listener*, o alerta deixa de avançar automaticamente para o ecrã seguinte (ou o anterior, consoante a forma como foi chamado).

O exemplo seguinte mostra como implementar um alerta com dois comandos e um *listener* próprio. Note-se que a invocação do método `setTimeout()` serve apenas para mostrar que o alerta é criado com um *timeout* mas é transformado em modal, apesar desse *timeout*.

Exemplo 5.3: `MIDletAlert` – Alertas

```
public class MIDletAlert extends MIDlet implements CommandListener {

    private Display display;

    private TextBox caixaTexto; //O ecrã inicial.
    private Alert alerta; //O nosso alerta.

    private Command cmdSair; // Sai da aplicação.
    private Command cmdAlerta; // Inicia o alerta.
    private Command cmdOk, cmdCancel; // Os comandos do alerta.

    public MIDletAlert() {
       display = Display.getDisplay(this);

       /* vamos contruir os comandos */
       cmdSair = new Command("Sair", Command.EXIT, 2);
       cmdAlerta = new Command("Alerta", Command.SCREEN, 1);

       cmdOk = new Command("OK", Command.SCREEN, 1);
       cmdCancel = new Command("Cancelar", Command.SCREEN, 1);

       caixaTexto = new TextBox("Exemplo Alerta",
           "Active o comando Alerta para começar",100, TextField.ANY);

       caixaTexto.addCommand(cmdSair);
       caixaTexto.addCommand(cmdAlerta);
       caixaTexto.setCommandListener(this);

       /* vamos construir o nosso alerta */
       alerta = new Alert("Alerta Vermelho!");
       alerta.setTimeout(3000);
       alerta.addCommand(cmdOk);
       alerta.addCommand(cmdCancel);
       alerta.setCommandListener(this);
    }

    public void commandAction(Command c, Displayable d) {
       if (c == cmdSair) {
           notifyDestroyed();
       } else if (c == cmdAlerta) {
           display.setCurrent(alerta);
       } else if (c == cmdOk) {
           display.setCurrent(caixaTexto);
           System.out.println("Alerta: Ok");
       } else if (c == cmdCancel) {
           System.out.println("Alerta: Cancel");
```

```
            display.setCurrent(caixaTexto);
        }
    }

    public void startApp() {
        display.setCurrent(caixaTexto);
    }
}
```

Os alertas podem ainda ter associado um indicador de actividade, ou seja, um `Gauge`. Esta associação é feita através do método `setIndicator(Gauge indicator)`. No entanto, o `Gauge` utilizado tem algumas restrições:

- tem de ser não-interactivo;
- não pode estar associado a mais nenhum ecrã (`Alert` ou `Form`);
- não pode ter comandos;
- não pode ter um `ItemCommandListener`;
- não pode ter *label*, *i.e.*, a *label* deve ser nula;
- os valores de altura e largura preferidos devem estar desbloqueados, *i.e.*, a aplicação não deve ter invocado o método `setPreferredSize()`; e
- a directiva de composição deve ser `LAYOUT_DEFAULT`.

5.8. *Tickers*

Os *tickers*[4] são pequenas faixas de texto continuamente a rolar no ecrã do telemóvel. Não são propriamente ecrãs, mas são sempre associados a um (ou vários) ecrã. A Figura 5.9 mostra um exemplo de um *ticker*.

4. O nome vem de *Tickertape* – uma tira comprida de papel produzida pelos telégrafos.

Figura 5.9 Exemplo de um `Ticker` sobre uma `TextBox`.

Para associar um *ticker* a um ecrã basta chamar o método `setTicker (Ticker ticker)` desse ecrã. O construtor da classe `Ticker` é apenas `Ticker(String str)`, em que "`str`" é o texto que irá aparecer. Este texto pode conter quebras de linha, embora sejam usadas apenas como separadores (o texto não aparece em múltiplas linhas).

CAPÍTULO 6
A Interface com o Utilizador – API de Baixo Nível

A API de baixo nível relativa à interface com o utilizador permite ao programador ter mais controlo sobre o aspecto gráfico da aplicação e sobre a interacção com o utilizador. O uso desta API reflecte-se, basicamente, em duas classes: a `Canvas`, que permite definir um ecrã, e a classe `CustomItem`, que permite definir um item para ser usado num formulário. Estas classes permitem-nos desenhar tudo o que quisermos no ecrã do telemóvel, desde texto, imagem, linhas, polígonos, etc. Neste capítulo iremos ver como utilizar esta API.

6.1. O *Canvas*

A classe `Canvas` é o centro da API de baixo nível e permite-nos implementar ecrãs totalmente personalizados. Esta classe deriva da classe `Displayable` e, por isso, pode ser usada como um qualquer outro ecrã. Isto significa que uma aplicação pode misturar ecrãs de alto nível com ecrãs `Canvas`. Por exemplo, um ecrã do tipo Lista pode ser utilizado para apresentar uma lista de ruas e um `Canvas` para apresentar o mapa da cidade.

Para além de permitir desenhar qualquer tipo de gráfico, o `Canvas` permite também obter *input* de baixo nível do utilizador. O `Canvas` define métodos que são invocados quando o utilizador pressiona uma tecla, ou quando utiliza o ponteiro (no caso do dispositivo possuir um ecrã táctil).

O único método que é necessário implementar quando se cria uma subclasse de `Canvas` é o método `paint()`. Tal como no Java SE, este método recebe como parâmetro um objecto `Graphics` que nos permite desenhar primitivas gráficas no ecrã. O código seguinte ilustra o uso do `Canvas`.

Exemplo 6.1: `EcraRGBCanvas` – Exemplo de um canvas básico

```
[Ficheiro EcraRGBCanvas.java]
import javax.microedition.lcdui.*;

public class EcraRGBCanvas extends Canvas {
    int altura, largura;
    int larguraterço;

    public EcraRGBCanvas() {
    }

    public void paint(Graphics g) {
```

```
            /* obter algumas medidas */
            largura = getWidth();
            altura = getHeight();
            larguraterço = largura/3;

            /* tira branca */
            g.setColor(255, 255, 255);
            g.fillRect(0, 0, larguraterço, altura);
            /* tira cinzenta */
            g.setColor(127, 127, 127);
            g.fillRect(larguraterço, 0, larguraterço, altura);
            /* tira preta */
            g.setColor(0, 0, 0);
            g.fillRect(largura-larguraterço, 0, larguraterço, altura);
        }
    }
```

[Ficheiro EcraRGB.java]

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.*;

public final class EcraRGB extends MIDlet {

    private EcraRGBCanvas ecrã;

    public EcraRGB() {
        ecrã = new EcraRGBCanvas();
    }

    public void startApp() {

        Display.getDisplay(this).setCurrent(ecrã);
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }
}
```

O resultado da execução desta MIDlet seria o da Figura 6.1.

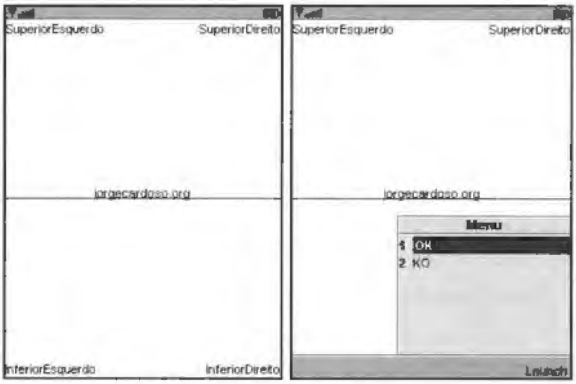
Figura 6.1 Um ecrã personalizado.

Normalmente, os ecrãs não ocupam toda a área disponível do ecrã do telemóvel.

Existem áreas reservadas para os indicadores de rede e bateria e espaço para os comandos, por exemplo. Para além disso, algum espaço é necessário para mostrar o título do ecrã e o *ticker*, caso esteja algum definido. Existe, no entanto, uma forma de pedir que o `Canvas` ocupe a maior área disponível possível: `setFullScreenMode(boolean)`. Quando este método é invocado com o valor "`true`", o `Canvas` irá ficar no modo "ecrã inteiro". Neste modo, o título e o *ticker* do ecrã não são exibidos e os comandos poderão ser apresentados de uma forma alternativa (recorrendo a menus *pop-up*, por exemplo). Mas, mesmo neste modo, pode acontecer que algum espaço do ecrã seja reservado pela implementação para os indicadores de estado do telemóvel.

A Figura 6.2 mostra um `Canvas` em "modo inteiro" no emulador do WTK. O ecrã tem dois comandos associados, que apenas são visíveis quando se pressiona um dos botões de selecção de comandos.

O `Canvas` pode inquirir o seu tamanho através dos métodos `getWidth()` e `getHeight()` que devolvem a largura e altura, em píxeis, da área de desenho, respectivamente. Há que ter em atenção, no entanto, o momento em que estes métodos são invocados, uma vez que o tamanho do *canvas* pode ser alterado durante a execução do programa. O `Canvas` é informado de cada vez que o sistema altera o seu tamanho através do método `sizeChanged(int width, int height)`, pelo que basta redefinir este método e guardar a largura e altura para termos a certeza que sabemos sempre qual o tamanho exacto do *canvas*.



(a) Estado norma (b) Menus activados

Figura 6.2 `Canvas` no modo "ecrã inteiro".

6.2. Texto

A classe `Graphics` permite-nos desenhar texto em qualquer ponto do ecrã. De forma a manter o número de cálculos no mínimo, existe uma série de pontos de referência – pontos de âncora (*anchor points*) – que ajudam a colocar o texto na posição exacta que pretendemos.

6.2.1. Pontos de âncora

Os pontos de âncora definidos para o texto são nove e estão representados na Figura 6.3. Estes pontos são definidos através da combinação de uma constante horizontal (`LEFT`, `HCENTER`, `RIGHT`) com uma constante vertical (`TOP`, `BASELINE`, `BOTTOM`).

Figura 6.3 Os pontos de âncora.

Os métodos disponíveis para desenhar texto são os seguintes:

```
void drawString(String str, int x, int y, int anchor)
void drawSubstring(String str, int offset, int len, int x,
    int y, int anchor)
void drawChar(char character, int x, int y, int anchor)
void drawChars(char[] data, int offset, int length, int x,
    int y, int anchor)
```

São todos pequenas variantes do mesmo, em que o texto está representado de forma diferente. Os parâmetros mais importantes são:

str O texto que queremos desenhar.

x, y A posição em que queremos desenhar o texto.

anchor A posição da cadeia de caracteres relativamente ao ponto (x, y).

Para exemplificar o uso dos pontos de âncora vamos observar um pequeno exemplo:

Exemplo 6.2: **AncoraCanvas** – Desenhar texto com pontos de âncora

```
import javax.microedition.lcdui.*;

public class AncoraCanvas extends Canvas {
    int altura, largura;


    public AncoraCanvas() {
        largura = getWidth();
        altura = getHeight();
    }

    public void paint(Graphics g) {
        g.setColor(255, 255, 255);
        g.fillRect(0, 0, largura, altura);

        g.setColor(0, 0, 0);
        g.drawString("SuperiorEsquerdo", 0, 0,
            Graphics.TOP|Graphics.LEFT);
        g.drawString("SuperiorDireito", largura, 0,
            Graphics.TOP|Graphics.RIGHT);
        g.drawString("InferiorEsquerdo", 0, altura,
            Graphics.BOTTOM|Graphics.LEFT);
        g.drawString("InferiorDireito", largura, altura,
            Graphics.BOTTOM|Graphics.RIGHT);

        g.drawLine(0, altura/2, largura, altura/2);
        g.drawString("jorgecardoso.org", largura/2, altura/2,
            Graphics.HCENTER|Graphics.BASELINE);
    }

    public void sizeChanged(int w, int h) {
        altura = h;
        largura = w;
    }
}
```

Se usarmos este ecrã numa MIDlet, o resultado será o apresentado na Figura 6.4.

Figura 6.4 Alinhamento de texto.

Obviamente que o mesmo resultado poderia ter sido obtido com diferentes valores. Por exemplo, para desenhar o texto do canto superior esquerdo poderíamos ter usado:

```
g.drawString("SuperiorEsquerdo", larguraTexto, 0,
    Graphics.TOP|Graphics.RIGHT)
```

em que "`larguraTexto`" seria a medida da largura do texto afixado, que pode ser obtido através da classe `Font`. Este exemplo é pouco útil, uma vez que, no caso concreto, é mais complexo do que o original, mas serve para mostrar que recorrendo aos pontos de âncora há várias formas de obter o mesmo resultado.

6.2.2. Fontes

O aspecto do texto, *i.e.*, a fonte *face*, o estilo e o tamanho[1] do texto desenhado no `Canvas` pode ser controlado através da `Font` utilizada pelo objecto `Graphics`.

A fonte utilizada para desenhar o texto é seleccionada através do método `Graphics.setFont(Font f)`. Após invocação, todas as operações de desenho de texto irão utilizar a fonte definida.

A classe `Font` representa fontes e métricas de fontes. Esta classe não possui construtores públicos; a única forma de obter instâncias é através dos métodos estáticos:

```
Font getFont(int face, int style, int size)
Font getFont(int fontSpecifier)
```

Em MIDP, as aplicações não podem criar fontes específicas. O que podem fazer é pedir uma fonte com determinados atributos (definidos em constantes na classe `Font`). A implementação trata de fornecer uma fonte que se aproxime do pedido. Os atributos usados para especificar uma fonte são:

face A "cara" da fonte. O valor deste atributo pode ser `FACE_MONOSPACE` para uma fonte monoespaçada; `FACE_PROPORTIONAL` para uma fonte proporcional ou `FACE_SYSTEM` para a fonte de sistema.

style O estilo da fonte, *i.e.*, negrito, itálico, normal ou sublinhado. Neste caso usam-se as constantes `STYLE_BOLD`, `STYLE_ITALIC`, `STYLE_PLAIN` e `STYLE_UNDERLINED`. Ao contrário do que se passa com o atributo anterior, estas constantes podem ser combinadas.

size O tamanho da fonte. Pode ser `SIZE_SMALL`, `SIZE_MEDIUM` ou `SIZE_LARGE`.

Alternativamente, a aplicação pode pedir algumas fontes predefinidas para determinados usos, com o método `getFont(int fontSpecifier)`. O parâmetro "`fontSpecifier`"

1. E também a cor do texto, mas isso é explicado mais à frente.

pode ser FONT_INPUT_TEXT, que resulta na fonte utilizada pela implementação para desenhar texto introduzido pelo utilizador; ou FONT_STATIC_TEXT, que resulta na fonte utilizada para desenhar os conteúdos de Item e Screen.

O Exemplo 6.3 mostra a utilização de fontes. O resultado é apresentado na Figura 6.5.

Exemplo 6.3: CarasCanvas – Utilização de fontes

```
public class CarasCanvas extends Canvas {
    int altura, largura;

    Font fonteMonoPequena,
         fonteProporcionalPequena,
         fonteSistemaPequena;
    Font fonteMonoMédiaItálico,
         fonteProporcionalMédiaNegrito,
         fonteSistemaMédiaSublinhado;
    Font fonteMonoGrandeNegrito,
         fonteProporcionalGrandeItálico,
         fonteSistemaGrandeNormal;


    public CarasCanvas() {
       altura = getHeight();
       largura = getWidth();

       /* vamos obter as fontes pretendidas */
       fonteMonoPequena = Font.getFont(Font.FACE_MONOSPACE,
           Font.STYLE_PLAIN, Font.SIZE_SMALL);
       fonteProporcionalPequena = Font.getFont(Font.FACE_PROPORTIONAL,
           Font.STYLE_PLAIN, Font.SIZE_SMALL);
       fonteSistemaPequena = Font.getFont(Font.FACE_SYSTEM,
           Font.STYLE_PLAIN, Font.SIZE_SMALL);
       fonteMonoMédiaItálico = Font.getFont(Font.FACE_MONOSPACE,
           Font.STYLE_ITALIC, Font.SIZE_MEDIUM);
       fonteProporcionalMédiaNegrito = Font.getFont(Font.FACE_PROPORTIONAL,
           Font.STYLE_BOLD, Font.SIZE_MEDIUM);
       fonteSistemaMédiaSublinhado = Font.getFont(Font.FACE_SYSTEM,
           Font.STYLE_UNDERLINED, Font.SIZE_MEDIUM);
       fonteMonoGrandeNegrito = Font.getFont(Font.FACE_MONOSPACE,
           Font.STYLE_BOLD, Font.SIZE_LARGE);
       fonteProporcionalGrandeItálico = Font.getFont(Font.FACE_PROPORTIONAL,
           Font.STYLE_ITALIC, Font.SIZE_LARGE);
       fonteSistemaGrandeNormal = Font.getFont(Font.FACE_SYSTEM,
           Font.STYLE_PLAIN, Font.SIZE_LARGE);
    }

    public void paint(Graphics g) {
       /* limpar o ecrã */
       g.setColor(255, 255, 255);
       g.fillRect(0, 0, largura, altura);

       g.setColor(0, 0, 0);

       /* desenhar os textos */
       g.setFont(fonteMonoPequena);
       g.drawString("Mono Normal Pequena", largura/2, 0,
           Graphics.TOP|Graphics.HCENTER);

       g.setFont(fonteMonoMédiaItálico);
```

```
            g.drawString("Mono Itálico Média", largura/2,
                fonteMonoPequena.getHeight(), Graphics.TOP|Graphics.HCENTER);

            g.setFont(fonteMonoGrandeNegrito);
            g.drawString("Mono Negrito Grande", largura/2,
                fonteMonoPequena.getHeight() +
                fonteMonoMédiaItálico.getHeight(),
                Graphics.TOP|Graphics.HCENTER);
            [...]
        }
    }
```

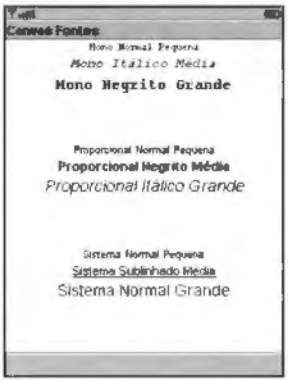


Figura 6.5 Utilização de fontes diversas.

Os atributos de uma determinada fonte podem ser obtidos com os métodos: `getFace()`, `getSize()` e `getStyle()`, que devolvem os valores das constantes para a *fonte face*, tamanho e estilo da fonte, respectivamente. No caso dos estilos existem métodos para determinar mais rapidamente se uma fonte possui determinado estilo: `isBold()`, `isItalic()`, `isPlain()` e `isUnderlined()`.

Estes métodos devolvem *true* ou *false*.

Tamanhos

A classe `Font` também nos permite obter as medidas do texto que queremos desenhar. O método `getHeight()` devolve a altura, em píxeis, de uma linha de texto desenhada com esta fonte. O valor devolvido inclui espaço extra de forma que linhas desenhadas a uma distância igual a este valor tenham espaço suficiente entre elas.

Para além da altura (independente do texto propriamente dito), é possível obter a largura de um pedaço de texto através de vários métodos:

```
int stringWidth(String str)
```

```
int substringWidth(String str, int offset, int len)
int charsWidth(char[] ch, int offset, int length)
int charWidth(char ch)
```

A última medida relativa à fonte que é possível obter é a distância entre o topo da linha e a *baseline*. Esta medida depende apenas da fonte (não depende do texto) e pode ser obtida com `getBaselinePosition()`.

6.3. Linhas, Formas e Cores

O método `Graphics.drawLine(int x1, int y1, int x2, int y2)` desenha uma linha, usando a cor e estilo de linha correntes entre os pontos $(x1, y1)$ e $(x2, y2)$. O estilo da linha pode ser alterado invocando `setStrokeStyle(int style)` com as constantes `SOLID`, para desenhar linhas sólidas, ou `DOTTED`, para linhas interrompidas.

A cor da linha também pode ser alterada, bastando invocar o método `Graphics.setColor()`. Este método tem duas variantes: `setColor(int RGB)` e `setColor(int red, int green, int blue)`. Utilizando a variante `setColor(int RGB)`, a cor é especificada através de um *int*. O valor passado é interpretado com os oito bits menos significativos representando o valor azul, os oito bits seguintes representando o valor verde e os oito seguintes o vermelho. Ou seja, é interpretado da seguinte forma: `0x00RRGGBB`. O byte mais significativo não é considerado. Na variante `setColor(int red, int green, int blue)`, a cor é dada através de três parâmetros, que variam de 0 a 255 e que definem os três componentes da cor.

Para além de linhas, é possível desenhar arcos, rectângulos e triângulos. A primitiva

```
drawArc(int x, int y, int width, int height, int startAngle,
    int arcAngle)
```

desenha um arco circular ou elíptico usando a cor e estilo de linha correntes. Para preencher o arco podemos utilizar o método

```
fillArc(int x, int y, int width, int height, int startAngle,
    int arcAngle).
```

A Figura 6.6 mostra a relação entre os vários parâmetros do desenho de um arco.

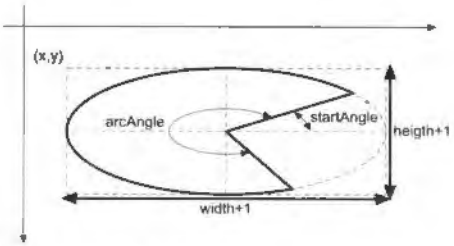


Figura 6.6 Desenho de um arco.

Os rectângulos podem ser desenhados com os cantos rectos ou arredondados. Os rectângulos com os cantos rectos são desenhados com os métodos

```
drawRect(int x, int y, int width, int height)
```

e

```
fillRect(int x, int y, int width, int height)
```

A variante com os cantos arredondados é desenhada com os métodos:

```
drawRoundRect(int x, int y, int width, int height,
    int arcWidth, int arcHeight)
```

e

```
fillRoundRect(int x, int y, int width, int height,
    int arcWidth, int arcHeight)
```

Nestes casos os cantos são desenhados como um arco de uma elipse de largura "`arcWidth`" e altura "`arcHeight`", como exemplificado na Figura 6.7.

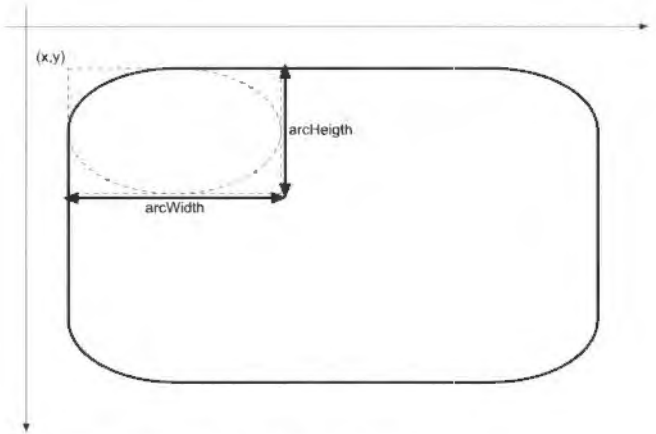


Figura 6.7 Cantos arredondados de um rectângulo.

No caso dos triângulos, apenas existe um método para preencher o triângulo:

`fillTriangle(int x1, int y1, int x2, int y2, int x3, int y3)`

Os parâmetros são os três pontos que definem os três vértices do triângulo.

O método para desenhar um triângulo não preenchido foi deixado de fora, talvez por ser fácil implementá-lo recorrendo a três `drawLine`...

Em todos estes casos, a cor da linha – no caso dos métodos *draw* – e a cor do preenchimento – no caso dos métodos *fill* – pode ser alterada recorrendo ao método `setColor()`. O mesmo acontece com a cor do texto.

Alguns dispositivos não suportam cor nos seus ecrãs, isto é, apenas são capazes de representar tons de cinzento. Nestes dispositivos a cor especificada para uma determinada operação é substituída por um tom de cinzento, consoante a cor original. É possível verificar se um dispositivo suporta cor e quantas cores diferentes suporta através dos métodos `Display.isColor()`, que devolve *true* se o dispositivo suportar cor e `Display.numColors()` que devolve o número de cores suportadas (um dispositivo a preto e branco suporta duas cores). É possível também determinar qual a cor que o dispositivo irá realmente usar através do método `Graphics.getDisplayColor(int color)`. Este método devolve a cor realmente utilizada quando indicamos que pretendemos utilizar a cor "`color`".

6.4. Imagens

É possível também, em MIDP, desenhar imagens no ecrã. Existem dois tipos de imagens: imutáveis e mutáveis. O tipo de imagem depende da forma como o objecto `Image` é criado. As imagens imutáveis são geralmente criadas através de ficheiros de imagens contidos nos recursos da MIDlet ou descarregando um ficheiro através da rede. Estas imagens não podem ser modificadas depois de terem sido criadas. As imagens mutáveis são criadas como imagens em branco que a aplicação pode alterar (a aplicação pode alterar uma imagem invocando `Image.getGraphics()` e utilizando o objecto `Graphics`, devolvido pelo método, para desenhar).

6.4.1. *Alpha blending*

As imagens imutáveis podem ter píxeis totalmente opacos, totalmente transparentes ou semitransparentes. As implementações são obrigadas a suportar píxeis totalmente opacos e píxeis totalmente transparentes. Se a implementação suportar *alpha blending*, então deve também suportar os píxeis semitransparentes nas imagens, caso contrário, estes píxeis devem ser substituídos por píxeis totalmente transparentes.

O nível de transparência na imagem resultante pode ser diferente da imagem-fonte, dependendo do número de níveis de transparência suportados pela implementação. É possível inquirir a plataforma para obter o número de níveis de transparência através do método `Display.numAlphaLevels()`.

O único formato de imagem que as implementações MIDP são obrigadas a suportar é o formato PNG, tal como especificado no documento *PNG (Portable Network Graphics) Specification, Version 1.0*.

6.4.2. Criar imagens

A classe `Image` não tem construtores. A única forma de criar objectos deste tipo é através de um dos seguintes métodos estáticos que retornam um objecto `Image`.

- `createImage(byte[] imageData, int imageOffset, int imageLength)`
  Cria uma imagem imutável a partir de um *array* com dados num formato de imagem suportado pela implementação MIDP, por exemplo, PNG. Com este método podemos ler um ficheiro de imagem, a partir da rede, por exemplo, e criar um objecto correspondente.
- `createRGBImage(int[] rgb, int width, int height, boolean processAlpha)`

Cria uma imagem imutável a partir de uma sequência de valores ARGB[2] – `0xAARRGGBB`. Se o parâmetro "`processAlpha`" for *true*, então o byte mais significativo define a opacidade do píxel, *i.e.*, `0x00RRGGBB` define um píxel completamente transparente e `0xFFRRGGBB` define um píxel completamente opaco. Valores intermédios definem píxeis semitransparentes, se a implementação o suportar.

- `createImage(InputStream stream)`
  Cria uma imagem imutável a partir de uma *stream* de dados num formato de imagem suportado. Semelhante ao método anterior, excepto que, neste caso, a fonte é um `InputStream`.
- `createImage(String name)`
  Cria uma imagem imutável a partir do nome de um ficheiro de recurso da MIDlet.
- `createImage(Image source)`
  Basicamente, cria uma cópia imutável da imagem. Se a imagem original já era imutável, então a implementação pode simplesmente retorná-la sem criar uma imagem nova.
- `createImage(Image image, int x, int y, int width, int height, int transform)`
  Cria uma imagem imutável a partir de uma região de outra imagem (imutável ou mutável). Para além disso, é aplicada uma transformação 2D, especificada no parâmetro "`transform`". As possíveis transformações são[3]:
  TRANS_NONE A região é copiada sem alteração nenhuma.
  TRANS_ROT90 A região é rodada 90 graus no sentido horário.
  TRANS_ROT180 A região é rodada 180 graus no sentido horário.
  TRANS_ROT270 A região é rodada 270 graus no sentido horário.
  TRANS_MIRROR A região é reflectida pelo eixo central vertical.
  TRANS_MIRROR_ROT90 A região é reflectida pelo eixo central vertical e depois rodada 90 graus no sentido horário.
  TRANS_MIRROR_ROT180 A região é reflectida pelo eixo central vertical e depois rodada 180 graus no sentido horário.
  TRANS_MIRROR_ROT270 A região é reflectida pelo eixo central vertical e depois rodada 270 graus no sentido horário.

2. Um *array* com informação ARGB é um *array* de inteiros em que os quatro bytes que o compõem representam os valores de *Alpha*, *Red*, *Green* e *Blue*, começando no byte mais significativo e terminando no menos significativo.
3. Estas constantes estão definidas na classe `javax.microedition.lcdui.game.Sprite`

- `createImage(int width, int height)`
  Cria uma nova imagem mutável que pode ser usada pela aplicação para desenhar.

O exemplo seguinte exemplifica o uso de duas formas para criar imagens. A primeira a partir de um ficheiro de imagem (PNG) localizado no directório de recursos da MIDlet. A segunda a partir de um *array* com dados ARGB em que se aumenta gradualmente a opacidade dos píxeis de linha para linha. O resultado é apresentado na Figura 6.8.

Exemplo 6.4: `CanvasImage` – Criar imagens a partir de ficheiro e através de *arrays*

```
import javax.microedition.lcdui.*;
import java.io.*;

public class CanvasImage extends Canvas {
    int altura, largura;

    Image imgFundo;
    Image imgGradiente;

    public CanvasImage() {

        altura = getHeight();
        largura = getWidth();

        /* criar a imagem de fundo */
        try {
            imgFundo = Image.createImage("/JavaPowered-2.png");
        } catch (IOException ioe) {
            System.err.println(ioe.getMessage());
        }

        /* criar uma imagem com um gradiente semi-transparente do tamanho
           do ecrã
        */
        int imgDados[] = new int[altura*largura];

        /* vamos usar um tom de cinza */
        int cor = 0x00666666;
        int c;
        for (int i = 0; i < altura; i++) {
            * definir a transparência de modo que a primeira linha seja
              completamente transparente e a última completamente opaca.
            */
            c = cor | ((i*255/altura)<<24);
            for (int j = 0; j < largura; j++) {
                    imgDados[i*largura +j] = c;
            }
        }
        imgGradiente = Image.createRGBImage(imgDados, largura, altura,
            true);
    }

    public void paint(Graphics g) {
        /* limpar o ecrã */
        g.setColor(255, 255, 255);
        g.fillRect(0, 0, largura, altura);
```

```
            g.drawImage(imgFundo, largura/2, altura/2,
                Graphics.HCENTER|Graphics.VCENTER);
            g.drawImage(imgGradiente, 0, 0, Graphics.TOP|Graphics.LEFT);
        }
    }
```

Figura 6.8 Imagens com transparência.

Não existe uma forma directa de criar uma imagem mutável a partir de um ficheiro de imagem, mas podemos "dar a volta" da seguinte forma:

```
Image imutável = Image.createImage("imagem.png");
Image mutável = Image.createImage(imutável.getWidth(),
    imutável.getHeight());
Graphics g = mutável.getGraphics();
g.drawImage(imutável, 0, 0, g.TOP|g.LEFT);
```

6.4.3. Desenhar imagens

A classe Graphics fornece-nos duas formas de colocar uma imagem no ecrã:

```
drawImage(Image img, int x, int y, int anchor)
drawRegion(Image src, int x_src, int y_src, int width,
    int height, int transform, int x_dest, int y_dest,
    int anchor)
```

A primeira alternativa desenha a imagem "`img`", na posição definida pelos parâmetros "`x`", "`y`" e "`anchor`". Tal como no caso do texto, também nas imagens existe o conceito de ponto de âncora. No caso das imagens a diferença é que existe mais um ponto de âncora, `VCENTER`, que corresponde ao centro vertical da imagem.

A segunda alternativa permite-nos desenhar uma região de uma determinada imagem, possivelmente aplicando uma transformação 2D.

O exemplo seguinte ilustra o uso das duas alternativas. O exemplo mostra como desenhar uma imagem no centro do ecrã e como desenhar uma região da mesma imagem aplicando uma rotação. O resultado é apresentado na Figura 6.9.

Exemplo 6.5: `CanvasImagemRegiao` – Desenhar regiões de imagens

```
import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;
import java.io.*;

public class CanvasImagemRegiao extends Canvas {
    int altura, largura;

    Image imgFundo;

    public CanvasImagemRegiao() {

        altura = getHeight();
        largura = getWidth();

        /* criar a imagem de fundo */
        try {
            imgFundo = Image.createImage("/JavaPowered-2.png");
        } catch (IOException ioe) {
            System.err.println(ioe.getMessage());
        }
    }

    public void paint(Graphics g) {
        /* limpar o ecrã */
        g.setColor(255, 255, 255);
        g.fillRect(0, 0, largura, altura);

        /* desenhar a imagem (60x90 pixeis) no centro do ecrã */
        g.drawImage(imgFundo, largura/2, altura/2,
            Graphics.HCENTER|Graphics.VCENTER);

        /* desenhar metade da imagem anterior rodada 90 graus */
        g.drawRegion(imgFundo, 30, 0, 30, 90, Sprite.TRANS_ROT90, 0, 0,
            Graphics.TOP|Graphics.LEFT);
    }
}
```

Figura 6.9 Desenhar regiões de uma imagem.

6.4.4. Duplo *buffer*

As imagens podem também ser utilizadas para implementar a técnica do duplo *buffer* para *rendering*. Em vez de desenhar directamente no ecrã, o que pode originar *flickering*, desenha-se numa imagem que é depois afixada de uma só vez no ecrã. O código seguinte exemplifica esta técnica.

```
public void paint(Graphics g) {
    if (duploBuffer == null) { // ainda não criámos o buffer
       /* criar uma imagem do tamanho do ecrã */
       duploBuffer = Image.createImage(getWidth(), getHeight());
    } else {
       Graphics gBuffer = duploBuffer.getGraphics();

       /* desenhar o que for necessário */
       gBuffer.drawLine...
    }
    g.drawImage(duploBuffer, 0, 0, Graphics.TOP|Graphics.LEFT);
}
```

Esta técnica, no entanto, nem sempre é necessária. Alguns dispositivos implementam já, de forma transparente para o programador, o duplo *buffer*. É possível saber se o nosso *canvas* tem duplo *buffer* invocando o método `Canvas.isDoubleBuffered()`.

6.5. Eventos de Baixo Nível

A classe `Canvas` é também a fonte para o uso de eventos de baixo nível. Nesta classe é possível determinar se o utilizador pressionou, largou ou manteve pressionada uma tecla e

qual a tecla pressionada. É também possível, caso o dispositivo o suporte, receber eventos de ponteiro, *i.e.*, ponteiro pressionado, arrastado e largado.

As aplicações que desejem receber eventos do teclado ou do ponteiro devem implementar os métodos associados:

```
keyPressed(int keyCode)
keyReleased(int keyCode)
keyRepeated(int keyCode)
pointerDragged(int x, int y)
pointerPressed(int x, int y)
pointerReleased(int x, int y)
```

Em MIDP não é necessário registar *event listeners* como em Java SE. Os métodos anteriores são invocados pela implementação sempre que o evento respectivo é gerado.

O evento *keyRepeated* e os eventos de ponteiro podem não ser suportados pelo dispositivo. A aplicação deve inquirir a implementação através dos métodos `hasRepeatEvents()` (*keyRepeated*), `hasPointerEvents()` (*pointerPressed* e *pointerReleased*) e `hasPointerMotionEvents()` (*pointerDragged*) para determinar se os eventos são suportados.

6.5.1. Eventos de teclas

As aplicações recebem os eventos de teclas através do códigos das teclas (*keycodes*). Cada tecla no teclado de um telemóvel tem um código diferente (excepto se duas teclas forem "sinónimos" uma da outra, caso em que terão o mesmo código). O MIDP define os códigos para algumas teclas, que correspondem às teclas de um teclado ITU-T. Estes códigos são: `KEY_NUM0`, `KEY_NUM1`, `KEY_NUM2`, `KEY_NUM3`, `KEY_NUM4`, `KEY_NUM5`, `KEY_NUM6`, `KEY_NUM7`, `KEY_NUM8`, `KEY_NUM9`, `KEY_STAR` e `KEY_POUND`. Obviamente, os teclados dos telemóveis podem ter (e normalmente têm) mais teclas do que as anteriores. Nestes casos os códigos atribuídos às teclas extra são diferentes e dependem do dispositivo. Para garantir portabilidade, as aplicações devem usar apenas as teclas-padrão.

Os códigos das teclas-padrão correspondem aos códigos Unicode para os caracteres que representam as teclas (no caso das teclas dos números, correspondem os códigos 48 a 57, em decimal). Quando o dispositivo inclui teclas extra com correspondência a caracteres Unicode, o código da tecla deve ser igual ao código Unicode. Nos restantes casos, a implementação deve atribuir códigos de valores negativos (zero é um código inválido). Isto significa que, se o código da tecla for positivo, podemos fazer uma conversão para *char* de forma a obter o carácter correspondente à tecla.

Para o utilizador é mais informativo o nome da tecla do que o seu código, pelo que existe um método que devolve o nome da tecla, dado o seu código: `getKeyName(int keyCode)`, que devolve uma `String`. O nome de uma tecla pode ser, por exemplo no caso da tecla numérica [6], simplesmente o carácter associado – "6".

O exemplo seguinte mostra um `Canvas` que desenha uma imagem e permite que o utilizador altere a posição da imagem mediante o pressionar das teclas. Neste caso, escolhi utilizar as teclas [2], [6], [8] e [4] para movimentar a imagem para cima, para a direita, para baixo e para a esquerda, respectivamente. O exemplo mostra também a utilização do método `getKeyName()` – sempre que o utilizador larga uma tecla, o nome dessa tecla é impresso na consola.

Exemplo 6.6: `CanvasEventos` – Eventos de teclas

```
import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;
import java.io.*;

public class CanvasEventos extends Canvas {
    /* As dimensões do canvas */
    int altura, largura;

    /* A imagem */
    Image imgFundo;

    /* A posição onde a imagem irá ser desenhada */
    int imgX, imgY;

    public CanvasEventos() {

      altura = getHeight();
      largura = getWidth();

      /* criar a imagem de fundo */
      try {
          imgFundo = Image.createImage("/JavaPowered-2.png");
      } catch (IOException ioe) {
          System.err.println(ioe.getMessage());
      }

      /* inicialmente a imagem está centrada no ecrã */
      imgX = largura/2;
      imgY = altura/2;

      /* determinar se o dispositivo gera "repeat events" */
      if (hasRepeatEvents()) {
          System.out.println("Canvas tem repeat events.");
      } else {
          System.out.println("Canvas não tem repeat events.");
      }
    }

    public void paint(Graphics g) {
      /* limpar o ecrã */
      g.setColor(255, 255, 255);
      g.fillRect(0, 0, largura, altura);
```

```
        /* desenhar a imagem na posição determinada por imgX e imgY */
        g.drawImage(imgFundo, imgX, imgY,
            Graphics.HCENTER|Graphics.VCENTER);
    }


    /**
     * Invocado quando o utilizador pressiona uma tecla.
     */
    public void keyPressed(int keyCode) {

        /* utilizar as teclas [2], [6], [8], e [4] para mover a imagem */
        switch(keyCode) {
            case KEY_NUM4:
                    imgX--;
                    break;
            case KEY_NUM6:
                    imgX++;
                    break;
             case KEY_NUM2:
                    imgY--;
                    break;
            case KEY_NUM8:
                    imgY++;
                    break;
        }
        repaint();
    }

    /**
     * Invocado quando o utilizador larga uma tecla.
     */
    public void keyReleased(int keyCode) {

        /* Escrever na consola o nome da tecla */
        System.out.println("Tecla Libertada: " + getKeyName(keyCode));
    }

    /**
     * Invocado quando o utilizador pressiona continuamente uma tecla
     * se o dispositivo suportar este tipo de eventos.
     */
    public void keyRepeated(int keyCode) {

        /* fazer o mesmo que no keyPressed */
        switch(keyCode) {
            case KEY_NUM4:
                    imgX--;
                    break;
            case KEY_NUM6:
                    imgX++;
                    break;
            case KEY_NUM2:
                    imgY--;
                    break;
            case KEY_NUM8:
                    imgY++;
                    break;
        }
        repaint();
    }
}
```

Game actions

No exemplo anterior fiz um mapeamento directo entre determinadas teclas e a sua função, de forma que a sua utilização parecesse natural – a tecla que move a imagem para a direita está à direita, a que move a imagem para cima está em cima, etc. No entanto, este esquema apenas funciona nos dispositivos em que a disposição das teclas está feita de determinada forma.

Para resolver este problema o MIDP oferece uma abstracção – as acções de jogos (*game actions*). O MIDP define as seguintes acções: `UP`, `DOWN`, `LEFT`, `RIGHT`, `FIRE`, `GAME_A`, `GAME_B`, `GAME_C` e `GAME_D`. Alguns códigos de teclas são mapeados para as acções de jogos pela implementação de forma que façam sentido no dispositivo. Por exemplo, as acções `UP`, `DOWN`, `LEFT` e `RIGHT` podem ser mapeadas para o *joystick* do telemóvel (se este possuir um). Desta forma o programador não tem de se preocupar com as atribuições de teclas.

O método `getGameAction(int keyCode)` é utilizado para determinar a *game action* associada com uma determinada tecla.

De notar que uma determinada *game action* pode estar associada a mais do que uma tecla, por exemplo, `LEFT` pode estar associado à tecla [4] e à tecla de navegação para a esquerda.

O código seguinte é o exemplo anterior adaptado para usar as *game actions*.

Exemplo 6.7: `CanvasEventosGame` – *Game Actions*

```
import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;
import java.io.*;

public class CanvasEventosGame extends Canvas {
    /* As dimensões do canvas */
    int altura, largura;

    /* A imagem */
    Image imgFundo;

    /* A posição onde a imagem irá ser desenhada */
    int imgX, imgY;

    public CanvasEventosGame() {

       altura = getHeight();
       largura = getWidth();

       /* criar a imagem de fundo */
       try {
           imgFundo = Image.createImage("/JavaPowered-2.png");
       } catch (IOException ioe) {
           System.err.println(ioe.getMessage());
       }

       /* inicialmente a imagem está centrada no ecrã */
       imgX = largura/2;
```

```
        imgY = altura/2;

        /* determinar se o dispositivo gera "repeat events" */
        if (hasRepeatEvents()) {
            System.out.println("Canvas tem repeat events.");
        } else {
            System.out.println("Canvas não tem repeat events.");
        }
    }

    public void paint(Graphics g) {
        /* limpar o ecrã */
        g.setColor(255, 255, 255);
        g.fillRect(0, 0, largura, altura);

        /* desenhar a imagem na posição determinada por imgX e imgY */
        g.drawImage(imgFundo, imgX, imgY,
            Graphics.HCENTER|Graphics.VCENTER);
    }

    /**
     * Invocado quando o utilizador pressiona uma tecla.
     */
    public void keyPressed(int keyCode) {

        switch(getGameAction(keyCode)) {
            case LEFT:
                    imgX--;
                    break;
            case RIGHT:
                    imgX++;
                    break;
            case UP:
                    imgY--;
                    break;
            case DOWN:
                    imgY++;
                    break;
        }
        repaint();
    }
}
```

6.5.2. Eventos de ponteiro

Alguns dispositivos possuem ecrãs tácteis e aceitam interacção através de um ponteiro (*stylus pen*). Nestes dispositivos, a implementação MIDP pode desencadear eventos de ponteiro no canvas.

É possível verificar de antemão se o dispositivo suporta este tipo de eventos através dos métodos `hasPointerEvents()` e `hasPointerMotionEvents()`. O primeiro indica se o dispositivo lança os eventos de ponteiro pressionado e libertado. O segundo indica se os eventos de ponteiro arrastado são suportados.

No que diz respeito à forma como as aplicações obtêm os eventos de ponteiro, o procedimento é exactamente igual ao dos eventos do teclado: basta implementar os métodos associados aos eventos:

```
pointerDragged(int x, int y)
pointerPressed(int x, int y)
pointerReleased(int x, int y)
```

A informação passada à aplicação é simplesmente a coordenada, em píxeis, do ponto onde o ponteiro foi pressionado ou largado (ou, no caso do arrastamento, o ponto onde se encontra no momento).

6.6. O *CustomItem*

O `CustomItem` é um item de formulário que podemos programar da forma que quisermos.

Do ponto de vista da forma como se estende a classe é muito semelhante ao `Canvas`. Redefinimos o método `paint` para podermos desenhar o item e podemos responder aos eventos do teclado e do ponteiro da mesma forma que no `Canvas`.

Existem, no entanto, algumas particularidades relacionadas com o facto de ser um item de formulário. Tal como os outros itens, o `CustomItem` tem os conceitos de tamanho mínimo e tamanho preferido. Tem também o conceito de tamanho do conteúdo, que não é mais do que o tamanho da área que contém o conteúdo propriamente dito do item. Enquanto que os tamanhos mínimo e preferido incluem o espaço ocupado pela etiqueta (*label*) e por possíveis bordas do item, o tamanho do conteúdo inclui apenas a área necessária para a apresentação do conteúdo. É a área do conteúdo que o programador define através dos métodos:

```
int getMinContentWidth()
int getMinContentHeight()
int getPrefContentWidth(int height)
int getPrefContentHeight(int width)
```

Os métodos `getMinContentWidth()` e `getMinContentHeight()` definem o tamanho mínimo que o programador acha que o item deve ter. Os métodos `getPrefContentWidth(intheight)` e `getPrefContentHeight(intwidth)` definem o tamanho preferido pelo item dada uma das dimensões. Estes métodos especificam uma das dimensões porque, quando o formulário está a ser disposto no ecrã, as dimensões máximas são definidas pelos itens maiores. Desta forma, a implementação pode inquirir o item para saber qual o tamanho preferido dado que uma das dimensões já está "bloqueada".

Para desenhar o item, as subclasses devem implementar o método `void paint(Graphics g, int w, int h)`. Os parâmetros "w" e "h" correspondem à largura e altura do item, respectivamente. Estes valores são passados neste método apenas por conveniência, uma vez que podiam ser obtidos através do método de rechamada (*callback*) `void sizeChanged(int w, int h)`, que a implementação invoca sempre que o tamanho do item é alterado.

O exemplo seguinte mostra como estender a classe `CustomItem` para implementar um item muito simples (e inútil!). O resultado é apresentado na Figura 6.10, em que o item foi colocado num formulário juntamente com mais dois *StringItem*.

Exemplo 6.8: **Gradiente** – Um **CustomItem** básico

```
import javax.microedition.lcdui.*;

public class Gradiente extends CustomItem {

    /**
     * As dimensões do item.
     */
    private int largura, altura;

    public Gradiente(String title) {
        super(title);
    }

    public void paint(Graphics g, int w, int h) {
        for (int i = 0; i < largura; i++) {
            g.setColor(i*2, i*2, i*2);
            g.drawLine(i, 0, i, h);
        }
    }

    public int getMinContentWidth() {
        return 100;
    }

    public int getMinContentHeight() {
        return 60;
    }

    public int getPrefContentWidth(int largura) {
        return getMinContentWidth();
    }

    public int getPrefContentHeight(int altura) {
        return getMinContentHeight();
    }

    public void sizeChanged(int w, int h) {
        this.altura = h;
        this.largura = w;
    }

    public void showNotify() {
        System.out.println("Visivel");
    }
```

```
    public void hideNotify() {
        System.out.println("Não Visivel");
    }
}
```

Figura 6.10 Um `CustomItem` simples.

6.6.1. Notificações de tamanho e visibilidade

O exemplo anterior implementa dois métodos `showNotify()` e `hideNotify()` que são invocados pela implementação quando o item se torna visível no ecrã (mesmo que não totalmente) e quando o item deixa de estar visível. Uma vez invocado o `showNotify()`, o método `paint()` poderá ser chamado para desenhar o item. Depois de `hideNotify()` ser invocado, não serão feitas mais chamadas a `paint()`, até `showNotify()` ser chamado novamente.

Quando o formulário está a ser disposto no ecrã, o tamanho do item pode ter de ser alterado para se adaptar à composição definida. Nestas situações, i.e., sempre que o tamanho do item é alterado pela implementação, o método `sizeChanged()` é invocado de forma que o item tenha disso conhecimento.

O próprio item pode também decidir que o seu tamanho tem de ser alterado. Nesse caso, deve invocar `invalidate()`. Isto irá informar a implementação que o formulário deve ser redisposto no ecrã e adaptado à nova dimensão do item.

6.6.2. Eventos

Tal como no caso do `Canvas`, o `CustomItem` pode também receber eventos de baixo nível do teclado e do ponteiro, implementando os métodos descritos anteriormente para o caso do `Canvas`.

No entanto, no caso do `CustomItem`, não é garantido que a implementação transmita estes eventos ao item. Para saber que tipo de eventos são suportados é necessário invocar `getInteractionModes()` que irá devolver uma combinação das seguintes constantes:

KEY_PRESS Indica que o item recebe eventos de tecla pressionada (`keyPressed()`).

KEY_RELEASE Indica que o item recebe eventos de tecla libertada (`keyReleased()`).

KEY_REPEAT Indica que o item recebe eventos de tecla continuamente pressionada (`keyRepeated()`).

POINTER_DRAG Indica que o item recebe eventos de ponteiro arrastado (`pointerDragged()`).

POINTER_PRESS Indica que o item recebe eventos de ponteiro pressionado (`pointerPressed()`).

POINTER_RELEASE Indica que o item recebe eventos de ponteiro libertado (`pointerReleased()`).

TRAVERSE_HORIZONTAL Indica que o item pode ser atravessado horizontalmente.

TRAVERSE_VERTICAL Indica que o item pode ser atravessado verticalmente.

Os dois últimos casos são descritos mais à frente. Para determinar que o item suporta eventos de tecla libertada podemos então fazer algo do género:

```
suportaTeclaLibertada = ((getInteractionModes() & KEY_RELEASED) != 0);
```

No caso de existir suporte para eventos de teclado, apenas é garantido que os códigos `KEY_NUM0` a `KEY_NUM9`, `KEY_POUND` e `KEY_STAR` são suportados. Os códigos relativos a outras teclas podem ou não ser transmitidos ao item dependendo da implementação (alguns códigos não podem ser usados porque as teclas a que correspondem são usadas para percorrer o formulário). Quando existe suporte para eventos de teclas então está também garantida a possibilidade de utilizar as acções de jogos.

As aplicações que utilizam eventos de teclado nos `CustomItem` não devem assumir que irão receber um evento de tecla pressionada por cada evento de tecla libertada ou continuamente pressionada. Nos itens, pode acontecer que o primeiro evento seja de tecla libertada, por exemplo, se a tecla tinha sido pressionada antes do item estar visível.

6.6.3. Atravessamento interno

Nos formulários MIDP existem os conceitos de atravessamento (*traversal*) e atravessamento interno a um item. O atravessamento de um formulário é, basicamente, o acto de percorrer os itens que compõem o formulário. No entanto, alguns itens têm estruturas com-

plexas que implicam que os próprios itens possam ser também percorridos – a isto chama-se atravessamento interno.

Esta ideia foi implementada, genericamente, da seguinte forma: o utilizador percorre o formulário alterando o item focado. Quando o item focado é o nosso *CustomItem*, o método `traverse()` é invocado. Se o nosso item necessitar de atravessamento interno (se o conteúdo for simples, é provável que não faça sentido usar atravessamento), este método deve retornar *true*. Neste caso, as acções de atravessamento feitas pelo utilizador são passadas ao `CustomItem`, novamente através do método `traverse()` (este método possui alguns parâmetros que vamos ver mais à frente), indicando, entre outras coisas, a direcção em que o utilizador está a percorrer o item (para cima, para baixo, esquerda ou direita). O item pode utilizar esta informação para percorrer os seus próprios elementos. Quando o utilizador chega ao "fim" do item, o método `traverse()` deve retornar *false*; só nesta altura o foco irá passar para outro item do formulário.

O método `traverse()` tem a seguinte assinatura:

```
boolean traverse(int dir, int viewportWidth,
    int viewportHeight, int[] visRect_inout)
```

A descrição dos parâmetros deste método é feita a seguir. Alguns destes parâmetros têm, pelo menos do meu ponto de vista, uma utilização duvidosa. A documentação não esclarece totalmente o seu uso, pelo que a descrição que faço se baseia não só na documentação (Javadoc), mas também no próprio código da implementação de referência da Sun (sim, dei-me ao trabalho de olhar para a implementação para tentar descobrir estas coisas...).

dir Este parâmetro é fácil de descrever: é um dos seguintes valores: `Canvas.UP`, `Canvas.DOWN`, `Canvas.LEFT` ou `Canvas.RIGHT`. Basicamente, indica a direcção em que o utilizador está a percorrer o item.

viewportWidth/viewportHeight Indicam o tamanho da área visível que o formulário atribuiu aos seus itens, ou seja, efectivamente indicam o tamanho da área útil do formulário. Os formulários geralmente não utilizam toda a área disponível para o conteúdo, para poderem ter margens, por exemplo. Estes valores podem ser usados pelo item para adaptar a forma como o atravessamente é feito. Por exemplo, se o item consistir em várias linhas de texto e a altura visível for de 150 píxeis ("`viewportHeight`" igual a 150), então o item pode mover o texto na direcção do atravessamento (se for vertical) em 130 píxeis, de forma a manter 20 píxeis de contexto.

visRect_inout Este parâmetro é mais estranho. É um parâmetro de entrada e saída, isto é, serve para o método receber informação e para devolver informação. É um *array* com quatro elementos que definem um rectângulo (x, y, largura e altura,

nesta mesma ordem). Segundo a documentação, quando o método é invocado, este parâmetro define a região do item que está visível. Ou seja, se tivermos um item maior do que o ecrã do telemóvel, este parâmetro indicar-nos-ia em que região o utilizador estaria no momento. No entanto, segundo o código da implementação de referência, este parâmetro é sempre passado com a região que corresponde exactamente à dimensão do item, ou seja:

```
visRect_inout[0] = 0; // X
visRect_inout[1] = 0; // Y
visRect_inout[2] = larguraItem; // Largura
visRect_inout[3] = alturaItem; // Altura
```

O que nos leva à segunda função deste parâmetro: devolver informação ao formulário. O *CustomItem* deve preencher este *array* com a região que o utilizador está a focar. Ou seja, se o item for composto por vários elementos (só assim faz sentido o atravessamento), este parâmetro deverá indicar a região que inclui o elemento que o utilizador está a focar no momento. Esta informação é utilizada pelo formulário para realizar *scrolling*, caso a dimensão do nosso item a isso obrigue, *i.e.*, se for demasiado grande para ser apresentado todo de uma só vez.

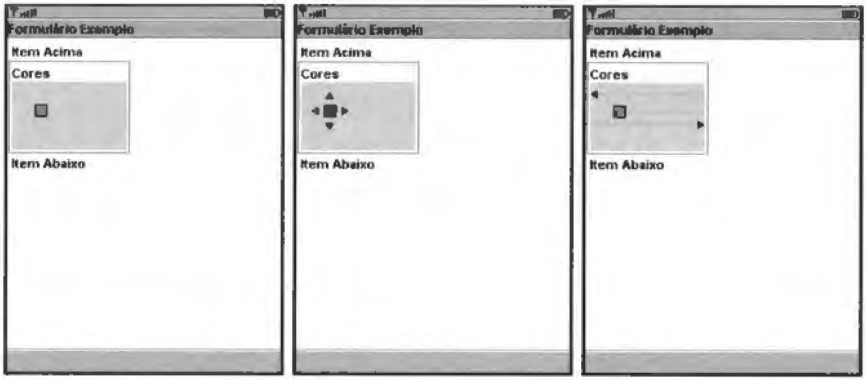
Nem todas as implementações suportam atravessamento interno e, quando suportam, podem não suportar as duas direcções, *i.e.*, podem suportar apenas `TRAVERSE_HORIZONTAL` ou apenas `TRAVERSE_VERTICAL`.

O `CustomItem` é responsável por terminar o atravessamento interno, devolvendo *false* no método `traverse()`, quando chegar ao último elemento (obviamente, o último elemento depende da forma como o atravessamento está a ser feito: para cima, para baixo, para a esquerda ou para a direita). Isto significa que não devem ser implementados comportamentos circulares, *i.e.*, voltar ao primeiro elemento quando se ultrapassa o último. Se isto for feito, o foco ficará permanentemente no item e os restantes itens do formulário não poderão ser acedidos.

Existe outro método relacionado com o atravessamento: o método `traverseOut()`. Este método é invocado pelo sistema quando o atravessamento saiu do item.

O exemplo seguinte ilustra um `CustomItem` que consiste numa grelha que pode ser percorrida pelo utilizador. A posição actual é indicada por um quadrado com borda a preto. O item foi implementado de forma a suportar atravessamento vertical e horizontal ou apenas um deles. Isto significa que se o dispositivo suportar atravessamento horizontal e vertical simultaneamente, o utilizador pode percorrer a grelha na horizontal e na vertical – pode mover o quadrado para a direita, esquerda, para cima e para baixo. Se o dispositivo apenas suportar atravessamento horizontal ou apenas vertical, então o movimento na grelha terá de ser feito em apenas uma dimensão – a grelha é percorrida na horizontal, como se a matriz tivesse sido colocada toda numa linha. A Figura 6.11 mostra o formulário com o nosso item

e as duas formas do atravessamento ser realizado consoante o dispositivo suporte, ou não, atravessamento vertical e horizontal.



(a) CustomItem (b) Atravessamento horizontal e vertical (c) Só atravessamento horizontal ou vertical

Figura 6.11 Um `CustomItem` com atravessamento interno.

O item implementado exemplifica também o uso de eventos de teclado. Quando o utilizador pressiona a tecla correspondente à *game action* `FIRE`, a cor do quadrado de selecção é alterada aleatoriamente.

Exemplo 6.9: `CoresItem` – Um `CustomItem` com atravessamento interno

```
importjavax.microedition.lcdui.*;
import java.util.Random;

public class CoresItem extends CustomItem {

    /**
     * A dimensão dos quadrados de cor.
     */
    private static final int LADO = 10;


    /**
     * Constantes que indicam a localização do atravessamento,
     * ACIMA, significa que o utilizador está no item acima do
     * nosso; DENTRO, significa que o utilizador está no nosso
     * item; ABAIXO, significa que está no item por baixo do nosso.
     */
    private static final int ACIMA = 0;
    private static final int DENTRO = 1;
    private static final int ABAIXO = 2;
```

```
/**
 * A dimensão do item.
 */
private int largura = 100;
private int altura = 60;

/**
 * O número de linhas e colunas de quadrados de cor.
 */
private int númeroLinhas, númeroColunas;

/**
 * A posição actual no atravessamento interno do item.
 */
private int posiçãoX = 0, posiçãoY = 0;

/**
 * Indica se o utilizador está a efectuar o atravessamento interno do
 * item ou se está no item acima ou abaixo.
 */
private int localização = ACIMA;

/**
 * Indicam o tipo de atravessamento suportado.
 */
private boolean atravessaHorizontal, atravessaVertical;

/**
 * A cor do quadrado seleccionado.
 */
private int cor = 255;

/**
 * Gerador de números aleatórios para a cor do item
 * quando se pressiona FIRE.
 */
Random random = new Random();


public CoresItem(String title) {

   super(title);

   /* calcular o número de linhas e colunas */
   númeroLinhas = altura/LADO;
   númeroColunas = largura/LADO;


   /* descobrir quais os tipos de atravessamento que a implementação
      permite
    */
   int modoInteracção = getInteractionModes();
   atravessaHorizontal = ((modoInteracção &
       CustomItem.TRAVERSE_HORIZONTAL)!= 0);
   atravessaVertical = ((modoInteracção &
       CustomItem.TRAVERSE_VERTICAL) != 0);

}

public void paint(Graphics g, int w, int h) {
   g.setColor(255, 0, 0);
   g.fillRect(0, 0, largura, altura);
```

```
        for (int i = 0; i < largura; i+= LADO) {
            for (int j = 0; j < altura; j += LADO) {

                g.setColor(200, 200, 200);

                g.fillRect(i, j, LADO, LADO);
            }
        }

        /* pintar o quadrado seleccionado */
        g.setColor(cor, cor, cor);
        g.fillRect(posiçãoX*LADO, posiçãoY*LADO, LADO, LADO);

        /* desenhar uma borda */
        g.setColor(0, 0, 0);
        g.drawRect(posiçãoX*LADO, posiçãoY*LADO, LADO, LADO);
    }

    public boolean traverse(int dir, int viewportWidth, int
     viewportHeight, int []visRect_inout) {
        boolean atravessando = true;

        /* guardar a posição actual */
        int posiçãoXAnterior = posiçãoX, posiçãoYAnterior = posiçãoY;

        /*
        Temos atravessamento horizontal e vertical.
        As teclas UP e DOWN fazem o quadrado deslocar-se para cima e para
        baixo, respectivamente.
        As teclas LEFT e RIGHT deslocam o quadrado para a esquerda e para
        a direita.
        */
        if (atravessaHorizontal && atravessaVertical) {
            switch(dir) {
                    case Canvas.UP:
                        if (localização == ACIMA) {

                        } else if (localização == ABAIXO) {
                            localização = DENTRO;
                            atravessando = true;
                        } else if (localização == DENTRO) {
                            if (posiçãoY == 0) {
                                localização = ACIMA;
                                atravessando = false;
                            } else {
                                posiçãoY--;
                                atravessando = true;
                            }
                        }
                        break;
                    case Canvas.DOWN:
                        if (localização == ACIMA) {
                            localização = DENTRO;
                            atravessando = true;
                        } else if (localização == ABAIXO) {

                        } else if (localização == DENTRO) {
                            if (posiçãoY == (numeroLinhas - 1)) {
                                localização = ABAIXO;
                                atravessando = false;
                            } else {
                                posiçãoY++;
                                atravessando = true;
```

```
                    }
                }
                break;
            case Canvas.LEFT:
                if (localização == ACIMA) {

                } else if (localização == ABAIXO) {
                    localização = DENTRO;
                    atravessando = true;
                } else if (localização == DENTRO) {
                    if (posiçãoX == 0) {
                        localização = ACIMA;
                        atravessando = false;
                    } else {
                        posiçãoX--;
                        atravessando = true;
                    }
                }
                break;
            case Canvas.RIGHT:
                if (localização == ACIMA) {
                    localização = DENTRO;
                    atravessando = true;
                } else if (localização == ABAIXO) {

                } else if (localização == DENTRO) {
                    if (posiçãoX == (númeroColunas - 1)) {
                        localização = ABAIXO;
                        atravessando = false;
                    } else {
                        posiçãoX++;
                        atravessando = true;
                    }
                }
                break;
        }
/*
Temos só atravessamento horizontal ou vertical (não interessa
qual).
As teclas UP e LEFT fazem o quadrado deslocar-se para a esquerda e
para cima (primeiro para a esquerda e só depois para cima).
As teclas DOWN e RIGHT fazem o inverso.
*/
} else if (atravessaHorizontal || atravessaVertical) {
    switch (dir) {
            case Canvas.UP:
            case Canvas.LEFT:
                if (localização == ACIMA) {

                } else if (localização == ABAIXO) {
                    localização = DENTRO;
                    posiçãoY = númeroLinhas - 1;
                    posiçãoX = númeroColunas - 1;
                    atravessando = true;
                } else if (localização == DENTRO) {
                    posiçãoX--;
                    if (posiçãoX < 0 ) {
                        posiçãoX = númeroColunas-1;
                        posiçãoY--;
                        if (posiçãoY < 0) {
                            localização = ACIMA;
                            posiçãoX = 0;
                            atravessando = false;
```

```
                        }
                    }
                }
                break;
            case Canvas.DOWN:
            case Canvas.RIGHT:
                if (localização == ACIMA) {
                    localização = DENTRO;
                    posiçãoY = 0;
                    posiçãoX = 0;
                    atravessando = true;

                } else if (localização == ABAIXO) {

                } else if (localização == DENTRO) {
                    posiçãoX++;
                    if (posiçãoX >= númeroColunas ) {
                        posiçãoX = 0;
                        posiçãoY++;
                        if (posiçãoY >= númeroLinhas) {
                            localização = ABAIXO;
                            posiçãoX = númeroColunas -1;
                            atravessando = false;
                        }
                    }
                }
                break;
            }
        }
        repaint();

        /* informar qual a área focada */
        visRect_inout[0] = posiçãoX * LADO;
        visRect_inout[1] = posiçãoY * LADO;
        visRect_inout[2] = LADO;
        visRect_inout[3] = LADO;

        /* notificar a aplicação que o valor foi alterado */
        if (posiçãoX != posiçãoXAnterior || posiçãoY != posiçãoYAnterior) {
            notifyStateChanged();
        }

        return atravessando;
    }

    public void traverseOut() {
        System.out.println("Atravessamento interno terminado.");
    }

    public void keyPressed(int keyCode) {
        if (getGameAction(keyCode) == Canvas.FIRE) {
            cor = random.nextInt() & 0x000000ff;
            repaint();
        }
    }

    public int getMinContentWidth() {
        return 100;
    }

    public int getMinContentHeight() {
        return 60;
    }
```

```
    public int getPrefContentWidth(int largura) {
        return getMinContentWidth();
    }

    public int getPrefContentHeight(int altura) {
        return getMinContentHeight();
    }

    public void sizeChanged(int w, int h) {
        this.altura = h;
        this.largura = w;
    }
}
```

Há ainda mais duas coisas a ter em atenção quando se implementa um `CustomItem`. A primeira é a de que se deve invocar o método `notifyStateChanged()` quando o utilizador altera o conteúdo do nosso item, tendo o cuidado de verificar que o valor foi realmente alterado (não apenas editado) e que essa alteração foi consequência da acção do utilizador e não de uma chamada a uma função da API do item. Segundo, pode dar-se o caso de a implementação não suportar nenhum dos tipos de interacção, i.e., não suportar eventos de teclado, nem ponteiro, nem atravessamento interno horizontal, nem vertical. Nesse caso como permitimos ao utilizador alterar o valor do nosso item? A solução é utilizar comandos associados ao item. O item pode criar um comando para invocar um ecrã para introdução de dados e retornar ao formulário quando a introdução tiver terminado.

Um `CustomItem` bem escrito deve verificar todas as capacidades de interacção do dispositivo e implementar alternativas, e não "confiar" que determinada capacidade está presente.

CAPÍTULO 7

Armazenamento Persistente – RMS

Em MIDP não existem ficheiros em que possamos armazenar informação de forma persistente[1]. Os dispositivos, mesmo que o seu sistema operativo possua sistema de ficheiros, não expõem esse sistema de ficheiros às aplicações MIDP[2].

Este capítulo apresenta o sistema utilizado em MIDP para armazenar dados de forma persistente.

7.1. *Record Management System* e *Record Stores*

Para as aplicações MIDP poderem armazenar dados de forma persistente foi criado o chamado *Record Management System* – RMS. O RMS é como um gestor de bases de dados muito (mas mesmo muito!) simplificado.

O RMS consiste num conjunto de *record stores*. Uma *record store* não é mais do que um conjunto de registos com dois campos: um identificador (um número) e um campo de dados (um *array* de bytes). Em terminologia de bases de dados, a *record store* pode ser vista como uma tabela com duas colunas, em que o identificador é a chave primária. As *record stores* são identificadas através do seu nome. O nome tem de ser uma cadeia de caracteres Unicode com tamanho entre 1 e 32 caracteres. Os nomes das *record stores* têm de ser únicos dentro de uma MIDlet Suite, *i.e.*, se tivermos várias MIDlets dentro de uma MIDlet Suite e quisermos que cada MIDlet aceda à sua *record store*, os nomes das *record stores* têm de ser diferentes. Isto porque as *record stores* estão associadas às MIDlet Suites e não às MIDlets. Em MIDP 2.0 existe também a possibilidade de partilhar *record stores* entre MIDlets pertencentes a MIDlet Suites diferentes.

1. Persistente, no sentido em que os dados armazenados sobrevivem mesmo que a MIDlet termine e, regra geral, mesmo que o dispositivo seja reiniciado.
2. Na verdade existe um pacote opcional que fornece uma API para sistema de ficheiros. No entanto, como pacote opcional, não faz parte do MIDP e poucos dispositivos o implementam, pelo que não o irei abordar aqui.

7.2. Criar uma *Record Store*

A classe que representa uma *record store* chama-se, apropriadamente, `RecordStore` e está inserida no pacote `javax.microedition.rms`.

Esta classe não possui construtores públicos. As *record stores* são criadas e abertas através de um dos três seguintes métodos estáticos:

- `RecordStore openRecordStore(String recordStoreName, boolean createIfNecessary)`
  Abre ou cria uma *record store* associada à MIDlet Suite em que a nossa MIDlet está inserida. Se a *record store* não existir e o parâmetro "`createIfNecessary`" for *true*, então será criada, caso contrário uma excepção é lançada.
- `RecordStore openRecordStore(String recordStoreName, boolean createIfNecessary, int authmode, boolean writable)`
  Este método cria (ou abre se já existir) uma *record store* com determinadas permissões para outras MIDlet Suites. O parâmetro "`authmode`" pode ser:
  RecordStore.AUTHMODE_PRIVATE Apenas permite que a *record store* seja acedida por MIDlets na mesma MIDlet Suite. É o mesmo que utilizar o método `openRecordStore(String recordStoreName, boolean createIfNecessary)`.
  RecordStore.AUTHMODE_ANY Permite que a *record store* seja acedida por qualquer MIDlet Suite. É necessário algum cuidado com esta opção porque, neste caso, a *record store* fica completamente desprotegida.
  O parâmetro "`writable`" apenas faz sentido quando se utiliza a opção `AUTHMODE_ANY`, uma vez que este parâmetro define se as MIDlets noutras MIDlet Suites podem, ou não, escrever nesta *record store*. As MIDlets na MIDlet Suite em que está a *record store* podem sempre ler e escrever.
  No caso da *record store* já existir, os parâmetros "`authmode`" e "`writable`" são ignorados.
- `RecordStore openRecordStore(String recordStoreName, String vendorName, String suiteName)`
  Ao contrário dos anteriores, este método apenas permite abrir uma *record store* já existente. Funciona como companheiro do método anterior na medida em que permite abrir *record stores* associadas a outras MIDlet Suites, caso tenha sido dada a permissão adequada. Os parâmetros "`vendorName`" e "`suiteName`" identificam a MIDlet Suite que possui a *record store* que pretendemos abrir. Os valores destes

parâmetros devem condizer com os valores dos atributos correspondentes no descritor da aplicação.

No caso de não termos permissão para abrir a *record store*, uma excepção do tipo `SecurityException` será lançada.

Se indicarmos a nossa própria MIDlet Suite, este método comporta-se como `openRecordStore(recordStoreName, false)`.

De notar que, no caso de a *record store* já ter sido aberta por uma MIDlet na nossa MIDlet Suite (incluindo a própria MIDlet), todos estes métodos devolvem uma referência para o mesmo objecto `RecordStore`.

Se quisermos alterar as permissões de uma *record store* já criada devemos utilizar o método `setMode(int authmode, boolean writable)`. Obviamente, apenas a MIDlet Suite que possui a *record store* pode alterar as permissões.

Quando não precisarmos mais da *record store* podemos (e devemos) fechá-la invocando o método `closeRecordStore()`. Há que ter em atenção, no entanto, que este método apenas liberta realmente os recursos associados com a *record store* quando tiver sido invocado tantas vezes quantas as chamadas a `openRecordStore()`. Isto porque é perfeitamente legal abrir *record stores* já abertas; o sistema simplesmente retorna a referência para a *record store*. Por isso, se abrirmos uma determinada *record store* duas vezes, temos de a fechar também duas vezes.

7.3. Inserir, Obter e Apagar Registos

Depois de aberta, podemos usar a *record store* para adicionar ou manipular os registos lá contidos.

Um registo da *record store* é constituído por apenas dois campos – um identificador e um *array* de bytes de tamanho variável:

ID (*int*)	*array* (*byte*)

A inserção de um registo novo na *record store* é feita usando o método `int addRecord(byte[] data, int offset, int numBytes)`. Este método adiciona um registo com os dados definidos pelo *array* "`data`" (apenas os "`numBytes`" bytes a partir da posição "`offset`" serão adicionados). O método retorna o identificador do novo registo.

Podemos também alterar o conteúdo de um registo já existente, se soubermos o seu ID:

```
void setRecord(int recordId, byte[] newData, int offset,
    int numBytes)
```

Também podemos apagar um registo através do seu identificador:

```
void deleteRecord(int recordId)
```

Uma vez apagado o registo, o identificador associado nunca mais será utilizado por outro registo.

Para obter um registo podemos usar um de dois métodos:

`byte[] getRecord(int recordId)` ou

`int getRecord(int recordId, byte[] buffer, int offset)`

O primeiro devolve um *array* com os dados do registo identificado por "`recordId`". O segundo preenche o *array* "`buffer`", a partir da posição "`offset`" com os dados do registo, retornando o número de bytes copiados para o *array*.

É possível determinar qual o espaço ocupado por um determinado registo, antes de o obter, com o método `int getRecordSize(int recordId)`.

O Exemplo 7.1 demonstra o uso de alguns dos métodos da API das *record stores*. A MIDlet do exemplo permite adicionar entradas a uma base de dados (*record store*) e visualizar essas entradas numa lista. A Figura 7.1 mostra o ecrã das entradas com três registos.

Exemplo 7.1: `RMSCriar` – Abrir uma `RecordStore`

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.*;
import javax.microedition.rms.*;

public final class RMSCriar extends MIDlet implements CommandListener {
    /**
     * A nossa base de dados.
     */
    private RecordStore bd;

    /**
     * A caixa de texto para inserir novas entradas.
     */
    private TextBox caixaTexto;

    /**
     * A lista com todas as entradas na base de dados.
     */
    private List lista;

    /**
     * Comandos para sair da aplicação, introduzir nova entrada, ver
```

```
 * lista de entradas e para confirmar nova entrada.
*/
private Command cmdSair, cmdNova, cmdLista, cmdOk;

public RMSCriar() {

   /* criar a caixa de entrada de texto */
   caixaTexto = new TextBox("Nova entrada", "", 255, TextField.ANY);

   /* criar a lista que mostra todas as entradas */
   lista = new List("Entradas", List.IMPLICIT);


   /* criar os comandos e associá-los aos ecrãs */
   cmdSair = new Command("Sair", Command.EXIT, 1);
   cmdNova = new Command("Nova", "Nova Entrada", Command.SCREEN, 2);
   cmdLista = new Command("Lista", "Lista Entradas", Command.SCREEN, 2);
   cmdOk = new Command("Ok", Command.OK, 0);

   caixaTexto.addCommand(cmdOk);
   caixaTexto.addCommand(cmdLista);
   caixaTexto.setCommandListener(this);

   lista.addCommand(cmdSair);
   lista.addCommand(cmdNova);
   lista.setCommandListener(this);
}
/**
 * Preenche a lista com todas as entradas encontradas na base de
 * dados.
 */
private void preencheLista() {
   byte [] entrada;

   /* limpar a lista porque vamos voltar a colocar todo o conteúdo da
      base de dados
   */
   lista.deleteAll();
   try {
       for (int i = 1; i <= bd.getNumRecords(); i++) {
            entrada = bd.getRecord(i);
            lista.append(new String(entrada), null);
       }
   } catch (RecordStoreNotOpenException rsnoe) {
            System.err.println("A base de dados não foi aberta: " +
                rsnoe.getMessage());
   } catch (InvalidRecordIDException irie) {
            System.err.println("Identificador de registo inválido: " +
                irie.getMessage());
   } catch (RecordStoreException rse) {
            System.err.println("Erro ao aceder à base de dados: " +
                se.getMessage());
   }
}

public void startApp() {
   /* vamos abrir (criar se ainda não tiver sido criada) a base de
      dados
   */
   try {
       bd = RecordStore.openRecordStore("ListaBD", true);
   } catch (RecordStoreException rse) {
```

```
            System.err.println("Não foi possivel abrir a base de dados: " +
                rse.getMessage());
        }

        /* preencher a lista com as entradas */
        preencheLista();

        Display.getDisplay(this).setCurrent(lista);
    }

    public void pauseApp() {
        /* libertar o recurso */
        if (bd != null) {
            try {
                bd.closeRecordStore();
            } catch (RecordStoreNotOpenException rsnoe) {
            } catch (RecordStoreException rse) {
            }
        }
    }

    public void destroyApp(boolean unconditional) {
        /* libertar o recurso */
        if (bd != null) {
            try {
                bd.closeRecordStore();
            } catch (RecordStoreNotOpenException rsnoe) {
            } catch (RecordStoreException rse) {
            }
        }
    }

    public void commandAction(Command c, Displayable d) {
        if (c == cmdSair) {
            notifyDestroyed();

        } else if (c == cmdOk) {
            /* inserir entrada */
            String entrada = caixaTexto.getString();
            if (entrada != null && entrada.length() > 0) {
                try {
                    byte []dados = entrada.getBytes();
                    bd.addRecord(dados, 0, dados.length);
                } catch (RecordStoreException rse) {
                    System.err.println(rse.getMessage());
                }
                preencheLista();
                Display.getDisplay(this).setCurrent(lista);
            }

        } else if (c == cmdLista) {
            preencheLista();
            Display.getDisplay(this).setCurrent(lista);

        } else if (c == cmdNova) {
            Display.getDisplay(this).setCurrent(caixaTexto);
        }
    }
}
```

Figura 7.1 `RMSCriar` – Criar registos numa *record store*.

7.4. Ler e Escrever Tipos Primitivos em *Record Stores*

Os registos utilizados no exemplo anterior eram registo simples compostos por apenas uma cadeia de caracteres, sem estrutura nenhuma. No entanto, se quisermos armazenar registos mais complexos, com vários campos, necessitamos de uma forma de os estruturar. Por exemplo, podemos ter necessidade de armazenar informação relativa à pontuação mais alta de um jogo:

Nome do jogador	Nível	Pontuação
Jorge	4	1289
Carlos	3	1078

A forma mais simples de fazer isto é utilizar tipos de dados primitivos para cada campo, *i.e.*:

Nome do jogador	Nível	Pontuação
`String`	`byte`	`int`

Mas como fazemos para escrever e ler cada um dos campos se a `RecordStore` não fornece nenhuma forma para ler e escrever tipos de dados primitivos? A solução passa por utilizar as *streams* do Java.

O exemplo seguinte mostra como escrever e ler tipos primitivos para uma *record store*. O exemplo é muito semelhante ao anterior, com a diferença de que os registos são agora

compostos pelos campos Nome, Nível e Pontuação. Alguns métodos menos relevantes foram omitidos.

Exemplo 7.2: `RMSTipos` – Tipos de dados primitivos com `RecordStore`

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.*;
import javax.microedition.rms.*;
import java.io.*;

public final class RMSTipos extends MIDlet implements CommandListener {
    /**
     * A nossa base de dados.
     */
    private RecordStore bd;

    /**
     * O formulário para inserir novos registos.
     */
    private Form formEntrada;

    /**
     * Os campos do formulário. Correspondem aos campos
     * do registo na RecordStore.
     */
    private TextField campoNome, campoNivel, campoPontuação;

    /**
     * A lista com todas as entradas na base de dados.
     */
    private List lista;

    /**
     * Comandos para sair da aplicação, introduzir nova entrada, ver
     * lista de entradas e para confirmar nova entrada.
     */
    private Command cmdSair, cmdNova, cmdLista, cmdOk;

    public RMSTipos() {

        /* criar a caixa de entrada de texto */
        formEntrada = new Form("Nova entrada");

        campoNome = new TextField("Nome", "", 255, TextField.ANY);
        campoNivel = new TextField("Nivel", "", 2, TextField.NUMERIC);
        campoPontuação = new TextField("Pontuação", "", 5,
        TextField.NUMERIC);

        formEntrada.append(campoNome);
        formEntrada.append(campoNivel);
        formEntrada.append(campoPontuação);


        /* criar a lista que mostra todas as entradas */
        lista = new List("Entradas", List.IMPLICIT);


        /* criar os comandos e associá-los aos ecrãs */
        cmdSair = new Command("Sair", Command.EXIT, 1);
        cmdNova = new Command("Nova", "Nova Entrada", Command.SCREEN, 2);
```

```
        cmdLista = new Command("Lista", "Lista Entradas", Command.SCREEN,
         2);
        cmdOk = new Command("Ok", Command.OK, 0);

        formEntrada.addCommand(cmdOk);
        formEntrada.addCommand(cmdLista);
        formEntrada.setCommandListener(this);

        lista.addCommand(cmdSair);
        lista.addCommand(cmdNova);
        lista.setCommandListener(this);
    }

    /**
     * Escreve um registo na RecordStore.
     */
    private void escreveRegisto(String nome, byte nível, int pontuação) {
        try {
            ByteArrayOutputStream streamBytes = new
            ByteArrayOutputStream();

            DataOutputStream streamDados = new DataOutputStream(streamBytes);

            byte []registo;

            /* escrever os tipos primitivos */
            streamDados.writeUTF(nome);
            streamDados.writeByte(nível);
            streamDados.writeInt(pontuação);

            streamDados.flush();

            /* obter o array para escrever na RecordStore */
            registo = streamBytes.toByteArray();

            System.out.println(new String(registo));

            bd.addRecord(registo, 0, registo.length);

            streamDados.close();
            streamBytes.close();
        } catch (IOException ioe) {
        } catch (RecordStoreException rse) {
        }
    }


    /**
      * Preenche a lista com todas as entradas encontradas na base de
      * dados.
      */
    private void preencheLista() {
        byte [] entrada = new byte[100];

        String nome;
        byte nível;
        int pontuação;

        ByteArrayInputStream streamBytes = new ByteArrayInputStream(entrada);
        DataInputStream streamDados = new DataInputStream(streamBytes);

        /* limpar a lista porque vamos voltar a colocar todo o conteúdo da
           base de dados.
```

```
            */
            lista.deleteAll();

            try {
                for (int i = 1; i <= bd.getNumRecords(); i++) {

                    /* ler o registo inteiro */
                    bd.getRecord(i, entrada, 0);

                    /* separar os campos */
                    nome = streamDados.readUTF();
                    nivel = streamDados.readByte();
                    pontuação = streamDados.readInt();

                    /* colocar entrada na lista com todos os campos */
                    lista.append(nome + " " + String.valueOf(nivel) + " " +
                    String.valueOf(pontuação), null);

                    /* reiniciar a stream */
                    streamBytes.reset();
                }
                streamDados.close();
                streamBytes.close();
            } catch (IOException ioe) {
                System.err.println("Erro de IO: " + ioe.getMessage());
            } catch (RecordStoreNotOpenException rsnoe) {
                System.err.println("A base de dados não foi aberta: " +
                    rsnoe.getMessage());
            } catch (InvalidRecordIDException irie) {
                System.err.println("Identificador de registo inválido: " +
                    irie.getMessage());
            } catch (RecordStoreException rse) {
                System.err.println("Erro ao aceder à base de dados: " +
                    rse.getMessage());
            }
        }
[...]

    public void commandAction(Command c, Displayable d) {
        if (c == cmdSair) {
            notifyDestroyed();

        } else if (c == cmdOk) {
            /* inserir entrada */
            String nome = campoNome.getString();
            byte nivel = Byte.parseByte(campoNivel.getString());
            int pontuação = Integer.parseInt(campoPontuação.getString());

            escreveRegisto(nome, nivel, pontuação);
            preencheLista();
            Display.getDisplay(this).setCurrent(lista);

        } else if (c == cmdLista) {
            preencheLista();
            Display.getDisplay(this).setCurrent(lista);

        } else if (c == cmdNova) {
            Display.getDisplay(this).setCurrent(formEntrada);
        }
    }
}
```

Há um ponto digno de nota neste último exemplo: o facto de ter usado o método `bd.getRecord(i, entrada, 0);` para ler o registo da *record store* e não `entrada = bd.getRecord(i);` isto porque não queremos obter uma referência para o *array* do registo, mas, antes, queremos que os dados sejam copiados para o nosso *array*. Caso contrário, a `ByteArrayInputStream` não estaria a trabalhar com o *array* dos dados do registo mas com um *array* vazio.

7.5. Enumerar Registos

Até agora tenho vindo a utilizar um simples ciclo *for* para percorrer a lista de registos na *record store*. Isto funciona nos exemplos dados porque nunca apagámos registos. Se apagássemos um registo, o método deixaria de funcionar porque iríamos tentar aceder a um registo inexistente. Vamos supor que temos uma *record store* com três registos com IDs 1, 2 e 3 e apagámos o registo 2. Ou seja, ficamos com dois registos com IDs 1 e 3. O ciclo que tenho vindo a utilizar para percorrer os registos iria gerar os IDs 1 e 2:

```
for (int i = 1; i <= bd.getNumRecords(); i++) {
    bd.getRecord(i, entrada, 0);
}
```

No entanto, o registo com ID 2 deixou de existir, pelo que nos vamos deparar com um problema na segunda iteração do ciclo.

Uma forma mais prática de percorrer a lista de registos é utilizar o `RecordEnumeration`. Na sua forma mais básica, esta classe fornece um mecanismo semelhante à interface *`Enumeration`* do Java SE.

Para obtermos uma `RecordEnumeration` invocamos o método

```
enumerateRecords(RecordFilter filter,
    RecordComparator comparator, boolean keepUpdated)
```

da classe `RecordStore`.

Não me vou debruçar já sobre os parâmetros do método. Na forma mais simples podemos simplesmente invocar `enumerateRecords(null, null, false)`.

Assim, o método para percorrer os registos da *record store* passa a ser algo como:

```
db = RecordStore.openRecordStore(...);
...
RecordEnumeration re = db.enumerateRecords(null, null, false);
while (re.hasNextElement()) {
    bytes[] registo = re.nextRecord();
}
```

O método `nextRecord()` retorna o *array* de bytes do próximo registo da `RecordEnumeration`. Se quisermos saber apenas o ID do próximo registo podemos

utilizar o método `int nextRecordId()`. Infelizmente, ambos os métodos fazem com que o registo corrente avance para o próximo registo, *i.e.*, não podemos utilizar algo do género:

```
db = RecordStore.openRecordStore(...);
...
RecordEnumeration re = db.enumerateRecords(null, null, false);
while (re.hasNextElement()) {
    /* não funciona como esperado! */
    int id = re.nextRecordId();
    /* este registo não corresponde ao id */
    bytes[] registo = re.nextRecord();
}
```

porque os dados obtidos correspondem ao registo a seguir ao ID.

Existem também métodos para "andar para trás" nos registos:

```
byte[] previousRecord()
int previousRecordId()
```

Se quisermos saber quantos registos existem na enumeração devemos invocar o método `int numRecords()`. Quando a `RecordEnumeration` é criada da forma que temos vindo a utilizar até agora, o valor devolvido por `numRecords()` é o mesmo que o valor devolvido por `getNumRecords()` da `RecordStore`. No entanto, estes valores podem ser diferentes se usarmos filtros, como vamos ver a seguir.

7.5.1. Filtros

Se apenas nos interessar percorrer determinados registos, podemos usar a `RecordEnumeration` para filtrar esses registos.

Para tal, temos de criar a `RecordEnumeration` com um filtro:

```
enumerateRecords(filtro, null, false)
```

em que "`filtro`" é um objecto que implementa a interface `RecordFilter`.

A interface `RecordFilter` define apenas um método:

```
boolean matches(byte[] candidate)
```

que temos de implementar para definir o filtro que pretendemos e que, basicamente, deve indicar se o registo passado no parâmetro deve, ou não, ser filtrado.

Usando o exemplo da base de dados de pontuações do jogo, vou implementar um filtro para devolver apenas os registos correspondentes às pontuações dos jogadores que chegaram a um determinado nível:

Exemplo 7.3: `FiltroNivel` – Um filtro para a `RecordEnumeration`

```
import javax.microedition.rms.*;
import java.io.*;

public class FiltroNivel implements RecordFilter {
```

```
    private byte nivelComparação = 3;

    public FiltroNivel(int nivel) {
        this.nivelComparação = (byte)nivel;
    }

    public boolean matches(byte[] candidate) {
        String nome;
        byte nivel = 0;
        int pontuação;

        try {
            ByteArrayInputStream streamBytes = new
            ByteArrayInputStream(candidate);
            DataInputStream streamDados = new DataInputStream(streamBytes);

            /* separar os campos */
            nome = streamDados.readUTF();
            nivel = streamDados.readByte();
            pontuação = streamDados.readInt();

            streamDados.close();
            streamBytes.close();
        } catch (IOException ioe) {
            System.err.println("Erro ao filtrar: " + ioe.getMessage());
        }

        /* filtragem propriamente dita */
        if (nivel >= nivelComparação) {
            return true;
        } else {
            return false;
        }
    }
}
```

Se quiséssemos obter os registos dos jogadores que chegaram ao nível 3, faríamos o seguinte:

```
FiltroNivel fn = new FiltroNivel(3);
RecordEnumeration re = bd.enumerateRecords(fn, null, false);

while (re.hasNextElement()) {
    byte []registo = re.nextRecord();
    ...
}
```

7.5.2. Comparadores

Da forma como temos vindo a utilizar a `RecordEnumeration`, não podemos saber em que ordem os registos vão ser devolvidos.

Se pretendermos obter os registo segundo uma determinada ordem temos de passar um `RecordComparator` à `RecordEnumeration`.

À semelhança da `RecordFilter`, `RecordComparator` é uma interface que define apenas um método:

```
int compare(byte[] rec1, byte[] rec2)
```

Este método é chamado pela `RecordEnumeration` para ordenar os registos da *record store*. O valor de retorno deste método é um dos seguintes valores:

RecordComparator.PRECEDES Se o primeiro registo ("`rec1`") precede o segundo ("`rec2`").

RecordComparator.FOLLOWS Se o primeiro registo sucede ao segundo registo.

RecordComparator.EQUIVALENT Se os dois registos são considerados equivalentes para efeito da ordenação.

O exemplo seguinte ilustra a implementação de um comparador que ordena os registos por ordem crescente de pontuação.

Exemplo 7.4: `ComparadorPontuacao` – Um comparador para a enumeração

```
import javax.microedition.rms.*;
import java.io.*;

public class ComparadorPontuacao implements RecordComparator {

    public int compare(byte[] rec1, byte[] rec2) {
        String nome;
        byte nivel;
        int pontuação1 = 0, pontuação2 = 0;

        try {
            /* campos do primeiro registo */
            ByteArrayInputStream streamBytes = new ByteArrayInputStream(rec1);
            DataInputStream streamDados = new DataInputStream(streamBytes);

            nome = streamDados.readUTF();
            nivel = streamDados.readByte();
            pontuação1 = streamDados.readInt();

            streamDados.close();
            streamBytes.close();

            /* campos do segundo registo */
            streamBytes = new ByteArrayInputStream(rec2);
            streamDados = new DataInputStream(streamBytes);

            nome = streamDados.readUTF();
            nivel = streamDados.readByte();
            pontuação2 = streamDados.readInt();

            streamDados.close();
            streamBytes.close();
        } catch (IOException ioe) {
            System.err.println("Erro ao filtrar: " + ioe.getMessage());
        }

        /* fazer a comparação. retorna campos na ordem crescente de
           pontuação.
        */
```

```
            if (pontuação2 > pontuação1) {
                return PRECEDES;
            } else if (pontuação2 < pontuação1) {
                return FOLLOWS;
            } else {
                return EQUIVALENT;
            }
        }
    }
```

Este comparador seria utilizado da seguinte forma para percorrermos os registos por ordem crescente de pontuação:

```
ComparadorPontuacao cp = new ComparadorPontuacao();
RecordEnumeration re = bd.enumerateRecords(null, cp, false);

while (re.hasNextElement()) {
    byte []registo = re.nextRecord();
    ...
}
```

Os filtros e os comparadores podem ser usados em simultâneo. Seguindo o tema dos exemplos anteriores poderíamos criar uma `RecordEnumeration` para percorrer, por ordem crescente de pontuação, os registos dos jogadores que chegaram até ao nível 2:

```
FiltroNivel fn = new FiltroNivel(2);
ComparadorPontuacao cp = new ComparadorPontuacao();

RecordEnumeration re = bd.enumerateRecords(fn, cp, false);

while (re.hasNextElement()) {
    byte []registo = re.nextRecord();
    ...
}
```

7.5.3. Manter a enumeração actualizada

Depois de termos criado uma `RecordEnumeration` nada nos impede de fazer alterações à *record store* associada, *i.e.*, adicionar, apagar ou alterar registos. Se nada fizermos, é possível que a `RecordEnumeration` esteja desactualizada relativamente à *record store*.

Se quisermos garantir que a enumeração está actualizada temos duas opções. A primeira é invocar o método `rebuild()`. Isto irá fazer com que a `RecordEnumeration` seja completamente refeita, como se tivesse sido criada novamente, com a excepção de que o índice do registo corrente não é alterado.

A segunda opção é indicar que queremos que a `RecordEnumeration` se mantenha actualizada automaticamente. Podemos fazer isto no momento da criação, passando o valor *true* no parâmetro "`keepUpdated`":

```
enumerateRecords(filtro, comparador, true)
```

ou podemos invocar o método `keepUpdated(boolean keepUpdated)` da `RecordEnumeration`.

É preciso, no entanto, algum cuidado com esta opção. Manter a `RecordEnumeration` actualizada pode fazer com que o desempenho da aplicação se degrade, caso sejam feitas muitas alterações à *record store*.

7.6. Eventos de *Record Store*

Quando se desenvolve uma aplicação com várias *threads* é útil haver mecanismos para "informar" uma *thread* de eventos despoletados por outra.

A classe `RecordStore` permite registar um `RecordListener` de forma que a aplicação receba um evento sempre que a *record store* seja alterada. O registo de um `RecordListener` é feito com o método `addRecordListener(RecordListener listener)`.

A interface `RecordListener` define os seguintes métodos:

- `void recordAdded(RecordStore recordStore, int recordId)`
  Invocado quando um registo é adicionado à *record store*.
- `void recordChanged(RecordStore recordStore, int recordId)`
  Invocado quando um registo é alterado.
- `void recordDeleted(RecordStore recordStore, int recordId)`
  Invocado quando um registo é apagado da *record store*.

Para ilustrar o uso de eventos vamos analisar parte do código da implementação de referência da classe `RecordEnumeration`. O exemplo serve também para mostrar o que acontece por detrás das cortinas quando escolhemos manter a enumeração actualizada.

Podemos ver que, para manter a enumeração actualizada, são usados os eventos da *record store*, actualizando a lista de registos da enumeração sempre que um registo é apagado, inserido ou alterado na *record store*.

```
public void keepUpdated(boolean keepUpdated) {
    checkDestroyed();
    if (keepUpdated != beObserver) {
       beObserver = keepUpdated;
       if (keepUpdated) {
           recordStore.addRecordListener(this);
           rebuild();
       } else {
           recordStore.removeRecordListener(this);
       }
    }
}
```

```
public synchronized void recordAdded(RecordStore recordStore,
     int recordId) {
     checkDestroyed();
     synchronized (recordStore.rsLock) {
        filterAdd(recordId);
     }
}

public synchronized void recordChanged(RecordStore recordStore,
     int recordId) {
     checkDestroyed();

     int recIndex = findIndexOfRecord(recordId);
     if (recIndex < 0) {
        return; // not in the enumeration
     }
     removeRecordAtIndex(recIndex);
     synchronized (recordStore.rsLock) {
        filterAdd(recordId);
     }
}

public synchronized void recordDeleted(RecordStore recordStore,
     int recordId) {
     checkDestroyed();
     /*
      * Remove the deleted element from the records array.
      * No resorting is required.
      */
     int recIndex = findIndexOfRecord(recordId);

     if (recIndex < 0) {
        return; // not in the enumeration
     }
     // remove this record from the enumeration
     removeRecordAtIndex(recIndex);
}
```

CAPÍTULO 8
Threads

O uso de *threads* em MIDP não difere do uso a que estamos habituados com o Java SE, no entanto, a sua utilização em MIDP reveste-se de especial importância pelo que julgo adequado dedicar inteiramente um capítulo a este assunto.

Em qualquer programa Java, as *threads* são um ponto importante. A usabilidade das interfaces depende, muitas vezes, da forma como as *threads* são utilizadas para executar tarefas em segundo plano (*background*) de forma a permitir que a interface com o utilizador continue a responder. Em MIDP isto é ainda mais importante. Nos computadores de secretária as operações são executadas muito mais depressa e os utilizadores são mais tolerantes a interfaces menos "polidas". Num telemóvel isso não acontece. Qualquer operação de rede demora muito tempo e a interface não pode deixar de responder sob pena de o utilizador pensar que a aplicação bloqueou.

8.1. A API

A API CLDC relativa às *threads* é uma versão simplificada da API Java SE. Os métodos da classe `Thread` são apenas:

```
static int activeCount()
static Thread currentThread()
String getName()
int getPriority()
void interrupt()
boolean isAlive()
void join()
void run()
void setPriority(int newPriority)
static void sleep(long millis)
void start()
String toString()
static void yield()
```

Para além de ter menos métodos, a versão CLDC também não possui suporte para *daemon threads* nem para grupos de *threads*.

8.2. Iniciar e Parar

Tal como descrito na documentação Javadoc, existem duas formas de criar uma *thread*. Uma das formas é construir uma subclasse de `Thread` e implementar o método `run`:

```
class MinhaThread extends Thread {
    String parâmetro;

    MinhaThread(String parâmetro) {
       this.parâmetro = parâmetro;
    }

    public void run() {
       /* processar parâmetro*/
    }
}
```

O código para criar e iniciar esta *thread* seria então:

```
MinhaThread minhaThread = new MinhaThread("jorgecardoso.org");
minhaThread.start();
```

A outra forma é declarar uma classe que estende a interface *Runnable*:

```
class MinhaThreadRun implements Runnable {
    String parâmetro;

    MinhaThreadRun(String parâmetro) {
       this.parâmetro = parâmetro;
    }

    public void run() {
    /* processar parâmetro*/
    }
}
```

Neste caso, a inicialização da *thread* seria:

```
MinhaThreadRun minhaThreadrun = new MinhaThreadRun("jorgecardoso.org");
Thread thread = new Thread(minhaThreadRun);
thread.start();
```

Não existe grande diferença entre as duas alternativas embora a versão que utiliza a interface *Runnable* permita que se reutilize uma classe que também sirva outros propósitos que não o de *thread*.

Os exemplos anteriores mostram também como se passam parâmetros às *threads*. O método `run()` não possui parâmetros, pelo que a única forma de os passar é através dos atributos da classe.

Normalmente as *threads* são utilizadas para executar processos morosos, ou mesmo para executar indefinidamente (pelo menos até ao programa terminar).

De qualquer forma a *thread* termina apenas quando terminar de executar o método `run()`[1]. Não existe forma de uma *thread* forçar outra *thread* a parar. A paragem tem de ser aceite pela *thread* que vai terminar. Assim, quando a *thread* é utilizada para executar até uma determinada condição externa (ou interna) ser satisfeita, é necessário implementar o método `run()` utilizando algo do género:

```
class MinhaThreadRun implements Runnable {
    boolean condiçãoParagem = false;

    public void run() {
       while(!condiçãoParagem) {
       /* fazer qualquer coisa */
       }
    }
}
```

Desta forma o programa pode sinalizar a *thread* para terminar em qualquer momento. Basta tornar a condição de paragem verdadeira.

Normalmente, quando paramos uma *thread* queremos que ela termine o mais rapidamente possível. Se o processamento consistir em várias operações demoradas podemos verificar a condição de paragem antes de executar cada uma dessas operações:

```
public void run() {
    while(!condiçãoParagem) {
       operaçãoDemorada1();
       if (condiçãoParagem) {
           break;
       }
       operaçãoDemorada2();
       if (condiçãoParagem) {
           break;
       }
       operaçãoDemorada3();
    }
}
```

Desta forma, se a aplicação quiser terminar a *thread* a meio do processamento, não tem de esperar que todo o ciclo termine.

Nalgumas aplicações faz sentido lançar várias instâncias da mesma *thread* mas de forma que a última a ser lançada substitua as outras. Isto pode ser feito mantendo uma referência para a *thread* mais recente e comparando a *thread* corrente com esta referência, no corpo do método `run()`:

```
class MinhaThreadRun implements Runnable {
    private Thread recente = null;
```

1. A classe `Thread` na versão Java SE possui o método `stop()` que permite parar uma *thread*. No entanto, mesmo em Java SE, este método foi tornado obsoleto (*deprecated*) por ser inconsistente e por não poder ser implementado de forma correcta em todas as plataformas. Este método foi removido da versão Java ME.

```
    public void recomeçar() {
        recente = new Thread(this);
        recente.start();
    }

    public void run() {
        Thread esta = Thread.currentThread();
        while (recente == esta) {
            /* processar */
        }
    }
}
```

Este exemplo utiliza o método estático `currentThread()`. Este método retorna uma referência para o objecto `Thread` que está a executar no momento.

8.3. Sincronização

Na grande maioria das vezes, quando um programa utiliza *threads* há necessidade de partilhar dados entre essas *threads*. Nestes casos é necessário ter cuidado com a forma como o acesso aos dados por parte de cada *thread* é feito – é necessário sincronizar as *threads*.

A sincronização de *threads*, em Java, depende de um mecanismo chamado *monitor*. Um monitor permite definir regiões de código que apenas podem ser executadas por uma *thread* de cada vez. Quando uma *thread* precisa de aceder à região de código protegida, pede ao monitor para reservar para uso exclusivo essa região de forma que apenas essa *thread* possa aceder. Quando a *thread* acaba de executar esse código, a região é libertada. Se uma *thread* tentar aceder a uma região de código reservada, ficará em lista de espera até que a região seja libertada. Nessa altura, o monitor irá dar permissão a uma das *threads* em lista de espera.

Em Java qualquer objecto pode funcionar como monitor. A definição de uma região de código protegida é feita através da palavra-chave `synchronized`:

```
public void métodoComRegiãoProtegida(StringBuffer dados) {
    synchronized(dados) {
        dados.append(Thread.currentThread().getName());
        dados.append(": ");
        dados.append(System.currentTimeMillis());
    }
}
```

Se duas *threads* invocarem este método ao mesmo tempo, uma delas irá ter de esperar que a outra termine de executar a região de código sincronizada.

Uma vez que qualquer objecto Java pode ser usado como monitor, podemos utilizar também a referência `this`:

```
public class Contador {
    private int contador = 0;
```

```
    public int incrementa() {
        synchronized(this) {
            contador++;
            return contador;
        }
    }

    public int decrementa() {
        synchronized(this) {
            contador--;
            return contador;
        }
    }
}
```

Reparem que, neste exemplo, apenas um dos métodos é executado de cada vez. Isto porque o monitor é a própria referência à instância da classe `Contador` e é o mesmo nos dois métodos.

A sincronização através da referência `this` tem um atalho em Java:

```
public class Contador {
    private int contador = 0;

    public synchronized int incrementa() {
        contador++;
        return contador;
    }

    public synchronized int decrementa() {
        contador--;
        return contador;
    }
}
```

Quando um método é declarado `synchronized`, é como se envolvêssemos todo o código do método num bloco sincronizado através da referência `this`. De facto, isto só é verdade nos métodos não estáticos. Se declararmos um método estático `synchronized`, o monitor utilizado é a referência ao objecto `Class` da classe em que o método foi declarado. Isto é:

```
public class MinhaClasse {
    public static synchronized void método() {
        /* código */
    }
}
```

é igual a:

```
public class MinhaClasse {
    public static void método() {
        synchronized(MinhaClasse.getClass()) {
            /* código */
        }
    }
}
```

A sincronização de *threads* pode ser um assunto complicado. É preciso ter algum cuidado com os monitores utilizados. Para dar um exemplo simples, o código seguinte:

```
public void dadosProtegidos(StringBuffer dados) {
    synchronized(dados) {
        dados.append(Thread.currentThread().getName());
        dados.append(": ");
        dados.append(System.currentTimeMillis());
    }
}
```

é muito diferente de:

```
public void synchronized dadosProtegidos(StringBuffer dados) {
    dados.append(Thread.currentThread().getName());
    dados.append(": ");
    dados.append(System.currentTimeMillis());
}
```

Nesta última versão apenas garantimos que o método é acedido exclusivamente. Nada nos garante, no entanto, que os dados não são modificados noutro método qualquer. Esta última versão é equivalente a:

```
public void dadosProtegidos(StringBuffer dados) {
    synchronized(this) {
        dados.append(Thread.currentThread().getName());
        dados.append(": ");
        dados.append(System.currentTimeMillis());
    }
}
```

É facilmente visível que o monitor utilizado é diferente do da primeira versão.

É necessário também ter alguma atenção a possíveis *deadlocks*. Os *deadlocks* acontecem quando uma *thread* A detém um recurso X e necessita de um recurso Y e, ao mesmo tempo uma *thread* B detém o recurso Y e necessita de X. Nesta situação, ambas as *threads* irão esperar indefinidamente pelo recurso. Uma forma de evitar este problema é utilizar sempre a mesma ordem quando se bloqueia um recurso, *i.e.*, bloquear sempre primeiro X e só depois o Y.

8.4. Aguardar e Notificar

Muitas vezes utilizamos *threads* para processar dados continuamente, à medida que estes são disponibilizados.

Uma forma simplista de fazer isto seria utilizar um ciclo que verifica continuamente se os dados estão disponíveis:

```
import java.util.Vector;

public class Processador implements Runnable {
```

```
    private boolean condiçãoParagem = false;
    private Vector fila = new Vector();

    public Processador() {
       new Thread(this).start();
    }

    public void maisDados(Object o) {
       fila.addElement(o);
    }
    public void run() {
       Object dados;

       while(!condiçãoParagem) {
          synchronized(fila) {
             if (fila.size() > 0) {
                dados = fila.elementAt(0);
                fila.removeElementAt(0);

                /* processar dados */
             }
          }
       }
    }
}
```

Neste tipo de ciclos existe aquilo que se designa por *espera activa*, uma vez que a *thread* está sempre a executar, mesmo que não existam dados para processar. Provavelmente, na maior parte do tempo, esta *thread* não executa código útil, i.e., processamento dos dados. No entanto, está a consumir recursos de processamento que podiam estar a ser utilizados por outras *threads*.

A solução para este problema passa por suspender a *thread* quando não existirem dados para processar e activá-la apenas quando os dados ficarem disponíveis. Uma vez que as *threads* suspensas não são escalonadas para processamento, não consomem processador.

Em Java, este mecanismo de suspenção e activação de *threads* é feito através dos métodos `wait()` e `notify()` da classe `Object` (lembrem-se que qualquer objecto pode ser utilizado como monitor). Assim, o exemplo anterior poderia ser reescrito da seguinte forma:

```
import java.util.Vector;

public class Processador implements Runnable {
    private boolean condiçãoParagem = false;
    private Vector fila = new Vector();

    public Processador() {
       new Thread(this).start();
    }

    public void maisDados(Object o) {
       synchronized(fila) {
          fila.addElement(o);
          /* notificar a thread de que estão disponíveis dados */
          fila.notify();
       }
```

```
    }

    public void run() {
        Object dados;

        while(!condiçãoParagem) {
            synchronized(fila) {
                if (fila.size() > 0) {
                    dados = fila.elementAt(0);
                    fila.removeElementAt(0);

                    /* processar dados */
                } else {
                    try {
                        /* suspender a thread até notify() ser invocado */
                        fila.wait();
                    }catch (InterruptedException ie) {
                    }
                }
            }
        }
    }
}
```

Reparem que, neste novo exemplo, a adição de um novo objecto à fila pelo método `maisDados()` teve de ser sincronizado através do objecto `fila`. Isto porque os métodos `wait()` e `notify()` apenas podem ser invocados por *threads* que possuam o monitor do objecto, *i.e.*, apenas pode ser invocado dentro de um bloco sincronizado através do próprio objecto, neste caso, `fila`.

8.5. *Threads* de Sistema

Uma *thread* de sistema é uma *thread* não iniciada nem gerida pela aplicação. As *threads* de sistema são geridas pelo AMS e são responsáveis pelo envio dos eventos e pela actualização do ecrã do telemóvel. Cada aplicação possui pelo menos uma *thread* de sistema e zero ou mais *threads* de aplicação.

São as *threads* de sistema que mantêm a MIDlet a "correr", notificando-a dos eventos. A MIDlet é notificada através dos chamados métodos de rechamada (*callback*)[2], *i.e.*, `paint()`, `keyPressed()`, `commandAction()`, etc. Existem quatro tipos de rechamada relacionados com a UI:

- Comandos abstractos que fazem parte da API de alto nível.
- Eventos de baixo nível que representam o pressionar e largar de teclas (e ponteiro).

2. Métodos de rechamada (*callback*) são métodos definidos num objecto mas invocados por outro quando um evento ocorre.

- Chamadas ao método `paint()` da classe `Canvas`.
- Chamadas ao método `run` de um objecto `Runnable`, devido a um pedido feito através de uma chamada a `callSerially()` da classe `Display`.

As chamadas a estes métodos são feitas de uma forma serializada por parte da *thread* de sistema. Isto significa que estes métodos nunca são invocados em paralelo e, por isso, uma chamada só é feita quando a chamada anterior tiver terminado[3].

Os métodos de rechamada são serializados em relação uns aos outros, i.e., são invocados sempre por uma determinada ordem (não significa que sejam todos invocados sempre):

- `Canvas.hideNotify`
- `Canvas.keyPressed`
- `Canvas.keyRepeated`
- `Canvas.keyReleased`
- `Canvas.paint`
- `Canvas.pointerDragged`
- `Canvas.pointerPressed`
- `Canvas.pointerReleased`
- `Canvas.showNotify`
- `Canvas.sizeChanged`
- `CommandListener.commandAction`
- `CustomItem.getMinContentHeight`
- `CustomItem.getMinContentWidth`
- `CustomItem.getPrefContentHeight`
- `CustomItem.getPrefContentWidth`
- `CustomItem.hideNotify`
- `CustomItem.keyPressed`
- `CustomItem.keyRepeated`
- `CustomItem.keyReleased`
- `CustomItem.paint`
- `CustomItem.pointerDragged`
- `CustomItem.pointerPressed`
- `CustomItem.pointerReleased`
- `CustomItem.showNotify`

3. De facto existe uma excepção a esta regra: quando o método `Canvas.serviceRepaints()` é invocado. Este método faz com que o método `paint()` seja invocado e espera que termine. Isto acontece mesmo que `serviceRepaints()` tenha sido chamado dentro de um *callback* activo.

- `CustomItem.sizeChanged`
- `CustomItem.traverse`
- `CustomItem.traverseOut`
- `Displayable.sizeChanged`
- `ItemCommandListener.commandAction`
- `ItenstateListener.itenstateChanged`
- `Runnable.run` resultante de uma chamada a `Display.callSerially`

Uma vez que estes métodos são invocados por uma *thread* que não pertence à aplicação, é preferível executá-los o mais rapidamente possível. A *thread* de sistema não pode processar mais nenhum evento enquanto o método invocado não terminar. Isto significa que, se o método executar uma operação muito lenta, ou bloquear, a UI deixa de responder.

É boa prática utilizar uma *thread* diferente para executar operações demoradas em resposta a uma entrada de dados do utilizador. Na verdade, nalguns casos é mesmo quase obrigatório – na criação de ligações à rede.

Quando uma MIDlet *untrusted* tenta executar uma operação protegida, como abrir uma ligação à rede, a implementação MIDP interrompe a execução do programa para perguntar ao utilizador se permite que a operação seja realizada. Isto significa que é necessário exibir um ecrã (provavelmente um `Alert`) e obter a resposta do utilizador. No entanto, se a ligação à rede for feita dentro de um método de rechamada, por exemplo dentro de um `commandAction` em resposta a uma escolha do utilizador, o sistema não pode processar mais nenhum evento enquando o `commandAction` não terminar. Isto é uma situação de *deadlock*: o método que abre a ligação à rede fica à espera de confirmação por parte do utilizador mas a confirmação por parte do utilizador só pode ser feita depois do método que abre a ligação à rede terminar. Por isso, as ligações à rede, como vamos ver no próximo capítulo, devem ser sempre feitas em *threads* diferentes.

CAPÍTULO 9
Comunicações

Uma das grandes vantagens de podermos programar dispositivos móveis é o facto de termos acesso a serviços *online*, onde quer que estejamos. Podemos, por exemplo, aceder a uma página Web, ver o correio electrónico, enviar uma mensagem SMS, etc. De forma a podermos realizar estas operações necessitamos, obviamente, de efectuar ligações à rede e comunicar com servidores. Este capítulo trata da API que nos permite fazer isso.

9.1. *Generic Connection Framework*

Em CLDC/MIDP, as conexões à rede são feitas de forma ligeiramente diferente daquilo a que estamos habituados em Java SE.

A configuração CLDC define o que se designou por *Generic Connection Framework* – GCF.

A Figura 9.1 mostra o diagrama de classes que fazem parte do GCF. As classes marcadas com asterisco fazem parte do perfil MIDP, as outras são definidas pela CLDC.

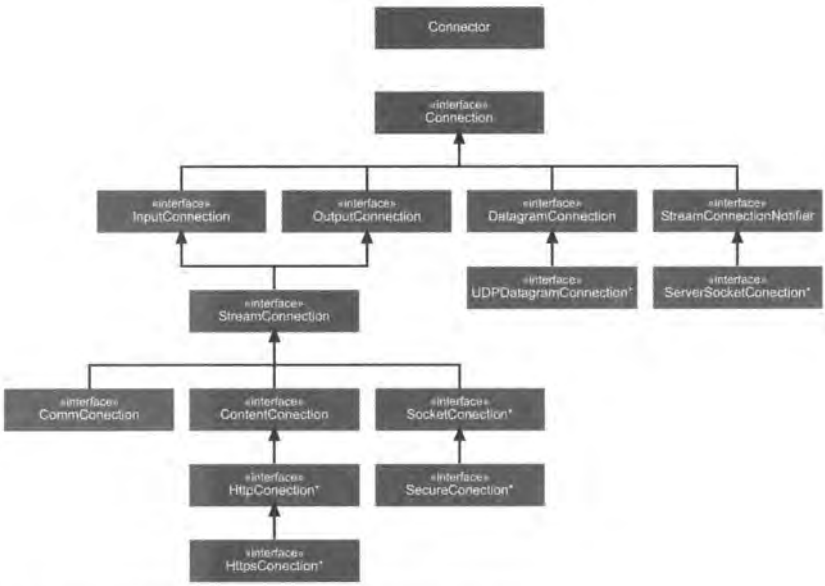


Figura 9.1 Diagrama de classes da *Generic Connection Framework*.

As classes/interfaces são as seguintes:

Connection Interface genérica que representa uma conexão.

InputConnection Este tipo de conexão representa um dispositivo do qual podemos ler dados.

OutputConnection Este tipo de conexão representa um dispositivo no qual podemos escrever dados.

DatagramConnection Define as capacidades básicas que uma conexão por datagramas deve ter.

UDPDatagramConnection Representa uma conexão UDP.

StreamConnectionNotifier Define as capacidades que um notificador de conexões deve possuir. Define apenas o método `acceptAndOpen()` que espera que uma ligação seja efectuada e devolve a `StreamConnection` correspondente.

ServerSocketConnection Define um *socket* de servidor, i.e., um *socket* que aceita ligações de clientes.

StreamConnection Define as capacidades que uma conexão do tipo *stream* deve possuir.

CommConnection Define uma conexão lógica de porta-série.

ContentConnection Define uma conexão do tipo *stream* através da qual passa conteúdo. É uma superinterface para `HttpConnection` e `HttpsConnection`.

HttpConnection Define uma conexão HTTP.

HttpsConnection Define uma conexão HTTP segura.

SocketConnection Define uma conexão do tipo *socket*.

SecureConnection Define uma conexão do tipo *socket* seguro.

Connector Classe Fábrica para a criação de objectos do tipo `Connection`.

Todas as conexões são obtidas através da classe `Connector`[1].

O método `static Connection Connector.open(String name)` devolve uma `Connection` de um tipo que depende do URL passado no parâmetro "`name`". A `Connection` obtida pode depois ser transformada (*cast*) num objecto de uma das outras classes de conexões, e.g.,

```
HttpConnection hc;
hc = (HttpConnection)Connector.open("http://jorgecardoso.org");
```

1. A *Generic Connection framework* utiliza o padrão de desenho de software Fábrica em que uma classe é responsável por criar e devolver instâncias de outras classes (neste caso, classes que representam conexões de rede) com base nos parâmetros passados.

O parâmetro passado deve estar conforme com o formato URI descrito no RFC 2396. Genericamente, toma a forma de:

```
{scheme}:[{target}][{params}]
```

em que

{scheme} é o nome de um esquema, normalmente com o nome de um protocolo, e.g., "http", "socket", "ssl".

{target} é normalmente um endereço de rede, mas pode ser qualquer coisa que identifique o recurso.

{params} parâmetros na forma ";nome=valor".

Exemplos de URI para obtenção de conexões são:

HTTP `Connector.open("http://jorgecardoso.org");`

Socket `Connector.open("socket://193.126.0.1:8000");`

Porta Série `Connector.open("comm:0;baudrate=9600").`

9.2. HTTP

O protocolo mais utilizado para comunicar através da rede é o protocolo HTTP. No caso do MIDP, este é o único protocolo que todos os dispositivos são obrigados a implementar (também são obrigados a implementar a versão segura do protocolo – HTTPS). O protocolo HTTP é um protocolo do tipo *pedido-resposta*, i.e., o cliente efectua um pedido e o servidor devolve uma resposta. Os pedidos mais comuns em HTTP são pedidos de páginas HTML ou outro tipo de ficheiros. As respostas são, normalmente, o conteúdo do ficheiro ou uma indicação de que o ficheiro pedido é desconhecido.

As mensagens (pedidos ou respostas) HTTP são compostas por duas partes. A primeira parte contém os cabeçalhos, que servem, essencialmente, para descrever as capacidades do emissor e para descrever a mensagem. A segunda parte contém o corpo, ou seja, os dados propriamente ditos da mensagem.

Existem vários tipos de pedidos HTTP (designados por *métodos*), mas o MIDP apenas obriga a que sejam implementados três: GET, POST e HEAD.

GET Um pedido simples que indica que o cliente pretende receber o recurso identificado pelo URL. Os (possíveis) parâmetros são passados no próprio URL.

POST Um método que permite que o cliente envie informação para o servidor. A informação é enviada no corpo do pedido, ao invés de no URL, como no método GET. Apesar de ser usado para enviar informação, o cliente pode receber como resposta um ficheiro (e normalmente recebe).

HEAD O método HEAD é em tudo semelhante ao GET, excepto que a resposta do servidor não inclui corpo, apenas os cabeçalhos. Pode ser usado para pedidos que não necessitam de resposta.

O primeiro cabeçalho da resposta HTTP contém um código que indica o estado do pedido. Por exemplo, se a página pedida não existir no servidor, o código de estado será 404 (a linha completa seria *HTTP/1.1 404 Not Found*).

9.2.1. `HttpConnection`

Uma conexão HTTP existe num de três estados possíveis: em preparação (*setup*), conectada ou terminada. O estado de preparação é o estado em que podemos definir os parâmetros da conexão HTTP, *i.e.*, o método utilizado (GET, POST ou HEAD) e os cabeçalhos do pedido. O estado conectado corresponde ao envio do pedido e recepção da resposta. A transição do estado de preparação para o estado conectado é efectuada automaticamente pela invocação de qualquer método da classe `HttpConnection` que implique obter informação relativa à resposta ao pedido HTTP. Esses métodos são:

- `openInputStream`
- `openDataInputStream`
- `getLength`
- `getType`
- `getEncoding`
- `getHeaderField`
- `getResponseCode`
- `getResponseMessage`
- `getHeaderFieldInt`
- `getHeaderFieldDate`
- `getExpiration`
- `getDate`
- `getLastModified`
- `getHeaderFieldKey`

A transição para o estado terminada é feita pela invocação do método `close()`.

9.2.2. GET

O processo típico para fazer um pedido HTTP e obter a resposta é o seguinte:

```
HttpConnection c = null;
InputStream is = null;
```

```
try {
    c = (HttpConnection)Connector.open(url);

    /* definir o método do pedido (GET é usado por omissão) */
    c.setRequestMethod(HttpConnection.GET);

    /* definir os cabeçalhos. Para pedidos simples
       não é necessário definir nenhum cabeçalho.

    */
    c.setRequestProperty("User-Agent", "Dispositivo MID");

    /* obter o código da resposta. Isto irá fazer com que
       o pedido seja enviado e a resposta obtida.
    */
    int código = c.getResponseCode();
    if (código != HttpConnection.HTTP_OK) {
       System.err.println("Código da resposta HTTP: " + código);
    }

    /* ler o corpo da resposta HTTP */
    is = c.openInputStream();
    texto = new StringBuffer();
    byte [] dados = new byte[256];
    int actual = 0;
    while ( actual != -1 ) {
       actual = is.read(dados, 0, 256);
       if (actual != -1) {
           texto.append(new String(dados, 0, actual));
       }
    }
} catch(ClassCastException cce) {
    System.err.println("Excepção:" + cce.getMessage());
} catch (IOException ioe) {
    System.err.println("Excepção:" + ioe.getMessage());
} finally {
    /* libertar os recursos, mesmo que tenha havido uma excepção.
    */
    try {
       if (is != null) {
           is.close();
       }
       if (c != null) {
           c.close();
       }
    } catch (IOException ioe) {
    }
}
```

O método `setRequestMethod(String method)` permite-nos definir o método HTTP utilizado (GET, POST ou HEAD). GET é utilizado por omissão, pelo que só precisamos de invocar este método se quisermos utilizar outro.

Os cabeçalhos HTTP do pedido podem ser definidos com o método

`setRequestProperty(String key, String value)`

em que "`key`" é o nome do cabeçalho, e.g., `Content-type` e "`value`" é o valor do cabeçalho, e.g., `text/html`.

Nalguns casos a resposta ao pedido HTTP inclui um cabeçalho que indica o tamanho, em bytes, do corpo da resposta. Este valor pode ser obtido directamente através do método `getLength()` e pode ser usado para inicializar um *buffer* do tamanho da resposta e para ler a resposta mais rapidamente. Este método é apenas uma forma mais rápida de obter o valor do cabeçalho HTTP `Content-length`.

É necessário ter em atenção que a conexão HTTP não segue redireccionamentos de páginas automaticamente. Cabe ao programador detectar estes casos e tratá-los convenientemente.

O exemplo 9.1 mostra como efectuar um pedido HTTP e obter a resposta de uma forma mais completa. O pedido é feito numa *thread* separada do resto da aplicação e os redireccionamentos de páginas são seguidos automaticamente. Este exemplo consiste numa aplicação que efectua um pedido (o URL é definido pelo utilizador) a um servidor e obtém a resposta. Os cabeçalhos e o corpo da resposta são exibidos em ecrãs diferentes. O pedido é feito por uma *thread* autónoma e pode ser cancelado pelo utilizador a qualquer momento. A aplicação utiliza um `Alert` com um `Gauge` para indicar que o ficheiro está a ser descarregado. A Figura 9.2 mostra os quatro ecrãs da MIDlet do exemplo: o ecrã de introdução do URL, o ecrã de espera enquanto o ficheiro é descarregado, o ecrã com o conteúdo do ficheiro e o ecrã com os cabeçalhos HTTP da resposta obtida.

Exemplo 9.1: `HTTPConexao` – Ligações HTTP

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.*;
import java.io.*;
import javax.microedition.io.*;

import org.jorgecardoso.net.HTTPRedirect;

public class HTTPConexao extends MIDlet implements Runnable,
    CommandListener {

    private Alert espera;
    private Gauge indicador;

    private Form urlForm;
    private TextField urlText;

    private TextBox resultado;
    private TextBox headers;

    private Command cmdSair, cmdBuscar, cmdInicio, cmdHeaders,
    cmdResultado, cmdCancelar;

    private Thread threadCorrente = null;

    private String url;

    public HTTPConexao() {
```

```
        urlText = new TextField("URL", "http://jorgecardoso.org", 255,
            TextField.URL);
        urlForm = new Form("URL");
        urlForm.append(urlText);
        resultado = new TextBox("Resultado", "", 256, TextField.ANY);
        headers = new TextBox("Headers HTTP", "", 256, TextField.ANY);

        indicador = new Gauge(null, false, Gauge.INDEFINITE, 0);
        espera = new Alert("A Descarregar");
        espera.setIndicator(indicador);
        espera.setTimeout(Alert.FOREVER);

        cmdSair = new Command("Sair", Command.EXIT, 1);
        cmdBuscar = new Command("Buscar", Command.SCREEN, 1);
        cmdInicio = new Command("Inicio", Command.SCREEN, 2);
        cmdHeaders = new Command("Headers", Command.SCREEN, 2);
        cmdResultado = new Command("Resultado", Command.SCREEN, 2);
        cmdCancelar = new Command("Cancelar", Command.CANCEL, 1);

        urlForm.addCommand(cmdSair);
        urlForm.addCommand(cmdBuscar);
        urlForm.setCommandListener(this);

        resultado.addCommand(cmdSair);
        resultado.addCommand(cmdInicio);
        resultado.addCommand(cmdHeaders);
        resultado.setCommandListener(this);

        headers.addCommand(cmdSair);
        headers.addCommand(cmdInicio);
        headers.addCommand(cmdResultado);
        headers.setCommandListener(this);

        espera.addCommand(cmdCancelar);
        espera.setCommandListener(this);
    }

    public void startApp() {
        System.out.println("StartApp");
        Display.getDisplay(this).setCurrent(urlForm);
    }

    public void destroyApp(boolean unconditional) {
        pararThread();
    }

    public void pauseApp() {
        System.out.println("PauseApp");
    }

    public void commandAction(Command c, Displayable d) {
        if (c == cmdSair) {
            destroyApp(true);
            notifyDestroyed();
        } else if (c == cmdCancelar) {
            pararThread();
            Display.getDisplay(this).setCurrent(urlForm);
        } else if (c == cmdBuscar) {
            Display.getDisplay(this).setCurrent(espera);
            url = urlText.getString();
            threadCorrente = new Thread(this);
            threadCorrente.start();
        } else if (c == cmdInicio) {
```

```
            Display.getDisplay(this).setCurrent(urlForm);
        } else if (c == cmdHeaders) {
            Display.getDisplay(this).setCurrent(headers);
        } else if (c == cmdResultado) {
            Display.getDisplay(this).setCurrent(resultado);
        }
    }

    private void pararThread() {
        Thread t = threadCorrente;

        threadCorrente = null;
        if (t != null) {
            try {
                t.join();
            } catch (InterruptedException ie) {
                System.err.println("Excepção:" + ie.getMessage());
            }
        }
    }

    public void run() {
        Thread estaThread = Thread.currentThread();

        HttpConnection c = null;
        InputStream is = null;

        if (estaThread != threadCorrente) {
            return;
        }

        indicador.setValue(Gauge.INCREMENTAL_UPDATING);
        try {

            System.out.println(estaThread + ":" + url);
            c = (HttpConnection)Connector.open(url);

            /* ler os cabeçalhos HTTP */
            String field, key;
            StringBuffer texto = new StringBuffer();
            int i = 0;
            do {
                field = c.getHeaderField(i);
                key = c.getHeaderFieldKey(i);
                if (key != null && key.equals("location")) {

                    url = HTTPRedirect.redirectURL(url, field);

                    /* antes de lançarmos outra thread vamos ver se
                       esta nao foi terminada pelo utilizador
                    */
                    if (estaThread != threadCorrente) {
                        return;
                    }
                    threadCorrente = new Thread(this);
                    threadCorrente.start();
                    return;
                }
                i++;
                if (key != null) {
                    texto.append(key).append(":").append(field).append("\n");
                    System.out.println(key + ":" + field);
                }
```

```
} while (key != null);

/* colocar os cabeçalhos na caixa de texto */
if (texto.length() > 0) {
    headers.setMaxSize(texto.length());
    headers.setString(texto.toString());
} else {
    headers.setString("");
}

/* indicar progresso */
indicador.setValue(Gauge.INCREMENTAL_UPDATING);

/* verificar se o utilizador cancelou */
if (estaThread != threadCorrente) {
    return;
}

/* ler o corpo da resposta HTTP */
is = c.openInputStream();

/* se soubermos o tamanho da resposta, podemos tentar ler tudo
   de uma só vez.
*/
int len = (int) c.getLength();
texto = new StringBuffer();
if (len > 0) {
    int actual = 0;
    int byteslidos = 0;
    byte [] dados = new byte[len];

    while ((byteslidos != len) && (actual != -1)) {
        actual = is.read(dados, byteslidos, len-byteslidos);

        byteslidos += actual;

        /* verificar cancelamento e indicar progresso */
        if (estaThread != threadCorrente) {
            return;
        }
        indicador.setValue(Gauge.INCREMENTAL_UPDATING);
    }

    texto.append(new String(dados));

} else {
    /* não sabemos o tamanho da resposta, por isso, vamos ler
       aos bocados até terminar.
    */
    byte [] dados = new byte[256];
    int actual = 0;
    while ( actual != -1 ) {
        actual = is.read(dados, 0, 256);

        if (actual != -1) {
            texto.append(new String(dados, 0, actual));
        }

        /* verificar cancelamento e indicar progresso */
        if (estaThread != threadCorrente) {
            return;
        }
        indicador.setValue(Gauge.INCREMENTAL_UPDATING);
```

```
                }

            }
            /* se o corpo da mensagem tiver algum conteúdo, vamos afixá-lo
               na caixa de texto 'resultado'. Se não vamos limpar a caixa
               de texto.
            */
            if (texto.length() > 0) {
                resultado.setMaxSize(texto.length());
                resultado.setString(texto.toString());
            } else {
                resultado.setString("");
            }

        } catch(ClassCastException cce) {
            System.err.println("Excepção:" + cce.getMessage());
        } catch (IOException ioe) {
            System.err.println("Excepção:" + ioe.getMessage());
        } finally {
            /* libertar os recursos, mesmo que tenha havido uma excepção.
            */
            try {
                if (is != null) {
                    is.close();
                }
                if (c != null) {
                    c.close();
                }
            } catch (IOException ioe) {
            }
        }

        /* mostrar o resultado */
        Display.getDisplay(this).setCurrent(resultado);
    }
}
```

(a) Introdução do URL (b) Ecrã de espera

(c) Resultado (d) Cabeçalhos HTTP

Figura 9.2 `HTTPConexao` – Pedidos HTTP.

Parâmetros codificados no URL

Se quisermos utilizar uma ligação HTTP para enviar dados para o servidor para serem tratados por uma *servlet* ou por uma página PHP, por exemplo, é necessário enviar esses dados como parâmetros do pedido. No caso de pedidos GET, os parâmetros são codificados no próprio URL:

```
http://servidor/pagina.php?parametro1=valor1&parametro2=valor2
```

Os pares *parâmetro-valor* têm de ser codificados antes de serem enviados para o servidor. Esta codificação corresponde ao *application/x-www-form-urlencoded mime type* e as regras para converter são as seguintes:

- Os caracteres alfanuméricos permanecem iguais, assim como o ponto final(.), o hífen (-), o asterisco (*) e o *underscore* (_).
- O espaço é substituído por "+".
- Todos os outros caracteres devem ser substituídos pelo valor hexadecimal que representa o carácter, precedido pelo símbolo "%".

Esta codificação é aplicada apenas aos nomes dos parâmetros e respectivos valores. Os símbolos "=" e "&" devem permanecer inalterados.

Por exemplo, o URL

`http://jorgecardoso.org/anagrama.php?palavra=j(o)r ge`

deverá ser enviado para o servidor como

`http://jorgecardoso.org/anagrama.php?palavra=j%28o%29r+ge`

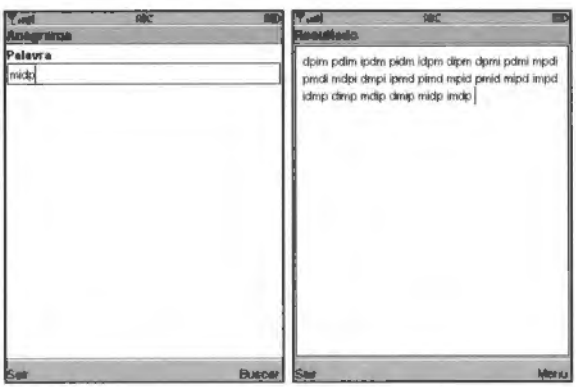
Podemos agora construir uma MIDlet que envia um parâmetro para um *script* PHP[2] e obtém o resultado. O Exemplo 9.2 é baseado no Exemplo 9.1 e sofreu poucas alterações, pelo que mostro apenas essas alterações. Basicamente, a aplicação permite ao utilizador escrever uma palavra que é enviada para um *script* PHP no servidor que constrói uma lista com todos os anagramas dessa palavra e devolve essa lista. Uma vez que a palavra pode conter qualquer carácter, é necessário codificá-la segundo as regras descritas anteriormente. Essa codificação é feita no método `URLEncoder.encode(String s)` (o código deste método pode ser consultado *online*). A Figura 9.3 mostra dois dos ecrãs da MIDlet. O primeiro é o ecrã em que o utilizador pode introduzir a palavra. O segundo mostra o resultado obtido do servidor.

Exemplo 9.2: **`HTTPGETParametro`** – Enviar parâmetros no URL

```
[...]
import org.jorgecardoso.net.URLEncoder;

public class HTTPGETParametro extends MIDlet implements Runnable,
    CommandListener{

    private final static String URLBASE =
    "http://jorgecardoso.org/misc/anagrama.php?palavra=";

    private Alert espera;
    private Gauge indicador;

    private Form palavraForm;
    private TextField palavraText;

    [...]

    private String url;
```

2. Implementei o *script* do lado do servidor em PHP porque o meu fornecedor do serviço Web não me permite disponibilizar páginas JSP ou servlets. De qualquer forma, isto serve também para mostrar que os programadores Java são poliglotas ;).

```
public HTTPGETParametro() {
   palavraText = new TextField("Palavra", "midp", 255,
      TextField.ANY);
   palavraForm = new Form("Anagrama");
   palavraForm.append(palavraText);

   [...]
}

public void run() {

   [...]

   indicador.setValue(Gauge.INCREMENTAL_UPDATING);
   try {

      url = URLBASE + URLEncoder.encode(palavraText.getString());
      c = (HttpConnection)Connector.open(url);

      [...]
   }
}
```



(a) Introdução da palavra (b) Lista de anagramas

Figura 9.3 `HTTPGETParametro` – MIDlet que obtém lista de anagramas.

9.2.3. POST

A utilização do método HTTP POST para enviar informação é muito semelhante ao que vimos para o método HTTP GET. A principal diferença é que, neste caso, temos de escrever no corpo do pedido HTTP.

Vou mostrar aqui apenas o código necessário para alterar o exemplo anterior de forma a enviar os parâmetros através do método POST em vez de GET:

```
/* já não enviamos os parâmetros no URL... */
private final static String URLBASE =
"http://jorgecardoso.org/misc/anagrama.php";
```

```
[...]

public void run() {
    Thread estaThread = Thread.currentThread();

    HttpConnection c = null;
    InputStream is = null;
    OutputStream os = null;

    if (estaThread != threadCorrente) {
       return;
    }

    indicador.setValue(Gauge.INCREMENTAL_UPDATING);
    try {
       url = URLBASE;
       dados = "palavra=" + URLEncoder.encode(palavraText.getString());

       c = (HttpConnection)Connector.open(url);

       /* indicar que queremos utilizar POST (GET é usado por omissão) */
       c.setRequestMethod(HttpConnection.POST);

       /* indicar o tamanho dos dados (não é obrigatório) */
       c.setRequestProperty("Content-Length",
           Integer.toString(dados.length()));

       /* temos de indicar o MIME Type dos dados */
       c.setRequestProperty("Content-Type",
           "application/x-www-form-urlencoded");

       /* escrever os dados */
       os = c.openOutputStream();
       os.write(dados.getBytes());
       os.close();


       /* ler os cabeçalhos HTTP */
       String field, key;
       StringBuffer texto = new StringBuffer();
       [...]
```

O *script* PHP utilizado é o mesmo do exemplo anterior. Este *script* está preparado para receber dados através de GET e POST.

9.3. HTTP seguro

O protocolo HTTP é um protocolo inseguro uma vez que os dados são transmitidos sem encriptação. No entanto, existe uma versão segura do mesmo protocolo – o HTTPS. Todos os dispositivos MIDP 2.0 são obrigados a implementar ambas as versões.

A maneira como se comunica através de HTTPS, em MIDP, é praticamente igual à forma como se comunica através de HTTP. A única diferença é a conexão utilizada e o protocolo do URL:

```
HttpsConnection c = (HttpsConnection)Connector.open("https://www.rsa.com");
```

Depois de termos a conexão, podemos obter informação relativa à ligação segura e ao certificado do sítio Web:

```
/* ler a informação do certificado */
SecurityInfo si = c.getSecurityInfo();

String cipherSuite = si.getCipherSuite();
String protocolo = si.getProtocolName();
String versãoProtocolo = si.getProtocolVersion();

Certificate cert = si.getServerCertificate();

String emissor = cert.getIssuer();
long inválidoAntesDe = cert.getNotBefore();
long inválidoApós = cert.getNotAfter();
String númeroSérie = cert.getSerialNumber();
String algoritmo = cert.getSigAlgName();
String sujeito = cert.getSubject();
String tipo = cert.getType();
String versão = cert.getVersion();
```

A interface `Certificate` está definida no pacote `javax.microedition.pki`.

9.4. *Sockets*

As ligações que temos vindo a utilizar até aqui são feitas através de um protocolo de aplicação – o HTTP. Isto significa que os dados transmitidos têm de obedecer a determinadas regras impostas pelo protocolo, i.e., os pedidos são estruturados em cabeçalho e corpo do pedido, alguns cabeçalhos são obrigatórios, etc.

No entanto, podemos ter necessidade de usar o nosso próprio protocolo. Para isso precisamos de obter uma ligação de mais baixo nível do que a `HttpConnection`. A classe `SocketConnection` permite-nos manipular conexões do tipo *sockets*.

Para obtermos um *socket* temos de especificar um URL do género `socket://host:port`, no parâmetro do método `Connector.open()`.

Para ilustrar o uso de conexões do tipo *socket* vamos implementar uma aplicação que permita fazer ligações a servidores POP3 e enviar comandos.

O protocolo POP3 (*Post Office Protocol* versão 3) é um protocolo para aceder às mensagens de correio electrónico de um utilizador. Os servidores POP3 respondem a um conjunto de comandos:

USER ***nome do utilizador*** Este comando e o seguinte são usados para autenticar o utilizador perante o servidor POP3.

PASS ***senha***

STAT Mostra o número de mensagens e o tamanho ocupado.

LIST Mostra o tamanho de cada mensagem.

RETR ***número_da_mensagem*** Mostra o conteúdo da mensagem identificada pelo número.

DELE ***número_da_mensagem*** Apaga a mensagem.

QUIT Termina a ligação.

TOP ***número_da_mensagem número_de_linhas*** Mostra *número_de_linhas* linhas da mensagem *número_da_mensagem*.

Um exemplo simples de interacção entre um cliente POP3 e um servidor é o seguinte (o nome do utilizador e a senha foram alterados):

```
telnet pop.jorgecardoso.org 110
+OK <93008.1122211543@64.34.66.51>
USER jc@jorgecardoso.org
+OK
PASS *****
+OK
STAT
+OK 4 3198
LIST
+OK
1 875
2 854
3 887
4 582
.
TOP 4 10
+OK
Return-Path: <jc@jorgecardoso.org>
Delivered-To: jorgecardoso.org-jc@jorgecardoso.org
Received: (qmail 97752 invoked from network); 24 Jul 2005 13:25:18 -0000
Received: from unknown (10.8.7.1)
 by 0 with QMQP; 24 Jul 2005 13:25:18 -0000
Date: 24 Jul 2005 13:25:18 -0000
Message-ID: <20050724132518.12719.qmail@wm1>
From: <jc@jorgecardoso.org>
To: jc@jorgecardoso.org
Subject: Teste SocketPOP3
X-Mailer: Netfirms Mailing - http://www.netfirms.com
X-IPAddress: 213.58.96.252
X-Sender: jc@jorgecardoso.org

Teste SocketPOP3
Fim Teste

.
quit
+OK
```

As respostas do servidor começam sempre por +OK em caso de sucesso ou por -ERR em caso de erro.

A aplicação utiliza duas *threads*: uma para enviar os comandos introduzidos pelo utilizador para o servidor e outra para receber o resultado da execução dos comandos.

Os comandos são introduzidos num `Vector` e, sempre que um comando é introduzido, a *thread* que envia os comandos é notificada, utilizando o mecanismo de notificações discutido no capítulo anterior:

```
private void adicionaComando(String comando) {
    synchronized(comandos) {
       comandos.addElement(comando);
       comandos.notify();
    }
}
```

Sempre que um comando for adicionado, a *thread* irá enviá-lo para o servidor:

```
while (!pararThread) {
    comando = null;
    synchronized(comandos) {
       if (comandos.size() > 0) {
           comando = (String)comandos.elementAt(0);
           comandos.removeElementAt(0);
       } else {
           try {
               comandos.wait();
           } catch (InterruptedException ie) {

           }
       }
    }
    if (comando != null) {
       try {
           System.out.println("Comando: " + comando);
           os.write(comando.getBytes());
           os.write("\r\n".getBytes());

       } catch (IOException ioe) {
           System.err.println("Excepção de IO: " + ioe.getMessage());
       }
    }
}
```

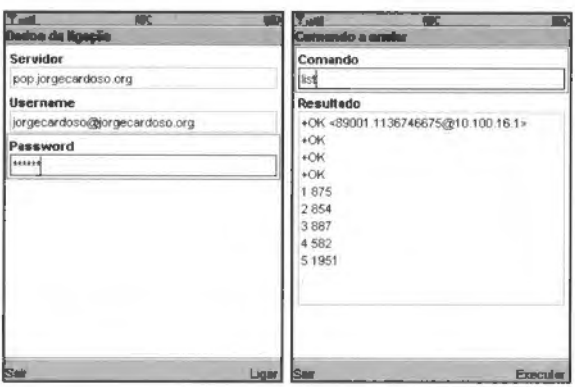
A outra *thread*, definida na classe `Leitor`, limita-se a ler continuamente da conexão, uma linha de cada vez, e a informar a aplicação de que mais informação foi lida:

```
public void run() {
    int lidos = 0;

    while (corre) {
       try {
           receptor.novosDados(lerLinha(isr));
       } catch (IOException ioe) {
           System.err.println("run:" + ioe.getMessage());
       }
    }
}

/**
```

```
    * Lê uma linha da resposta do servidor POP3. Todas as linhas
    * terminam com \r\n.
    */
   private String lerLinha(InputStreamReader isr) throws IOException {
       int c;
       StringBuffer sb = new StringBuffer();

       do {
          c = isr.read();
          if ((char)c == '\r') {
             c = isr.read(); // ler \n
          }
          if (c != -1 && (char)c != '\n') {
             sb.append((char)c);
          }
       } while (c != -1 && (char)c != '\n');
       if (sb.length() > 0 ) {
          return sb.append("\n").toString();
       } else {
          return null;
       }
   }
```

A Figura 9.4 mostra os dois ecrãs da aplicação. O primeiro contém os dados necessários para efectuar a ligação ao servidor. O segundo é o ecrã que permite executar comandos e ver o resultado. Podemos ver na figura o resultado da execução do comando `list`, assim como o resultado da autenticação.

O código completo do exemplo é listado a seguir.



(a) Ecrã de ligação (b) Ecrã de execução de comandos

Figura 9.4 SocketPOP3 – Ligações *socket*

Exemplo 9.3: **SocketPOP3** – Conexões através de *sockets*

```
[Ficheiro SocketPOP3.java]
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.*;
import java.io.*;
import javax.microedition.io.*;
import java.util.*;

public class SocketPOP3 extends MIDlet implements Runnable,
    CommandListener {

    /* Os dados para ligação ao servidor POP3. */
    private Form formLigação;
    private TextField servidorText;
    private TextField utilizadorText;
    private TextField passwordText;

    /* O ecrã de comandos a enviar */
    private Form formComando;
    private TextField comandoText;

    /* O resultado da execução de um comando */
    private TextField resultado;

    /* Os comandos da aplicação */
    private Command cmdSair, cmdLigar, cmdBuscar, cmdComando;

    /* A thread que envia os comandos para o servidor */
    private Thread executor = null;

    /* Flag para interromper a thread */
    private boolean pararThread = false;

    /* A lista de comandos a enviar, funciona como uma pilha */
    private Vector comandos = new Vector();

    public SocketPOP3() {
       cmdSair = new Command("Sair", Command.EXIT, 1);
       cmdBuscar = new Command("Executar", Command.SCREEN, 1);
       cmdLigar = new Command("Ligar", Command.SCREEN, 1);
       cmdComando = new Command("Comando", Command.SCREEN, 1);

       formLigação = new Form("Dados da ligação");
       servidorText = new TextField("Servidor",
           "pop.jorgecardoso.org", 256, TextField.ANY);
       utilizadorText = new TextField("Username",
           "jorgecardoso@jorgecardoso.org", 256, TextField.ANY);
       passwordText = new TextField("Password", "", 32,
           TextField.PASSWORD);
       formLigação.append(servidorText);
       formLigação.append(utilizadorText);
       formLigação.append(passwordText);
       formLigação.addCommand(cmdSair);
       formLigação.addCommand(cmdLigar);
       formLigação.setCommandListener(this);

       formComando = new Form("Comando a enviar");
       comandoText = new TextField("Comando", "", 256, TextField.ANY);
       resultado = new TextField("Resultado", "", 256, TextField.ANY);
       formComando.append(comandoText);
       formComando.append(resultado);
       formComando.addCommand(cmdSair);
```

```
        formComando.addCommand(cmdBuscar);
        formComando.setCommandListener(this);
    }

    public void startApp() {
        Display.getDisplay(this).setCurrent(formLigação);
    }

    public void destroyApp(boolean unconditional) {
        pararThread();
    }

    public void pauseApp() {
    }

    public void commandAction(Command c, Displayable d) {
        if (c == cmdSair) {
            destroyApp(true);
            notifyDestroyed();
        } else if (c == cmdLigar) {
            Display.getDisplay(this).setCurrent(formComando);
            executor = new Thread(this);
            executor.start();
            adicionaComando("user " + utilizadorText.getString());
            adicionaComando("pass " + passwordText.getString());
            adicionaComando("list");
        } else if (c == cmdComando) {
            Display.getDisplay(this).setCurrent(formComando);
        } else if (c == cmdBuscar) {
            adicionaComando(comandoText.getString());
        }
    }

    private void adicionaComando(String comando) {
        synchronized(comandos) {
            comandos.addElement(comando);
            comandos.notify();
        }
    }

    private void pararThread() {
        pararThread = true;
        synchronized(comandos) {
            comandos.notify();
        }
        if (executor != null) {
            try {
                executor.join();
            } catch (InterruptedException ie) {
                System.err.println("Excepção:" + ie.getMessage());
            }
        }
    }

    public void run() {

        InputStream is = null;
        OutputStream os = null;
        String comando = null;
        SocketConnection sc = null;
        /* criar a ligação socket ao servidor */
        try {
```

```
            sc = (SocketConnection) Connector.open("socket://" +
                servidorText.getString()+":110");
            is = sc.openInputStream();
            os = sc.openOutputStream();
        } catch (IOException ioe) {
            Alert a = new Alert("Erro", ioe.getMessage(), null, null);
            a.setTimeout(Alert.FOREVER);
            Display.getDisplay(this).setCurrent(a, formLigação);
            return;
        }
        Leitor l = new Leitor(this);
        l.começar(is);

        while (!pararThread) {
            comando = null;
            synchronized(comandos) {
                if (comandos.size() > 0) {
                    comando = (String)comandos.elementAt(0);
                    comandos.removeElementAt(0);
                } else {
                    try {
                        comandos.wait();
                    } catch (InterruptedException ie) {
                    }
                }
        }
            if (comando != null) {
                try {
                    System.out.println("Comando: " + comando);
                    os.write(comando.getBytes());
                    os.write("\r\n".getBytes());

                } catch (IOException ioe) {
                    System.err.println("Excepção de IO: " + ioe.
                    getMessage());
                }
            }

        }
        try {
            if (is != null) {
                is.close();
            }
            if (os != null) {
                os.close();
            }
            if (sc != null) {
                sc.close();
            }
        } catch (IOException ioe) {
                System.err.println(ioe.getMessage());
        }
        l.parar();
    }

    public void novosDados(String linha) {
        if (linha == null) {
            return;
        }
        resultado.setMaxSize(resultado.getString().length()+
            linha.length());
        resultado.setString(resultado.getString()+linha);
    }
}
```

```
[Ficheiro Leitor.java]
import java.io.*;

public class Leitor implements Runnable {

    private boolean corre = false;

    private InputStreamReader isr = null;

    private SocketPOP3 receptor = null;

    private Thread leitor = null;

    public Leitor(SocketPOP3 receptor) {
       this.receptor = receptor;
    }

    public void parar() {
       corre = false;

       if (isr != null) {
           try {
               isr.close();
           } catch (IOException ioe) {
               System.err.println(ioe.getMessage());
           }
       }

       if (leitor != null) {
           try {
               leitor.join();
           } catch (InterruptedException ie) {
           }
       }
    }

    public void começar(InputStream is) {
       parar();
       this.isr = new InputStreamReader(is);
       corre = true;
       leitor = new Thread(this);
       leitor.start();
    }

    public void run() {
       int lidos = 0;

       while (corre) {
           try {
               receptor.novosDados(lerLinha(isr));
           } catch (IOException ioe) {
               System.err.println("run:" + ioe.getMessage());
           }
       }
    }

    /**
     * Lê uma linha da resposta do servidor POP3. Todas as linhas
     * terminam com \r\n.
     */
    private String lerLinha(InputStreamReader isr) throws IOException {
       int c;
       StringBuffer sb = new StringBuffer();
```

```
        do {
            c = isr.read();
            if ((char)c == '\r') {
                c = isr.read(); // let \n
            }
            if (c != -1 && (char)c != '\n') {
                sb.append((char)c);
            }
        } while (c != -1 && (char)c != '\n');
        if (sb.length() > 0 ) {
            return sb.append("\n").toString();
        } else {
            return null;
        }
    }
}
```

9.4.1. *Sockets* seguros

As ligações *socket* seguras são feitas definindo o protocolo SSL no URL da ligação e utilizando a classe `SecureConnection`:

```
SecureConnection sc;
sc = (SecureConnection)Connector.open("ssl://servidor:porta");
```

A `SecureConnection`, à semelhança do que se passava com a `HttpsConnection`, permite-nos também obter informação sobre a ligação segura através do método `getSecurityInfo()`.

De resto, trabalhamos com *sockets* seguros da mesma forma que o fazemos com *sockets* normais.

9.4.2. *Sockets* de servidor

Até agora temos utilizado sempre as ligações para agir como ligações cliente a servidores. No entanto, é possível tornar o nosso próprio telemóvel num servidor, *i.e.*, num dispositivo que aceita conexões externas. Para tal utilizamos a conexão `ServerSocketConnection`. A utilização desta classe é feita de forma análoga ao que temos visto até agora, a diferença é que não especificamos um endereço completo quando criamos a ligação. O endereço utilizado pelo *socket* é determinado pelo sistema.

A sequência de passos típica para criar e utilizar um `ServerSocketConnection` é a seguinte:

```
try {
    /* criar o socket do servidor */
    ServerSocketConnection ssc;
    ssc = (ServerSocketConnection)Connector.open("socket://");

    /* determinar o endereço atribuido */
    String endereço = ssc.getLocalAddress();
    int porta = ssc.getLocalPort();
```

```
        while (servidorActivo) {
            /* esperar que uma ligação externa seja feita */
            StreamConnection sc = ssc.acceptAndOpen();

            InputStream is = sc.openInputStream();
            [ler dados]

            OutputStream os = sc.openOutputStrem();
            [escrever resposta]
        }
    } catch (IOException ioe) {
    }
```

O método `acceptAndOpen()` bloqueia até que uma ligação externa seja feita. Nessa altura, devolve uma referência para um `StreamConnection` que pode ser usado para ler a informação enviada pelo cliente e para escrever a resposta. Há que ter em atenção que, uma vez que este método bloqueia, a terminação da *thread* tem de ser feita de forma diferente da que temos utilizado.

Para desbloquear o método `acceptAndOpen()` temos de fechar a ligação utilizada. Isto irá fazer com que o método lance uma `IOException`, que a nossa aplicação deverá apanhar e determinar que a *thread* deve ser terminada.

9.5. Datagramas

As conexões por datagramas são conexões *connectionless* (sim, isto parece contraditório, mas é assim mesmo). Ao contrário de uma ligação HTTP, por exemplo, não existe uma ligação lógica entre os dois pontos da ligação (cliente e servidor). O HTTP funciona sobre o protocolo de rede TCP. Isto significa que, para comunicarmos, temos primeiro de estabelecer uma ligação ao destino e só depois enviamos a informação. O TCP garante-nos que a informação que enviamos chega ao destino, e na ordem correcta. Isto acontece assim, porque, por detrás das cortinas, se estabeleceu um caminho entre os dois pontos da ligação.

As ligações por datagramas funcionam sobre o protocolo UDP. Ao contrário do TCP, não é necessário criar uma ligação. A informação que enviamos está inserida num datagrama que contém o endereço do destino. Uma vez que não há estabelecimento de uma ligação, a comunicação por UDP é normalmente mais rápida. Por outro lado, o protocolo UDP nada pode garantir relativamente à entrega dos datagramas, nem relativamente à ordem a que chegam ao destino. Isto significa que um determinado datagrama pode nunca chegar ao destino ou que dois datagramas A e B enviados por esta ordem podem chegar pela ordem B, A. Por esta razão, o UDP utiliza-se normalmente quando não é muito importante garantir que toda a informação chega ao destino, mas é importante enviar a informação o mais rápido possível.

Tal como no resto das ligações que vimos até agora, uma ligação por datagramas cria-se através do `Connector`, utilizando o esquema `datagram`:

```
DatagramConnection dc;
dc = (DatagramConnection)Connector.open("datagram://169.254.2.10:6789");
```

As conexões por datagramas podem ser abertas em modo *cliente* ou em modo *servidor* consoante o URL de ligação. Se especificarmos "datagram://:1234", sem definirmos o nome ou endereço IP do servidor, então a ligação é aberta em modo servidor; i.e., uma aplicação cliente irá iniciar a comunicação. Se especificarmos o nome ou endereço IP do servidor, então a ligação será aberta em modo cliente. No modo servidor, a porta especificada é a porta local que irá receber os datagramas. Em modo cliente, a porta é a porta do servidor que está à escuta de ligações.

9.5.1. Ligações cliente

No caso de uma ligação em modo cliente, depois de termos obtido a conexão, podemos enviar datagramas, usando o método

`DatagramConnection.send(Datagram dgram)`.

Para obtermos um objecto `Datagram` temos de invocar um dos quatro métodos `DatagramConnection.newDatagram()`:

- `Datagram newDatagram(byte[] buf, int size)`
  Cria um novo datagrama, utilizando o *buffer* e o tamanho especificados. O valor do parâmetro "`size`" deve ser menor ou igual ao tamanho de "`buf`".
- `Datagram newDatagram(byte[] buf, int size, String addr)`
  Cria um novo datagrama com o *buffer*, tamanho e endereço de destino especificados. O endereço – "`addr`" – deve estar na forma "datagram://servidor:porta".
- `Datagram newDatagram(int size)`
  Cria um novo datagrama, alocando um *buffer* do tamanho especificado por "`size`".
- `Datagram newDatagram(int size, String addr)`
  Cria um novo datagrama, alocando um *buffer* do tamanho especificado e com o endereço de destino definido por "addr".

Todos os datagramas têm um *buffer* que contém os dados a enviar e um endereço de destino para o qual o datagrama será enviado. No caso de não especificarmos um endereço de destino quando criamos um datagrama, será utilizado o endereço definido aquando da criação da conexão.

Apesar de definirmos um endereço quando criamos a conexão, nada nos impede de enviar datagramas para outros endereços, usando a mesma conexão. Basta colocar no datagrama o endereço de destino pretendido.

Depois de obtermos um `Datagram` podemos inserir os dados a transmitir (caso não tenhamos criado o `Datagram` já com o *buffer* ou caso estejamos a reutilizar um `Datagram`) com o método `setData(byte[] buffer, int offset, int len)`.

Depois de enviarmos o datagrama, podemos querer receber uma resposta do servidor. Para tal podemos invocar o método

`DatagramConnection.receive(Datagram dgram)`

Este método bloqueia até receber um datagrama.

No caso das ligações em modo cliente, a porta que recebe os datagramas é alocada dinamicamente pelo sistema. No caso de estarmos interessados em conhecer a porta local (é esta a porta que o servidor irá contactar para enviar a resposta ao pedido) temos de utilizar a interface *`UDPDatagramConnection`*, em vez de *`DatagramConnection`*:

```
UDPDatagramConnection dc;
dc = (UDPDatagramConnection)
     Connector.open("datagram://169.254.2.10:6789");

int portaLocal = dc.getLocalPort();
String endereçoLocal = dc.getLocalAddress();
```

9.5.2. Escrever e ler tipos primitivos

A classe `Datagram` fornece alguns mecanismos para facilitar a escrita e leitura de tipos de dados primitivos, de forma semelhante às classes `DataInputStream` e `DataOutpuStream`. Uma vez que `Datagram` implementa as interfaces `DataInput` e `DataOutput`, podemos utilizar os seguintes métodos para ler e escrever no datagrama:

- `boolean readBoolean(), writeBoolean(boolean v)`
- `byte readByte(), writeByte(int v), write(int b)`
- `char readChar(), writeChar(int v), writeChars(String s)`
- `void readFully(byte[] b), write(byte[] b)`
- `void readFully(byte[] b, int off, int len),`
- `write(byte[] b, int off, int len)`
- `int readInt(), writeInt(int v)`
- `long readLong(), writeLong(long v)`
- `short readShort(), writeShort(int v)`
- `int readUnsignedByte()`
- `int readUnsignedShort()`
- `String readUTF(), writeUTF(String str)`
- `int skipBytes(int n)`

Para ilustrar o uso de datagramas, vamos implementar uma aplicação que nos permite enviar mensagens para um servidor de eco, *i.e.*, um servidor que responde com a própria mensagem recebida, através de UDP. Para além da mensagem do cliente, o servidor devolve o instante em que recebeu a mensagem, seguido da mensagem propriamente dita. O instante é representado por um *long* e a mensagem é enviada no formato UTF-8.

O código mais importante correspondente ao servidor (Java SE) é o seguinte:

```
try {
     aSocket = new DatagramSocket(Integer.parseInt(args[0]));

     while(true) {
        /* receber um pedido */
        DatagramPacket request = new DatagramPacket(buffer, buffer.length);
        aSocket.receive(request);

        /* determinar o endereço do cliente para lhe enviarmos a resposta */
        endereçoIP = request.getAddress();
        porta = request.getPort();

        String pedidoCliente = new String(request.getData(), 0,
        request.getLength());

        /* escrever o timestamp e a frase do cliente */
        long timeStamp = System.currentTimeMillis();

        baos = new ByteArrayOutputStream();
        DataOutputStream dos = new DataOutputStream(baos);

        dos.writeLong(timeStamp);
        dos.writeUTF(pedidoCliente);

        byte buf[] = baos.toByteArray();

        /* enviar para o cliente */
        DatagramPacket response = new DatagramPacket(buf, buf.length,
            endereçoIP, porta);
        aSocket.send(response);
     }
} catch (SocketException e){
     System.out.println("Socket: " + e.getMessage());
}catch (IOException e) {
     System.out.println("IO: " + e.getMessage());
}finally {
     if(aSocket != null) {
        aSocket.close();
     }
}
```

A MIDlet do Exemplo 9.4 permite enviar mensagens para o servidor e receber a resposta. A MIDlet lê os dois campos da resposta (instante em que o servidor recebeu o datagrama e a mensagem) e apresenta-os ao utilizador, formatando o instante como uma data. A Figura 9.5 mostra o ecrã da aplicação.

Exemplo 9.4: `DatagramEco` – Conexões através de datagramas

```
import javax.microedition.midlet.MIDlet;
```

```
import javax.microedition.lcdui.*;
import java.io.*;
import javax.microedition.io.*;
import java.util.*;

public class DatagramEco extends MIDlet implements Runnable,
    CommandListener {

    private Form formEco;
    private TextField mensagem;
    private TextField eco;

    /* Os comandos da aplicação */
    private Command cmdSair, cmdBuscar, cmdComando;

    /* A thread que envia os comandos para o servidor */
    private Thread executor = null;

    /* Flag para interromper a thread */
    private boolean pararThread = false;

    /* A lista de comandos a enviar, funciona como uma pilha */
    private Vector comandos = new Vector();

    private UDPDatagramConnection udpDC = null;

    public DatagramEco() {
       cmdSair = new Command("Sair", Command.EXIT, 1);
       cmdBuscar = new Command("Enviar", Command.SCREEN, 1);
       cmdComando = new Command("Comando", Command.SCREEN, 1);

       formEco = new Form("Mensagem");
       mensagem = new TextField("Mensagem", "Ola", 256, TextField.ANY);
       eco = new TextField("Eco", "", 256, TextField.ANY);
       formEco.append(mensagem);
       formEco.append(eco);
       formEco.addCommand(cmdSair);
       formEco.addCommand(cmdBuscar);
       formEco.addCommand(cmdSair);
       formEco.setCommandListener(this);
       executor = new Thread(this);
       executor.start();
    }

    public void startApp() {
       Display.getDisplay(this).setCurrent(formEco);
    }

    public void destroyApp(boolean unconditional) {
       pararThread();
    }

    public void pauseApp() {
    }

    public void commandAction(Command c, Displayable d) {
       if (c == cmdSair) {
           destroyApp(true);
           notifyDestroyed();
       } else if (c == cmdBuscar) {
           adicionaComando(mensagem.getString());
       }
    }
```

```
private void adicionaComando(String comando) {
    synchronized(comandos) {
        comandos.addElement(comando);
        comandos.notify();
    }
}

private void pararThread() {
    pararThread = true;
    if (udpDC != null) {
        try {
            udpDC.close();
        } catch (IOException ioe) {
        }
    }
    synchronized(comandos) {
        comandos.notify();
    }
    if (executor != null) {
        try {
            executor.join();
        } catch (InterruptedException ie) {
            System.err.println("Excepção:" + ie.getMessage());
        }
    }
}

public void run() {
    String comando = null;

    Datagram pedido = null;
    Datagram resposta = null;

    byte dados[];

    /* criar a ligação por datagrama ao servidor */
    try {
        udpDC = (UDPDatagramConnection)
        Connector.open("datagram://10.10.255.195:1026");
        System.out.println("Tamanho Máximo: " +
            udpDC.getMaximumLength());
        System.out.println("Tamanho Nominal: " +
            udpDC.getNominalLength());
        System.out.println(udpDC.getLocalAddress() + ":" +
            udpDC.getLocalPort());

    } catch (IOException ioe) {
        Alert a = new Alert("Erro", ioe.getMessage(), null, null);
        a.setTimeout(Alert.FOREVER);
        Display.getDisplay(this).setCurrent(a, formEco);
        return;
    }

    while (!pararThread) {
        comando = null;
        synchronized(comandos) {
            if (comandos.size() > 0) {
                comando = (String)comandos.elementAt(0);
                comandos.removeElementAt(0);
            } else {
                try {
                    comandos.wait();
                } catch (InterruptedException ie) {
```

```
                    }
                }
            }
            /* enviar o datagrama e esperar resposta */
            if (comando != null) {
                try {
                    /* preparar o datagrama para enviar */
                    dados = comando.getBytes();
                    pedido = udpDC.newDatagram(dados, dados.length);
                    udpDC.send(pedido);
                    System.out.println(udpDC.getLocalAddress() + ":" +
                        udpDC.getLocalPort());
                    /* receber a resposta */
                    resposta = udpDC.newDatagram(udpDC.getMaximumLength());
                    udpDC.receive(resposta);

                    /* ler os dados da datagrama recebido */
                    long timeStamp = resposta.readLong();
                    String ecoResposta = resposta.readUTF();

                    Date data = new Date(timeStamp);

                    System.out.println(resposta.getAddress());
                    eco.setString(data.toString() + ": " + ecoResposta);
                } catch (IOException ioe) {
                    System.err.println("Excepção de IO: " +
                        ioe.getMessage());
                }
            }
        }
        try {

            if (udpDC != null) {
                udpDC.close();
            }
        } catch (IOException ioe) {
            System.err.println(ioe.getMessage());
        }
    }
}
```

Figura 9.5 MIDlet DatagramEco.

9.6. Ferramentas de Rede do WTK

O Wireless Toolkit vem com uma ferramenta que nos permite analisar o tráfego de rede utilizado pela nossa aplicação. Esta ferramenta – o monitor de rede (*Network Monitor*) – é bastante útil quando estamos à procura de erros na aplicação.

Para utilizarmos esta ferramenta temos de a activar nas preferências do WTK: *Edit->Preferences*. Esta opção abre a janela de preferências (ver Figura 9.6). No *tab Monitor* temos de marcar a caixa *Enable Network Monitoring*.

Quando executarmos o emulador, a janela do monitor de rede irá ser aberta e iremos poder inspeccionar todo o tráfego gerado e recebido pela aplicação. A janela possui vários *tabs* que correspondem aos diferentes tipos de protocolos de comunicação.

A Figura 9.7 mostra o resultado da execução da MIDlet `DatagramEco` quando se envia a mensagem "Ola" para o servidor.

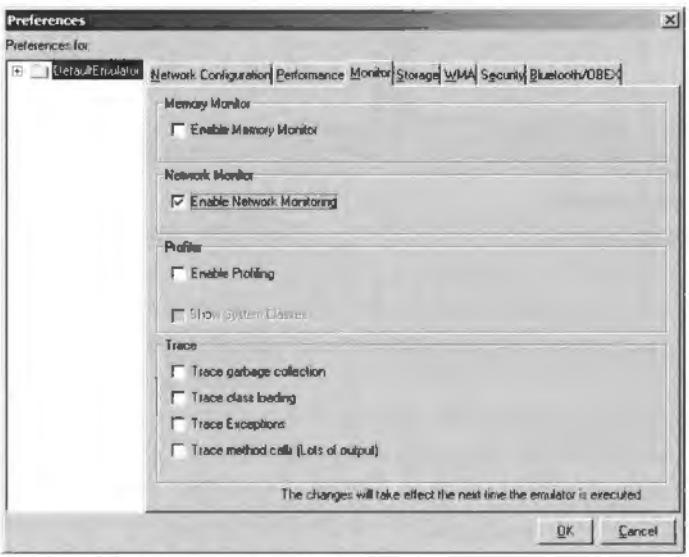
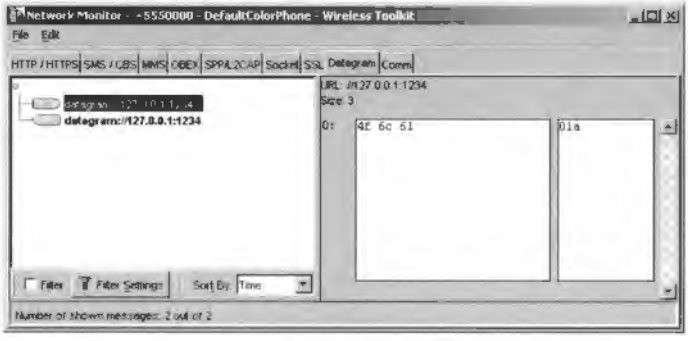
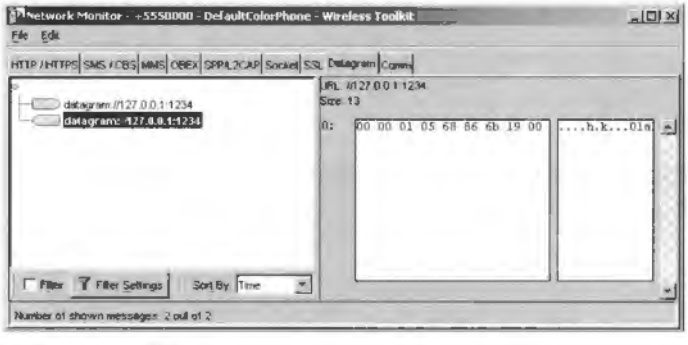


Figura 9.6 Janela de preferências do WTK.



(a) Pacote enviado



(b) Pacote recebido

Figura 9.7 Monitor de rede.

CAPÍTULO 10
Push Registry

Existem duas formas de publicar informação relativamente à forma como esta chega ao utilizador final: a publicação *pull* e a publicação *push*.

Na publicação *pull* (puxar), cabe ao utilizador iniciar o processo de transmissão da informação, pedindo à fonte a informação mais recente. Para se manter actualizado, o cliente terá de fazer pedidos periodicamente.

A publicação *push* (empurrar) funciona de forma inversa. Neste caso a fonte de informação avisa os vários clientes quando existirem actualizações. A recepção da informação é feita de forma assíncrona, *i.e.*, o cliente não sabe à partida quando a irá receber. Normalmente, este mecanismo implica que os clientes interessados se registem, de alguma forma, junto da fonte de informação. Só assim esta poderá saber quem deverá ser contactado quando houver nova informação disponível.

Este capítulo descreve o mecanismo *push* do MIDP que permite que as aplicações sejam iniciadas como resposta a eventos de rede.

10.1. O Mecanismo *Push* em MIDP

Já vimos, nas secções anteriores, como criar ligações servidor que nos permitem esperar por ligações do exterior. No entanto, para uma ligação do exterior ser atendida é necessário que a nossa MIDlet esteja a executar.

O mecanismo *push registry* do MIDP 2.0 permite que as ligações iniciadas no exterior sejam atendidas, mesmo que a MIDlet não esteja a ser executada naquele momento. O *push registry* permite lançar uma MIDlet em resposta a uma conexão. Na verdade, este mecanismo é mais genérico, permitindo lançar uma MIDlet em resposta a um evento de rede (ligação) ou em resposta a um evento temporal (permite-nos indicar que queremos lançar a MIDlet em determinada hora).

O *push registry* é, basicamente, uma tabela com três campos: o URL que identifica a ligação de servidor que a nossa MIDlet escuta; o nome da classe que representa a MIDlet que irá ser lançada quando uma ligação do exterior for efectuada; e um filtro, que permite definir uma gama de endereços remotos que pretendemos sejam capazes de lançar a MIDlet.

Um exemplo de uma tabela deste género seria:

URL de Ligação	Nome da classe MIDlet	Filtro
`socket://:1234`	`MIDletSocket`	`10.10.1?.*`
`datagram://:6789`	`MIDletDatagrama`	`*`

A primeira entrada da tabela indica que quando chegar uma ligação exterior do tipo *socket* para a porta 1234, a MIDlet `MIDletSocket` deverá ser lançada. No entanto, a MIDlet apenas é lançada se o endereço IP da origem estiver na gama `10.10.10.0` a `10.10.19.255` (o '?' pode ser substituído por qualquer carácter e o '*' pode ser substituído por qualquer texto). A sintaxe e a semântica do filtro dependem do tipo de endereçamento utilizado pelo protocolo da ligação. No caso de *sockets* ou datagramas são endereços IP, mas noutro tipo de ligações podem ter outro formato.

O facto de uma MIDlet ser lançada em resposta a uma conexão não significa que a ligação seja aceite automaticamente. A MIDlet é apenas lançada pelo AMS. O processo para aceitar a ligação tem de ser feito explicitamente, como se a aplicação não utilizasse o mecanismo *push registry* e pretendesse aceitar ligações exteriores.

Para uma MIDlet poder utilizar o mecanismo *push registry* é necessário que esteja registada no AMS do dispositivo. Existem duas formas de o fazer: dinamicamente, durante a execução da MIDlet; ou estaticamente, aquando da instalação, através de atributos no descritor da aplicação. Mais à frente vamos ver concretamente como fazer o registo.

10.2. A Classe `PushRegistry`

O *push registry* é representado pela classe `PushRegistry` que possui os seguintes métodos:

- `static String getFilter(String connection)`
  Retorna o filtro, *i.e.*, a gama de endereços de origem que são aceites, para a conexão "`connection`".
- `static String getMIDlet(String connection)`
  Retorna o nome da classe da MIDlet registada para a conexão "`connection`"
- `static String[] listConnections(boolean available)`
  Retorna a lista de conexões registadas para a MIDlet corrente. Se "`available`" for verdadeiro, então retorna apenas a lista de conexões com dados disponíveis.
- `static long registerAlarm(String midlet, long time)`
  Regista um instante em que a MIDlet "`midlet`" deverá ser lançada.
- `static void registerConnection(String connection, String midlet, String filter)`

Regista dinamicamente a conexão "`connection`", para lançar a MIDlet "`midlet`", se o endereço de origem estiver contido no filtro "`filter`".

- `static boolean unregisterConnection(String connection)`
  Remove um registo dinâmico. Retorna verdadeiro se a remoção foi bem-sucedida, ou falso se a conexão não estava registada ou se "`connection`" for nulo.

10.3. Activação por Conexão

O primeiro tipo de funcionalidade do *push registry* que vamos abordar é a activação de uma MIDlet em resposta a uma conexão vinda do exterior. Para isso precisamos primeiro de construir uma MIDlet, ainda sem pensar no *push registry*, que aceite conexões do exterior. Para tal, vamos criar uma MIDlet que responde a conexões do tipo datagrama, mostrando apenas o conteúdo do datagrama recebido:

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.*;
import java.io.*;
import javax.microedition.io.*;
import java.util.*;

public class DatagramServidor extends MIDlet implements Runnable,
    CommandListener {

    private Form formMsg;
    private TextField mensagem;
    private Alert dadosLigação;

    private Command cmdSair, cmdDadosLigação;

    private Thread executor = null;

    private boolean pararThread = false;

    UDPDatagramConnection udpDC = null;

    public DatagramServidor() {
       cmdSair = new Command("Sair", Command.EXIT, 1);
       cmdDadosLigação = new Command("Dados Ligação",
           Command.SCREEN, 1);

       formMsg = new Form("Mensagens");
       mensagem = new TextField("Mensagem", "", 256, TextField.ANY);
       formMsg.append(mensagem);
       formMsg.addCommand(cmdSair);
       formMsg.addCommand(cmdDadosLigação);
       formMsg.setCommandListener(this);

       dadosLigação = new Alert("Dados Conexão", "", null, null);
       executor = new Thread(this);
       executor.start();
    }

    public void startApp() {
       Display.getDisplay(this).setCurrent(dadosLigação, formMsg);
```

```
}

public void destroyApp(boolean unconditional) {
   pararThread();
}

public void pauseApp() {
}

public void commandAction(Command c, Displayable d) {
   if (c == cmdSair) {
       destroyApp(true);
       notifyDestroyed();
   } else if(c == cmdDadosLigação) {
       Display.getDisplay(this).setCurrent(dadosLigação, formMsg);
   }
}

private void pararThread() {
   pararThread = true;
   try {
       if (udpDC != null) {
       udpDC.close();
       }
   } catch (IOException ioe) {
       System.err.println(ioe.getMessage());
   }

   if (executor != null) {
       try {
           executor.join();
       } catch (InterruptedException ie) {
           System.err.println("Excepção:" + ie.getMessage());
       }
   }
}

public void run() {
   Datagram pedido = null;
   Datagram resposta = null;
   try {
       /* aceitar ligações ao servidor */
       udpDC = (UDPDatagramConnection)
           Connector.open("datagram://:1234");

       String s = udpDC.getLocalAddress() + ":" +
           udpDC.getLocalPort();

       dadosLigação.setString(s);
       dadosLigação.setTimeout(Alert.FOREVER);

   } catch (IOException ioe) {
       dadosLigação.setString(ioe.getMessage());
       Display.getDisplay(this).setCurrent(dadosLigação, formMsg);

       System.err.println(ioe.getMessage());
       return;
   }

   try {
       while (!pararThread) {
           pedido = udpDC.newDatagram(udpDC.getMaximumLength());
           udpDC.receive(pedido);
```

```
                String dados = new String(pedido.getData(), 0,
                    pedido.getLength());

                System.out.println(dados);
                mensagem.setString(dados);
            }
        } catch (IOException ioe) {
            System.err.println(ioe.getMessage());
        } finally {
            try {
                if (udpDC != null) {
                    udpDC.close();
                }
            } catch (IOException ioe) {}
        }
    }
}
```

Este programa é uma MIDlet normal que recebe ligações do tipo datagrama. Obviamente que, para tal, é preciso que a MIDlet esteja a ser executada.

Vamos ver agora como fazer com que a MIDlet seja iniciada automaticamente quando uma conexão for feita. Temos duas formas de o fazer: registando a MIDlet dinamicamente junto do AMS ou registando-a estaticamente.

10.3.1. Registo dinâmico

O registo dinâmico é feito invocando o método

```
PushRegistry.registerConnection(String connection,
    String midlet, String filter)
```

em que

connection deve ser a *string* utilizada no método `Connector.open()`,

midlet deve ser o nome da classe da nossa MIDlet e

filter o filtro que define quais as fontes aceites ("*", se quisermos aceitar ligações de qualquer fonte).

O registo apenas precisa de ser feito uma vez (da primeira vez que a MIDlet é lançada, por exemplo), pelo que podemos testar se o registo já foi feito antes de o efectuar. Para verificar se o registo já foi efectuado podemos utilizar o método `listConnections(false)`, que retorna todas as conexões registadas. Se o *array* retornado tiver tamanho zero, significa que o registo ainda não foi efectuado. Assim, o nosso código do exemplo anterior pode ser alterado para (apenas modificamos a parte que cria a ligação):

```
try {
    /* criar a ligação por datagrama ao servidor */
    udpDC = (UDPDatagramConnection)Connector.open("datagram://:1234");

    /* efectuar o registo se ainda não tiver sido feito */
    String [] conexões = PushRegistry.listConnections(false);
```

```
        if (conexões.length == 0) {
            PushRegistry.registerConnection("datagram://:1234",
                "DatagramServidor", "*");
        }
        String s = udpDC.getLocalAddress() + ":" + udpDC.getLocalPort();

        dadosLigação.setString(s);

        dadosLigação.setTimeout(Alert.FOREVER);
    } catch (IOException ioe) {

        dadosLigação.setString(ioe.getMessage());
        Display.getDisplay(this).setCurrent(dadosLigação, formMsg);

        System.err.println(ioe.getMessage());
        return;
    } catch (ClassNotFoundException cnfe) {
        System.err.println(cnfe.getMessage());
    }
```

Mais à frente vamos ver como testar o *push registry* usando o WTK, mas antes vejamos como efectuar o registo estático.

10.3.2. Registo estático

Para registarmos uma MIDlet estaticamente usamos atributos no descritor da aplicação:

`MIDlet-Push-<n>` Indica um registo *push*. Cada entrada deve obedecer ao formato: `<URL de Conexão>, <Nome da Classe>, <Filtro>`. O valor `<n>` deve começar em `1` e devem ser usados números consecutivos para cada uma das entradas.

No caso do nosso exemplo a entrada correspondente no ficheiro JAD seria: `MIDlet-Push-1: datagram://:1234, DatagramServidor, *`

Para definirmos os atributos *push* no WTK, basta clicar no botão [Settings] e escolher o *tab Push Registry* (ver Figura 10.1). Depois podemos adicionar os atributos clicando em [Add].

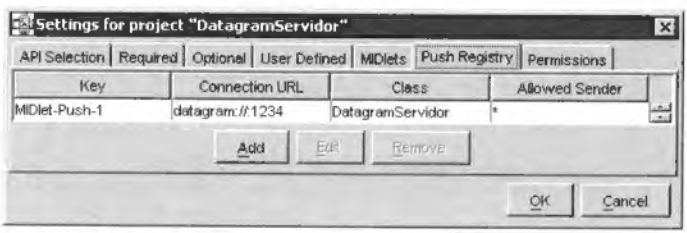


Figura 10.1 Janela de definição de atributos *push* no WTK.

Há que ter em atenção que apenas devemos usar os registos estáticos, i.e., os registos no ficheiro JAD, para as ligações sem as quais a nossa aplicação não pode mesmo funcionar. Se uma ligação *push* for opcional, devemos efectuar o registo dinamicamente. Isto porque podem existir conflitos nos registos *push* estáticos, por exemplo, se a mesma ligação estiver a ser utilizada por outra MIDlet Suite. Nestes casos a MIDlet não pode ser instalada. Por outro lado, se fizermos um registo dinâmico, mesmo que haja algum conflito, a aplicação pode detectar o conflito (apanhando a excepção durante o registo) e resolver a situação graciosamente.

10.3.3. Testar o *Push Registry* no WTK

Para testarmos o mecanismo *push* usando o WTK precisamos de efectuar alguns passos diferentes do habitual. Primeiro, temos de criar o pacote da aplicação, i.e., criar o ficheiro JAD e JAR. No WTK, podemos fazer isto acedendo ao menu *Project->Package->Create Package*. No final, iremos ter no directório `bin` do nosso projecto três ficheiros: o ficheiro de manifesto (que é colocado automaticamente também no JAR), o ficheiro JAD e o ficheiro JAR.

De seguida temos de executar a nossa aplicação via OTA no emulador. Esta forma de testar a aplicação simula os passos que o utilizador final teria de efectuar para instalar e executar a nossa MIDlet. Isto permite-nos instalar a nossa aplicação no emulador, tal como se a estivéssemos a instalar num telemóvel real. Para executar via OTA temos de seleccionar a opção *Project->Run via OTA*. O emulador será lançado e vamos ter acesso a uma opção para instalar a aplicação. Basta seguir os passos indicados (ver Figura 10.2) e responder "sim" às confirmações para permitir à aplicação usar o mecanismo *push* e aceder à rede. No final a nossa aplicação está instalada no emulador e podemos testar o *push registry* fazendo uma ligação UDP para o endereço do computador que está a correr o emulador e para a porta definida na nossa aplicação.

Podemos testar o funcionamento da nossa aplicação com duas janelas do emulador: mantendo o emulador OTA a funcionar e abrindo o projecto `DatagramEco` e executando [Run] (é necessário primeiro modificar o código do projecto `DatagramEco` de modo a que a ligação seja feita para o servidor do projecto que queremos testar – `datagram://127.0.0.1:1234`). Ao enviar o comando, o emulador OTA irá lançar a nossa MIDlet, depois de nos perguntar se pretendemos dar permissão para tal.

(a) (b) (c) (d)

(e) (f) (g)

Figura 10.2 Instalar uma aplicação no emulador via OTA.

10.4. Activação por Temporizador

O outro tipo de activação automática que podemos ter usando o *push registry* é a activação por temporizador. Este tipo de activação permite-nos definir uma hora em que queremos que a nossa MIDlet seja lançada.

Ao contrário do que acontece com a activação por conexão de rede, não é possível registar estaticamente as activações por temporizador, apenas dinamicamente.

O registo do temporizador, ou alarme, é feito através do método

```
static long registerAlarm(String midlet, long time)
```

em que

midlet é o nome da classe da MIDlet que queremos lançar e

time é a hora a que queremos que a MIDlet seja lançada, no formato devolvido por `Date.getTime()`.

O Exemplo 10.1 mostra uma MIDlet que permite ao utilizador definir a hora a que irá ser lançada da próxima vez. Reparem que o código que efectua o registo está colocado numa *thread* separada. Temos de fazer isto desta forma pelas mesmas razões que as ligações à rede devem ser feitas em *threads* separadas: para evitar possíveis *deadlocks*. A Figura 10.3 mostra o ecrã da aplicação.

Exemplo 10.1: `Temporizador` – Activação por temporizador

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.*;
import java.io.*;
import javax.microedition.io.*;
import java.util.*;

public class Temporizador extends MIDlet implements Runnable,
CommandListener {
    private Form formData;
    private DateField data;

    private Command cmdSair, cmdNovaExecução;
    public Temporizador() {
       cmdSair = new Command("Sair", Command.EXIT, 1);
       cmdNovaExecução = new Command("Definir", Command.SCREEN, 1);

       formData = new Form("Temporizador");
       data = new DateField("Próxima execução", DateField.DATE_TIME);
       data.setDate(new Date());

       formData.append(data);
       formData.addCommand(cmdSair);
       formData.addCommand(cmdNovaExecução);
       formData.setCommandListener(this);
    }

    public void startApp() {
       Display.getDisplay(this).setCurrent(formData);
    }

    public void destroyApp(boolean unconditional) {
    }

    public void pauseApp() {
    }

    public void commandAction(Command c, Displayable d) {
       if (c == cmdSair) {
          destroyApp(true);
          notifyDestroyed();
       } else if(c == cmdNovaExecução) {
          new Thread(this).start();
       }
    }

    public void run() {
```

```
            try {
                PushRegistry.registerAlarm("Temporizador",
                data.getDate().getTime());
            } catch (ClassNotFoundException cnfe) {
                System.err.println("Erro de registo: " +cnfe.getMessage());
            } catch (ConnectionNotFoundException cnfe) {
                System.err.println("Erro de registo: " +cnfe.getMessage());
            }
            System.out.println("Nova execução definida.");
        }
    }
```

Figura 10.3 `Temporizador` – Activação por temporizador.

CAPÍTULO 11
Áudio

Ao contrário do MIDP 1.0, a versão 2.0 deste perfil inclui algumas capacidades básicas de geração e reprodução de áudio. Este capítulo descreve a API de áudio do MIDP 2.0.

11.1. *Multimedia* API – MMAPI

A API de áudio do MIDP 2.0 é um subconjunto da API definido pelo pacote opcional MMAPI (API multimédia).

O MMAPI é um pacote opcional para MIDP 1.0 (e continua a ser para o MIDP 2.0) que permite dotar as MIDlets de capacidades multimédia. Com este pacote é possível gerar e reproduzir não apenas áudio, mas também vídeo, para além de permitir efectuar a captura de som e imagem.

Quando a versão 2.0 do MIDP foi desenvolvida, optou-se por incorporar algumas das funcionalidades da MMAPI. Foi incluído no MIDP 2.0 o chamado *Audio Building Block* (ABB) da MMAPI. Basicamente, é um subconjunto da API que permite reproduzir tons sonoros, notas MIDI e áudio amostrado.

As interfaces e classes que fazem parte da API de áudio do MIDP 2.0 são apresentadas na Figura 11.1.

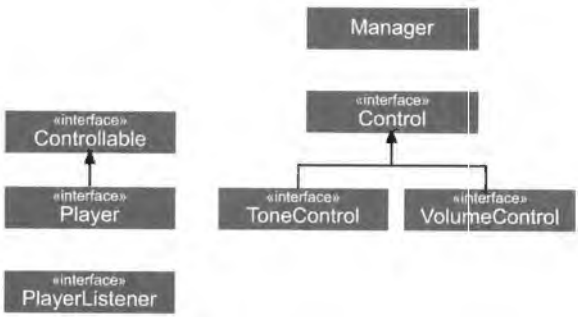


Figura 11.1 Diagrama de classes da API de áudio.

11.2. Tons

Uma das funções mais básicas da API é a geração de um tom. Um tom é caracterizado pela nota, duração e pelo volume. Podemos gerar um tom invocando o método `static void playTone(int note, int duration, int volume)`

Este método está definido na classe `javax.microedition.media.Manager` e possui os seguintes parâmetros:

note A nota MIDI a tocar.

duration A duração, em milissegundos, da nota.

volume O volume da nota: 0 é o silêncio e 100 o volume mais alto.

O Exemplo 11.1 mostra como tornar o telemóvel num instrumento musical usando este método. É um `Canvas` que recebe eventos do teclado e atribui uma nota a cada tecla, começando pela nota MIDI 60 (dó; na secção sobre sequências de tons, as notas MIDI são explicadas em mais detalhe). As notas tocam durante 250 milissegundos e no volume máximo.

Exemplo 11.1: CanvasTons – Geração de tons

```
import javax.microedition.lcdui.*;
import javax.microedition.media.*;

public class CanvasTons extends Canvas {

    public void paint(Graphics g) {
        g.setColor(255, 255, 255);
        g.fillRect(0, 0, getWidth(), getHeight());
        g.setColor(0, 0, 0);
        g.drawString("Pressione os botões", getWidth()/2, getHeight()/2,
            Graphics.HCENTER|Graphics.BASELINE);
    }

    public void keyPressed(int keyCode) {
        try {
            switch (keyCode) {
                case KEY_NUM1:
                    Manager.playTone(60, 250, 100);
                    break;
                case KEY_NUM2:
                    Manager.playTone(62, 250, 100);
                    break;
                case KEY_NUM3:
                    Manager.playTone(63, 250, 100);
                    break;
                case KEY_NUM4:
                    Manager.playTone(64, 250, 100);
                    break;
                case KEY_NUM5:
                    Manager.playTone(65, 250, 100);
                    break;
                case KEY_NUM6:
```

```
                    Manager.playTone(66, 250, 100);
                    break;
                case KEY_NUM7:
                    Manager.playTone(67, 250, 100);
                    break;
                case KEY_NUM8:
                    Manager.playTone(68, 250, 100);
                    break;
                case KEY_NUM9:
                    Manager.playTone(69, 250, 100);
                    break;
                case KEY_STAR:
                    Manager.playTone(70, 250, 100);
                    break;
                case KEY_NUM0:
                    Manager.playTone(71, 250, 100);
                    break;
                case KEY_POUND:
                    Manager.playTone(72, 250, 100);
                    break;
            }
        } catch (MediaException me) {
            System.err.println(me.getMessage());
        }
    }
}
```

11.3. Ficheiros de Áudio

A API de média do MIDP também permite a reprodução de ficheiros de áudio, embora isto não seja exigido pela especificação.

Para reproduzir um ficheiro de áudio é necessário criar um `Player`. À semelhança do que acontece com a criação de conexões de rede, não instanciamos um `Player` directamente. Em vez disso, usamos uma classe Fábrica – `Manager`.

Existem duas formas de obter um `Player`, que diferem na forma como o ficheiro de áudio é obtido:

- `static Player createPlayer(InputStream stream, String type)`
  Constrói um `Player` que lê o ficheiro de áudio de uma `InputStream`. O tipo de ficheiro – o *MIME Type* – é indicado por "`type`". Alguns *MIME Types* são:
  - Ficheiros Wave: `audio/x-wav`
  - Ficheiros AU: `audio/basic`
  - Ficheiros MP3: `audio/mpeg`
  - Ficheiros MIDI: `audio/midi`
  - Sequências de tons: `audio/x-tone-seq`
- `static Player createPlayer(String locator)`
  Constrói um `Player` a partir de um localizador de média (*media locator*).

Um localizador de média é especificado através da sintaxe utilizada nos URI: `<scheme>:<scheme-specific-part>`, em que `<scheme>` identifica o protocolo utilizado para transmitir o ficheiro. Um exemplo de um localizador de média seria: `http://jorgecardoso.org/musica/teste.wav`.

11.3.1. Ciclo de vida do `Player`

A alocação dos recursos necessários para um `Player` poder realizar a reprodução do áudio associado pode ser uma operação muito demorada. De forma a permitir ao programador ter um controlo mais minucioso sobre esta alocação de recursos, um `Player` possui um ciclo de vida composto por uma série de estados:

UNREALIZED Quando um `Player` é construído, está no estado UNREALIZED. Neste estado, o `Player` não possui informação suficiente para funcionar. Neste estado os métodos `getContentType()`, `setMediaTime()`, `getControls()` e `getControl()` não podem ser invocados no `Player` – uma excepção `IllegalStateException` será lançada se tal acontecer.

REALIZED Quando passa do estado UNREALIZED para o estado REALIZED, o `Player` inicia o processo de localização dos recursos necessários (como, por exemplo, comunicação com o servidor). A passagem para o estado REALIZED é feita invocando o método `realize()`. Isto permite ao programador iniciar na melhor altura um processo que pode ser moroso. Se, durante este processo, o método `deallocate()` for invocado, o processo é interrompido e o `Player` passará novamente para o estado UNREALIZED (se este método for invocado quando o `Player` está no estado UNREALIZED ou REALIZED o pedido é ignorado).

PREFETCHED Depois do estado REALIZED, o `Player` passa para o estado PREFETCHED. Esta passagem implica a execução de outras operações, que também podem ser morosas, como o enchimento de *buffers* ou outro processamento de inicialização. Esta operação reduz a latência de início do `Player` ao mínimo, isto é, uma vez no estado PREFETCHED, garante-se que a reprodução terá início o mais rapidamente possível quando se invocar `start()`. É possível voltar ao estado REALIZED, invocando `deallocate()` quando o `Player` está no estado PREFETCHED. Isto irá fazer com que o `Player` liberte os recursos adquiridos.

STARTED Ao invocar `start()`, o `Player` iniciará o mais rapidamente possível a reprodução do áudio. Quando o áudio chegar ao fim ou quando o método `stop()` for invocado, o `Player` passa novamente para o estado PREFETCHED. Enquanto o `Player` estiver neste estado, o método `setLoopCount()` não deverá ser chamado.

CLOSED Quando `close()` é invocado, o `Player` liberta quase todos os seus recursos e não pode ser usado novamente.

A Figura 11.2 mostra o diagrama de estados correspondente ao ciclo de vida de um `Player`.

Note-se que não é obrigatório invocar os métodos `realize()`, `prefetch()` e `start()` em sequência. Se invocarmos `prefetch()` sem antes termos invocado `realize()`, este será chamado implicitamente. Da mesma forma, podemos invocar `start()` sem invocar `realize()` ou `prefetch()`; estes serão invocados automaticamente.

O Exemplo 11.2 mostra como reproduzir ficheiros de áudio presentes no JAR da aplicação. Para isto, utilizamos o método

```
createPlayer(InputStream stream, String type)
```

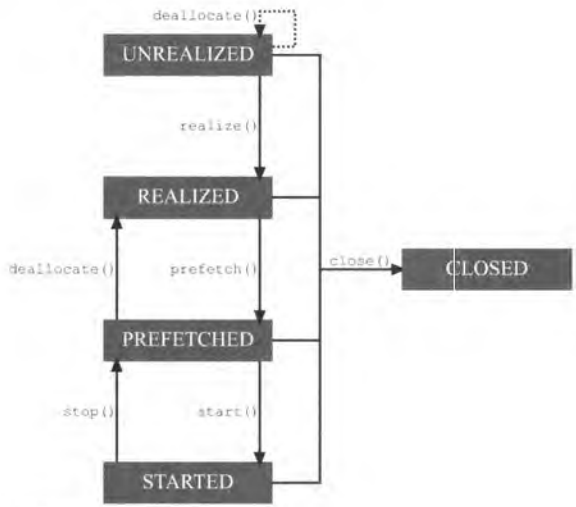


Figura 11.2 Ciclo de vida de um `Player`.

Este exemplo serve também para ilustrar mais alguns conceitos relacionados com os `Player`. Os dispositivos apenas suportam alguns *MIME Types* e alguns protocolos; seria impossível suportarem todos os existentes! Para descobrirmos quais os protocolos e *MIME Types* suportados podemos utilizar os seguintes dois métodos da classe `Manager`:

- `static String[] getSupportedContentTypes(String protocol)`
  Este método devolve a lista de tipos de conteúdo suportados para o protocolo especificado. Por exemplo, se "http" for passado, devolve a lista de *MIME Types* que podem ser reproduzidos através de HTTP. Se "`protocol`" for *null*, devolve todos os tipos de conteúdo suportados independentemente do protocolo.
- `static String[] getSupportedProtocols(String content_type)`
  Devolve a lista de protocolos suportados para o tipo de conteúdo dado. Se "`content_type`" for *null*, então devolve todos os protocolos suportados.

O exemplo mostra ainda como obter a duração total do áudio através do método `getDuration()` da classe `Player` e como colocar o `Player` numa determinada posição temporal (*media time*) através de `setMediaTime()` de modo que o áudio comece a tocar a meio do ficheiro (podemos também usar `getMediaTime()` para descobrir a posição actual do `Player`).

Exemplo 11.2: `AudioPlayer` – Reprodução de áudio

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.*;
import java.io.*;
import javax.microedition.media.*;

public class AudioPlayer extends MIDlet implements CommandListener {

    /* Os comandos da aplicação */
    private Command cmdSair;

    private String [][]ficheiros = {{"/bark.wav", "audio/x-wav"},
                                    {"/pattern.mid", "audio/midi"},
                                    {"/test-wav.wav", "audio/x-wav"}};
    List lista;

    public AudioPlayer() {
       cmdSair = new Command("Sair", Command.EXIT, 1);
       lista = new List("Ficheiros áudio", List.IMPLICIT);
       lista.append(ficheiros[0][0], null);
       lista.append(ficheiros[1][0], null);
       lista.append(ficheiros[2][0], null);
       lista.setCommandListener(this);
    }

    public void startApp() {
       Display.getDisplay(this).setCurrent(lista);

       /* Imprimir os protocolos e MIME Types suportados */
       String [] tiposConteúdo;
       String [] protocolos = Manager.getSupportedProtocols(null);
       for (int i = 0; i < protocolos.length; i++) {
           tiposConteúdo = Manager.getSupportedContentTypes(protocolos[i]);
```

```
                System.out.println("Protocolo: " +protocolos[i]);
                System.out.print("\tMIME Type: ");
                for (int j = 0; j < tiposConteúdo.length; j++) {
                    System.out.print(tiposConteúdo[j] +" ");
                }
                System.out.println();
            }
        }

        public void commandAction(Command c, Displayable d) {
            if (c == cmdSair) {
                destroyApp(true);
                notifyDestroyed();
            } else if (c == List.SELECT_COMMAND) {

                System.out.println(lista.getString(lista.getSelectedIndex()));
                String ficheiro = ficheiros[lista.getSelectedIndex()][0];
                String mime = ficheiros[lista.getSelectedIndex()][1];
                try {
                    Player player = Manager.createPlayer(getClass().
                    getResourceAsStream(ficheiro), mime);
                    player.realize();
                    player.setMediaTime(player.getDuration()/2);
                    System.out.println((player.getDuration()) +
                        " microssegundos.");
                    System.out.println((player.getMediaTime()) +
                        " microssegundos.");
                    player.prefetch();
                    player.start();
                } catch (IOException ioe) {
                    System.err.println(ioe.getMessage());
                } catch (MediaException me) {
                    System.err.println(me.getMessage());
                }
            }
        }

        public void destroyApp(boolean unconditional) {}

        public void pauseApp() {}
    }
```

11.3.2. Eventos

O estado do `Player` pode ser conhecido através de eventos. A aplicação pode registar um `PlayerListener` junto do `Player` de forma a ser informada sobre mudanças no seu estado ou outro tipo de informação.

O registo de um `PlayerListener` é feito através do método:

```
addPlayerListener(PlayerListener playerListener)
```

A interface `PlayerListener` define apenas um método:

```
void playerUpdate(Player player, String event,
    Object eventData)
```

em que:

player é o `Player` que gerou o evento;
event é o tipo de evento gerado; e
eventData é informação específica do tipo de evento.

Os tipos de eventos definidos pela especificação são os seguintes (as implementações são livres para adicionar outros tipos de eventos):

CLOSED Gerado quando o `Player` é fechado (`closed()`). Neste tipo de evento, "`eventData`" é *null*.

DEVICE_UNAVAILABLE Gerado quando o sistema tomou controlo sobre um dispositivo exclusivo que estava anteriormente disponível para o `Player`. Este evento só é gerado quando o `Player` está no estado REALIZED e será seguido de um evento do tipo `DEVICE_AVAILABLE` quando o dispositivo estiver novamente livre, ou ERROR se o dispositivo ficar indisponível permanentemente. Neste tipo de evento, "`eventData`" é uma `String` com o nome do dispositivo.

DEVICE_AVAILABLE Gerado quando um dispositivo detido pelo sistema fica novamente disponível para o `Player`. Quando este evento é gerado o `Player` está no estado REALIZED e a aplicação pode obter o dispositivo através do método `prefetch()` ou `start()`. Este evento é sempre precedido de um evento do tipo `DEVICE_UNAVAILABLE`. "`eventData`" é uma `String` que contém o nome do dispositivo.

DURATION_UPDATED Gerado quando a duração do `Player` é actualizada. Isto acontece com alguns tipos de média em que a duração não é conhecida *a priori*. Neste tipo de evento, "`eventData`" será um objecto `Long` com a duração.

END_OF_MEDIA Gerado quando o `Player` chega ao final do média. Neste evento, "`eventData`" é um `Long` com o instante em o média se encontrava quando chegou ao fim (em circunstâncias normais é igual à duração).

ERROR Gerado quando ocorre um erro. Nestes casos, "`eventData`" é uma `String` com a descrição do erro.

STARTED Gerado quando o `Player` é iniciado. Quando este evento é gerado, "`eventData`" é um `Long` com o instante em que o média se encontrava quando o `Player` foi iniciado.

STOPPED Gerado quando o `Player` pára em resposta à invocação de `stop()`. Neste evento, "`eventData`" contém um objecto do tipo `Long` com o instante em que o média se encontrava quando o `Player` parou.

VOLUME_CHANGED Gerado quando o volume é alterado. Quando este evento é recebido, "`eventData`" contém um objecto do tipo `VolumeControl` que pode ser usado para inquirir qual o novo volume.

O exemplo seguinte mostra um exemplo de utilização de eventos de áudio.

Exemplo 11.3: `AudioPlayerEventos` – Eventos de áudio

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.*;
import java.io.*;
import javax.microedition.media.*;

public class AudioPlayerEventos extends MIDlet implements CommandListener,
    PlayerListener {

    /* Os comandos da aplicação */
    private Command cmdSair, cmdComeçar, cmdParar;

    private String [][]ficheiros = {{"/bark.wav", "audio/x-wav"},
                                    {"/pattern.mid", "audio/midi"},
                                    {"/test-wav.wav", "audio/x-wav"}};
    List lista;

    Player player = null;
    public AudioPlayerEventos() {
       cmdSair = new Command("Sair", Command.EXIT, 2);
       cmdComeçar = new Command("Começar", Command.SCREEN, 1);
       cmdParar = new Command("Parar", Command.SCREEN, 1);

       lista = new List("Ficheiros áudio", List.IMPLICIT);
       lista.append(ficheiros[0][0], null);
       lista.append(ficheiros[1][0], null);
       lista.append(ficheiros[2][0], null);

       lista.addCommand(cmdParar);
       lista.addCommand(cmdSair);

       lista.setCommandListener(this);
    }

    public void startApp() {
       Display.getDisplay(this).setCurrent(lista);
    }

    public void commandAction(Command c, Displayable d) {
       if (c == cmdSair) {
           destroyApp(true);
           notifyDestroyed();
       } else if (c == List.SELECT_COMMAND) {

           System.out.println(lista.getString(lista.getSelectedIndex()));
           String ficheiro = ficheiros[lista.getSelectedIndex()][0];
           String mime = ficheiros[lista.getSelectedIndex()][1];
           try {
               /* fechar o player anterior */
               if (player != null) {
                  player.close();
               }
               player =
               Manager.createPlayer(getClass().
               getResourceAsStream(ficheiro), mime);
               player.addPlayerListener(this);
               player.start();
           } catch (IOException ioe) {
               System.err.println(ioe.getMessage());
           } catch (MediaException me) {
```

```
                System.err.println(me.getMessage());
            }
        } else if (c == cmdParar) {
            if (player != null) {
                try {
                    player.stop();
                } catch (MediaException me) {
                    System.err.println(me.getMessage());
                }
            }
        }
    }

    public void playerUpdate(Player player, String event,
    Object eventData) {
        System.out.println("Evento: " + event);

        if (event == PlayerListener.STARTED) {
            System.out.println("\tMedia Time: " +
                ((Long)eventData).toString());
        } else if (event == PlayerListener.END_OF_MEDIA) {
            System.out.println("\tMedia Time: " +
                ((Long)eventData).toString());
        } else if (event == PlayerListener.STOPPED) {
            System.out.println("\tMedia Time: " +
                ((Long)eventData).toString());
        } else if (event == PlayerListener.CLOSED) {
        }
    }

    public void destroyApp(boolean unconditional) {}

    public void pauseApp() {}
}
```

11.3.3. Controlos

De forma a manter a API o mais genérica possível e, ao mesmo tempo, permitir extensibilidade, a classe `Player` possui poucos métodos para controlar a reprodução do som. Os controlos específicos a determinado `Player` podem ser acedidos através das classes que implementam a interface *`Control`*, que podem ser obtidas através do método `getControl()`.

Por exemplo, para obter um controlo para alterar o volume do som, faríamos:

```
player = Manager.createPlayer(getClass().getResourceAsStream(ficheiro),
    mime);
VolumeControl vc = player.getControl("VolumeControl");
vc.setLevel(50);
```

A API apenas define dois controlos – `VolumeControl` e `ToneControl` (este último é descrito na secção seguinte) – mas as implementações são livres para acrescentar controlos apropriados aos tipos de conteúdos e protocolos suportados.

Para obter um controlo usamos o método `getControl(String controlType)` da classe `Player`. O parâmetro "`controlType`" é o nome da classe que implementa o

controlo. Deve ser usado o nome completo da classe, i.e., nome do pacote mais nome da classe. Se não for usado o nome do pacote, `javax.microedition.media.control` é assumido.

Se quisermos saber quais os controlos suportados por determinado `Player` podemos utilizar o método `String[] getControls()` que devolve a lista de controlos suportados. Se nenhum controlo for suportado a lista terá tamanho zero.

11.4. Sequências de Tons

A última coisa a saber sobre a API de áudio do MIDP 2.0 é a construção de sequências de tons. Já vimos anteriormente como gerar tons simples, mas se quisermos construir "músicas" mais complexas existe uma forma mais flexível.

A reprodução de sequências de tons é feita recorrendo a um `Player` especial, que se obtém através do localizador `Manager.TONE_DEVICE_LOCATOR` e a um controlo – o `ToneControl`.

O código-base para reproduzir uma sequência de tons é o seguinte:

```
Player player = Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR);
player.realize();
ToneControl tc = (ToneControl)player.getControl("ToneControl");
tc.setSequence(música);
player.start();
```

O controlo `ToneControl` possui apenas um método:

`setSequence(byte sequence[])`

Este método aceita um *array* de bytes que constitui a sequência de tons a ser reproduzida. A construção da sequência obedece a uma gramática definida na documentação da classe, mas, basicamente, resume-se ao seguinte:

```
private byte [] música = {
    ToneControl.VERSION, 1, // versão (apenas existe a versão 1)
    ToneControl.TEMPO, 40, // opcional - definição do ritmo
    ToneControl.RESOLUTION, 64 // opcional - definição da resolução
    [pares nota-duração] // a sequência propriamente dita
};
```

A sequência começa com a definição da versão do formato da mesma. Neste momento está apenas definida a versão 1, pelo que os dois primeiros bytes são sempre iguais ao exemplo dado. Nos bytes seguintes podemos alterar as definições por omissão do tempo (120 pm) e da resolução (1/64). Seguem-se pares nota-duração, que constituem a sequência de tons propriamente dita.

O tempo é especificado como 1/4 do tempo efectivo, *i.e.*, se pretendermos um tempo de 160 bpm devemos especificar um valor de 40 (isto porque como todos os valores são bytes o valor máximo é 127, o que seria um valor máximo muito baixo para o tempo).

A resolução é especificada como o inverso do valor pretendido, ou seja, se quisermos uma resolução de 1/64, devemos passar o valor 64.

Os valores das notas são os definidos na especificação MIDI (o valor 60 corresponde à nota dó (quarta oitava), ao que se seguem as notas dó sustenido (61), ré (62), etc.). A duração da nota é especificada como um múltiplo da resolução. Por omissão, a resolução é 1/64 de um tempo 4/4. Ou seja, uma nota tem sempre a duração de 4 batidas e a resolução indica em quantas partes podemos dividir uma nota. A tabela seguinte indica a duração para algumas durações típicas de notas. A tabela mostra a duração para duas resoluções diferentes (64 e 96).

Tamanho da Nota	Duração	
	Resolução = 64	Resolução = 96
semibreve	64	96
mínima	32	48
semínima	16	24
semínima com ponto de aumentação	24	36
colcheia	8	12
tercina	-	8
semicolcheia	4	6
fusa	2	3
semifusa	1	-

Reparem que algumas durações não podem ser expressas com uma determinada resolução por não serem submúltiplas dessa resolução.

A duração efectiva de uma nota, em milissegundos, é dada pela seguinte fórmula:

$$duração * 60 * 1000 * 4/(resolução * tempo) \qquad (11.1)$$

Para um tempo de 60 bpm e uma resolução de 64, uma nota com duração 16 terá uma duração efectiva de 16 * 60 * 1000 * 4/(64 * 60), ou seja, 1000 ms.

O silêncio é uma nota especial que tem o valor de `ToneControl.SILENCE`.

Podemos alterar o volume do som em qualquer altura da sequência através da instrução `ToneControl.SET_VOLUME`. Por exemplo:

```
ToneControl.SET_VOLUME, 80
```

Coloca o volume a 80% do seu valor máximo (o valor pode apenas variar entre 0 e 100).

Se quisermos repetir uma nota várias vezes podemos utilizar o código especial `ToneControl.REPEAT`, seguido do número de vezes que a nota deve ser repetida, seguido da nota e da duração.

Podemos também repetir blocos inteiros de notas. Para isso é preciso primeiro definir o bloco:

```
private byte [] música = {
 ToneControl.VERSION,          1,  // versão 1
 ToneControl.TEMPO,            40, // 160 bpm
 ToneControl.RESOLUTION,       resolução,
 ToneControl.BLOCK_START,      0,  // definição do bloco número zero
 60,          8,
 [...]
 ToneControl.BLOCK_END,        0,
 [... definição de outros blocos]
 61,          16,     // Início da sequência
 ToneControl.PLAY_BLOCK,       0   // Reproduz o bloco zero
}
```

Os blocos têm de ser definidos depois das definições de `VERSION`, `TEMPO` e `RESOLUTION` e têm de ser todos definidos em sequência. Os blocos são identificados por um número (0 a 127).

O exemplo seguinte mostra um exemplo da utilização de sequências de tons. Para construir a sequência, defini primeiro algumas notas musicais mapeando o seu valor MIDI de forma que a sequência final possa ser lida mais facilmente.

Exemplo 11.4: `SequenciaTons` – Sequências de tons

```
import java.io.*;
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.*;
import javax.microedition.media.*;
import javax.microedition.media.control.*;

public class SequenciaTons extends MIDlet implements CommandListener {
    /* Os comandos da aplicação */
    private Command cmdSair, cmdTocar;

    private TextBox ecrã;

    /* as notas musicais de acordo com a especificação MIDI
     (quarta oitava) */
    byte DO             = 60;
    byte DOsus          = 61;
    byte RÉ             = 62;
    byte RÉsus          = 63;
    byte MI             = 64;
    byte FÁ             = 65;
    byte FÁsus          = 66;
    byte SOL            = 67;
    byte SOLsus         = 68;
    byte LÁ             = 69;
```

```
byte LÁsus          = 70;
byte SI             = 71;
byte DO5            = 72; // DO da quinta oitava

/* as durações das notas (apenas precisamos de uma resolução de 32)
 */
byte resolução = 32;
byte semicolcheia = (byte)(resolução/16);
byte colcheia = (byte)(resolução/8);
byte seminima = (byte)(resolução/4);
byte minima = (byte)(resolução/2);
byte semibreve = resolução;

/* melodia da "Pantera Cor de Rosa" */
private byte [] música = {
   ToneControl.VERSION,                1, // versão 1
   ToneControl.TEMPO,                  40, // 160 bpm
   ToneControl.RESOLUTION,     resolução,
   ToneControl.BLOCK_START,    0,
   DOsus,                              colcheia,
   MI,                                 colcheia,
   ToneControl.BLOCK_END,              0,
   ToneControl.PLAY_BLOCK,     0,
   ToneControl.SILENCE,                minima,
   FÁsus,                              colcheia,
   SOL,                                      colcheia,
   ToneControl.SILENCE,                minima,
   ToneControl.PLAY_BLOCK,     0,
   ToneControl.SILENCE,                colcheia,
   FÁsus,                              colcheia,
   SOL,                                      colcheia,
   ToneControl.SILENCE,                colcheia,
   DO5,                                      colcheia,
   SI,                                 colcheia,
   ToneControl.SILENCE,                colcheia,
   ToneControl.PLAY_BLOCK,     0,
   ToneControl.SILENCE,                colcheia,
   SI,                                 colcheia,
   LÁsus,                              semibreve,
   ToneControl.SILENCE,                minima,
   LÁ,                                 semicolcheia,
   SOL,                                      semicolcheia,
   MI,                                 semicolcheia,
   RE,                                 semicolcheia,
   MI,                                 minima
};

public SequenciaTons() {
   cmdSair = new Command("Sair", Command.EXIT, 1);
   cmdTocar = new Command("Tocar", Command.SCREEN, 1);

   ecrã = new TextBox("Sequência de Tons", "Pantera Cor de Rosa",
   255, TextField.ANY);

   ecrã.addCommand(cmdSair);
   ecrã.addCommand(cmdTocar);
   ecrã.setCommandListener(this);
}

public void startApp() {
   Display.getDisplay(this).setCurrent(ecrã);
}

public void commandAction(Command c, Displayable d) {
```

```
        if (c == cmdSair) {
            destroyApp(true);
            notifyDestroyed();
        } else if (c == cmdTocar) {

            try {
                Player player =
                Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR);
                player.realize();
                ToneControl tc =
                    (ToneControl)player.getControl("ToneControl");
                tc.setSequence(música);
                player.start();
            } catch (IOException ioe) {
                System.err.println(ioe.getMessage());
            } catch (MediaException me) {
                System.err.println(me.getMessage());
            }
        }
    }

    public void destroyApp(boolean unconditional) {}

    public void pauseApp() {}
}
```

CAPÍTULO 12

Jogos – API e Técnicas Básicas

A maioria das aplicações distribuídas para telemóveis são jogos. A programação de jogos, nomeadamente os que recorrem a gráficos e animações, possui necessidades muito próprias tais como *scroll* de cenários, utilização de *sprites*, cenários compostos por camadas, etc. Atendendo a isto, a especificação MIDP 2.0 introduziu na sua API um pacote de classes destinadas a facilitar o trabalho do programador de jogos – o pacote `javax.microedition.lcdui.game`. Este capítulo descreve essa API.

12.1. A API

A API de jogos do MIDP 2.0 é composta pelas seguintes classes:

`GameCanvas` Esta classe é a base da interface com o utilizador do jogo. Fornece mecanismos para obter o *input* do utilizador através de *polling* e um duplo *buffer* para os gráficos, para além das funcionalidades herdadas da classe `Canvas`.

`Layer` A classe `Layer` é uma classe abstracta que representa um elemento visual no jogo. Fornece mecanismos para alterar a posição do elemento no ecrã e para o tornar visível/invisível.

`TiledLayer` Um tipo de *layer* que é construída através de pequenas imagens organizadas numa grelha, como se de azulejos se tratassem.

`Sprite` Uma *layer* que representa um elemento (*sprite*), normalmente animado, do jogo. É possível testar a colisão de *sprites* com os outros elementos visuais do jogo.

`LayerManager` Esta classe simplifica a organização das várias *layers* de um jogo. Permite definir uma ordem pela qual as *layers* são desenhadas, definir uma janela de visualização, inserir e apagar *layers*, etc.

A relação entre estas classes é apresentada no diagrama de classes da Figura 12.1

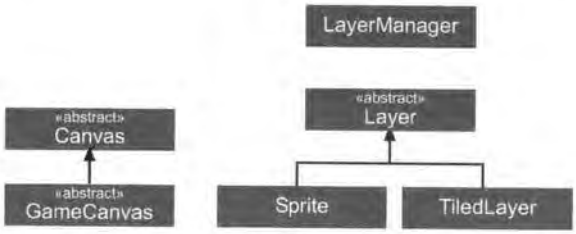


Figura 12.1 Diagrama de classes da API de jogos.

12.2. GameCanvas

A classe `GameCanvas` é a base da interface com o utilizador na API de jogos do MIDP 2.0.

Esta classe altera ligeiramente a forma como a interface é programada, relativamente ao que fazíamos no caso do `Canvas`. As principais diferenças são:

- Não implementamos o método `paint()`. Para desenharmos no ecrã temos de invocar o método `getGraphics()` para obter o objecto `Graphics` e só depois desenhamos. Para o que desenhámos seja afixado no ecrã é necessário invocar `flushGraphics()`, uma vez que o `Graphics` que obtemos corresponde a um duplo *buffer* de desenho.
- Os eventos de teclas podem ser obtidos através de *polling*. Em vez de implementarmos os métodos `keyPressed()`, `keyReleased()` e `keyRepeated()`, o `GameCanvas` dá-nos outra forma de determinar se uma tecla foi premida ou não: através de `getKeyStates()`, que devolve um valor inteiro cujos bits representam o estado dos vários botões do teclado do telemóvel.

O ciclo típico de um jogo que utiliza o `GameCanvas` é o seguinte:

```
Graphics g = getGraphics();
while (aJogar) {
    [determinar as teclas pressionadas]
    [actualizar estado do jogo]
    [desenhar o novo estado do jogo]
    flushGraphics();
    esperar alguns milissegundos]
}
```

12.2.1. Estado das teclas

O GameCanvas fornece ao programador a possibilidade de obter o estado das teclas de forma mais controlada do que através do mecanismo de eventos do Canvas.

Em vez de esperarmos que o evento seja lançado, podemos determinar imediatamente o estado das teclas através de `int getKeyStates()`. Com este mecanismo podemos determinar o estado das teclas correspondentes às acções de jogos. O método `getKeyStates()` devolve um inteiro cujos bits correspondem ao estado de cada uma das teclas. Se o bit correspondente a uma determinada tecla for 1, isso significa que a tecla foi pressionada. Para determinarmos se a tecla correspondente à acção LEFT foi pressionada faríamos:

```
int keyStates = getKeyStates();
if ((keyStates & LEFT_PRESSED) != 0) {
  // tecla esquerda pressionada
}
```

A constante `LEFT_PRESSED` é uma das constantes definidas pelo `GameCanvas` para cada tecla. A lista completa é a seguinte:

- `DOWN_PRESSED`
- `LEFT_PRESSED`
- `RIGHT_PRESSED`
- `UP_PRESSED`
- `FIRE_PRESSED`
- `GAME_A_PRESSED`
- `GAME_B_PRESSED`
- `GAME_C_PRESSED`
- `GAME_D_PRESSED`

Este mecanismo funciona mesmo que o utilizador pressione e largue a tecla antes do método ser chamado. Quando uma tecla é pressionada, o bit correspondente é colocado a 1 e apenas passa a 0 quando o método for invocado, *i.e.*, o acto de largar a tecla não afecta o estado retornado por `getKeyStates()`.

O Exemplo 12.1 ilustra uma utilização típica do `GameCanvas`. A classe `MeuGameCanvas` estende a classe `GameCanvas` e implementa `Runnable` – o "motor" do jogo é implementado numa *thread* diferente. No construtor da classe é invocado o construtor do `GameCanvas`:

```
GameCanvas(boolean suppressKeyEvents)
```

O parâmetro "suppressKeyEvents" permite-nos indicar se pretendemos que o mecanismo de eventos de teclas do Canvas permaneça activo ou não. Se não necessitarmos deste mecanismo devemos passar o valor falso neste parâmetro para melhorar o desempenho da aplicação, uma vez que, assim, algumas chamadas de sistema não precisam de ser feitas.

O exemplo permite que o utilizador mova um quadrado preto no ecrã, pressionando as teclas direccionais (LEFT, RIGHT, UP e DOWN).

Exemplo 12.1: MeuGameCanvas – Uso do GameCanvas

```
import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

public class MeuGameCanvas extends GameCanvas implements Runnable {

    private Thread motor;

    private boolean ligado = true;

    private int x = 0, y = 0;

    public MeuGameCanvas() {
        super(false);
        motor = new Thread(this);
        motor.start();
    }

    public void run() {
        Graphics g = getGraphics();
        while (ligado) {

            /* determinar a tecla pressionada */
            int keyStates = getKeyStates();

            if ((keyStates & RIGHT_PRESSED) != 0) {
                x++;
            } else if ((keyStates & LEFT_PRESSED) != 0) {
                x--;
            } else if ((keyStates & UP_PRESSED) != 0) {
                y--;
            } else if ((keyStates & DOWN_PRESSED) != 0) {
                y++;
            }

            /* limpar o fundo (pintar a branco) */
            g.setColor(255, 255, 255);
            g.fillRect(0, 0, getWidth(), getHeight());

            /* desenhar o quadrado na nova posição */
            g.setColor(0, 0, 0);
            g.fillRect(x, y, 40, 40);

            /* passar o buffer para o ecrã */
            flushGraphics();
            try {
                Thread.sleep(100);
            } catch(InterruptedException ie) {
                System.err.println(ie.getMessage());
```

```
            }
        }
    }

    public void parar() {
        ligado = false;
        try {
            motor.join();
        } catch (InterruptedException ie) {
            System.err.println(ie.getMessage());
        }
    }
}
```

12.3. *Layers*

As *layers* são, tal como o nome indica, elementos visuais que podem ser organizados em camadas, de forma que uns são desenhados por cima dos outros.

De uma forma genérica, uma `Layer` fornece os seguintes métodos:

paint(Graphics g) Desenha esta *layer*, caso esteja visível. A *layer* é desenhada na sua posição (x, y) corrente, relativamente à origem do objecto `Graphics`.

setPosition(int x, int y) Define uma nova posição para a *layer*, de modo que o canto superior esquerdo esteja localizado na posição (x, y) relativamente à origem do sistema de coordenadas do `Graphics`.

move(int dx, int dy) Método auxiliar para mover mais facilmente uma *layer*. Com este método podemos mover a *layer* relativamente à sua posição actual.

setVisible(boolean visible) Define a visibilidade da *layer*. Se "`visible`" for *true* então a *layer* será visível, caso contrário será invisível (não é desenhada).

boolean isVisible() Retorna a visibilidade da *layer*.

int getX() Retorna a posição horizontal da *layer*.

int getY() Retorna a posição vertical da *layer*.

int getHeight() Devolve a altura da *layer*.

int getWidth() Devolve a largura da *layer*.

A API MIDP 2.0 define duas classes que concretizam a classe `Layer`, mas antes de vermos esses exemplos concretos deixem-me mostrar como se utilizariam as *layers*, de forma genérica:

```
Layer personagem;
Layer fundo;
Graphics g = getGraphics();
while (aJogar) {
    [determinar as teclas pressionadas]
    /*actualizar estado do jogo*/
    personagem.move(x, y);
```

```
        /*desenhar o novo estado do jogo*/
        fundo.paint(g);
        personagem.paint(g);
        flushGraphics();
        [esperar alguns milissegundos]
    }
```

A ordem em que se invoca o método `paint()` é importante; as *layers* mais afastadas devem ser desenhadas primeiro, uma vez que são desenhadas umas por cima das outras.

12.3.1. *Sprites*

As *sprites* são um tipo de *layer* tipicamente utilizado para objectos visuais animados. Exemplos de *sprites* são as personagens dos jogos, um tiro de canhão, etc.

As *sprites* são construídas a partir de uma imagem que contém as várias *frames* usadas para a animação. A classe `Sprite` possui três construtores:

- `Sprite(Image image)`
  Cria uma *sprite* não animada, *i.e.*, uma *sprite* com apenas uma *frame*.
- `Sprite(Image image, int frameWidth, int frameHeight)`
  Cria uma *sprite* animada usando as *frames* contidas na imagem "`image`". A largura e altura de cada *frame* são dadas por "`frameWidth`" e "`frameHeight`".
- `Sprite(Sprite s)`
  Cria uma *sprite* a partir de outra. Este construtor serve para facilitar o trabalho do programador, permitindo duplicar rapidamente uma *sprite*.

As *frames* dentro de uma imagem podem ser organizadas de várias formas. A Figura 12.2 mostra algumas das alternativas. Cabe ao programador decidir qual a melhor forma de as organizar. As *frames* serão sempre numeradas da esquerda para a direita e de cima para baixo.

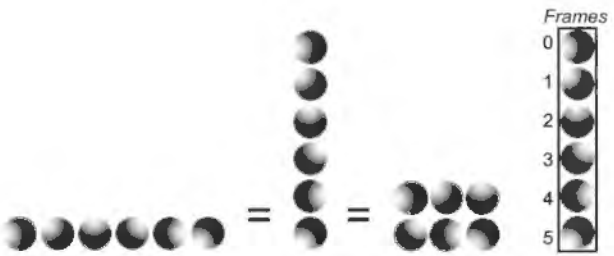


Figura 12.2 Organização das *frames* numa imagem.

A sequência geral de passos para usar uma *sprite* é a seguinte:

```
Sprite personagem = new Sprite(imagem, 20, 30); // criar a sprite
Graphics g = getGraphics();
while (aJogar) {
    [determinar as teclas pressionadas]
    /* actualizar estado do jogo*/

    personagem.move(x, y); // mover a personagem

    personagem.nextFrame(); // animar a personagem

    /*desenhar o novo estado do jogo*/
    personagem.paint(g);
    flushGraphics();
    [esperar alguns milissegundos]
}
```

O método `nextFrame()` da classe `Sprite` faz avançar a *frame* corrente da *sprite*. Desta forma as *frames* são desenhadas consecutivamente, dando a aparência de movimento. A sequência de *frames* é considerada circular, do ponto de vista deste método. Ou seja, invocar `nextFrame()` quando a *frame* corrente é a última volta à primeira. Também é possível passar à *frame* anterior através do método `prevFrame()`.

A sequência de *frames* que compõem a *sprite* pode ser definida com o método `setFrameSequence(int [] sequence)`. Neste método, "`sequence`" é um *array* com os índices das *frames* da imagem, sendo que as *frames* são numeradas da esquerda para a direita e de cima para baixo. A sequência de *frames* da *sprite* pode, obviamente, conter *frames* em qualquer ordem, inclusivamente repetidas. A sequência definida por omissão corresponde à ordem das *frames* presentes na imagem, i.e., no caso da imagem da Figura 12.2, a sequência seria {0, 1, 2, 3, 4, 5}. As únicas restrições para a sequência de *frames* de uma *sprite* são a necessidade de conter pelo menos um elemento e a necessidade de cada elemento ser um índice de frame válido, i.e., entre 0 e o número de frames da imagem menos 1.

É possível avançar directamente para uma determinada *frame* com o método `setFrame(int sequenceIndex)`, em que "`sequenceIndex`" é um índice da sequência de *frames* e não o número de uma *frame* da imagem original. Inversamente, é possível determinar a *frame* actual, através de `int getFrame()`.

Para mostrar algumas funcionalidades básicas das *sprites* vamos observar o exemplo seguinte. Este exemplo é um `GameCanvas` que implementa um jogo em que o utilizador controla a posição, no eixo horizontal, de um balde no fundo do ecrã. Existe uma esfera que "cai" desde o topo do ecrã até ao fundo. O objectivo é apanhar a esfera com o balde. A esfera é uma *sprite* animada com 6 *frames* enquanto que o balde é uma *sprite* estática. Neste exemplo, ainda não é possível apanhar a esfera – iremos ver isso na secção seguinte. A Figura 12.3 mostra o ecrã tal como aparece no emulador do WTK.

Exemplo 12.2: **SpriteBasicaCanvas** – Uso de **Sprite**

```
import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;
import java.io.*;
import java.util.*;

public class SpriteBasicaCanvas extends GameCanvas implements Runnable {

    /* Número de posições que o balde pode ter no ecrã */
    private static final int NUM_POSIÇÕES = 20;

    /* Velocidade a que a bola se desloca */
    private static final int VELOCIDADE = 8;

    /* A velocidade a que o balde se desloca de cada vez que
    o utilizador pressiona uma tecla
    */
    private int velocidadeBalde;

    /* A posição do balde */
    private int baldeX = 0;

    /* As sprites para o balde e para a bola */
    private Sprite balde;
    private Sprite bola;

    Random random = new Random();
    /* A thread do motor do jogo */
    private Thread motor;
    /* A thread executa enquanto ligado == true */
    private boolean ligado = true;

    public SpriteBasicaCanvas() {
        super(false);

        /* criar as sprites */
        try {
            Image baldeImg = Image.createImage("/balde.png");
            balde = new Sprite(baldeImg, 40, 44);

            Image bolaImg = Image.createImage("/drops.png");
            bola = new Sprite(bolaImg, 20, 20);
        } catch (IOException ioe) {
            System.err.println(ioe.getMessage());
        }

        /* calcular a velocidade do balde */
        velocidadeBalde = getWidth()/NUM_POSIÇÕES;

        /* iniciar o motor do jogo */
        motor = new Thread(this);
        motor.start();
    }

    public void run() {
        int largura = getWidth();

        Graphics g = getGraphics();

        /* inicializar a posição da primeira bola */
        bola.setPosition(aleatório(largura), 0);
```

```
        while (ligado) {

            /* determinar a tecla pressionada */
            int keyStates = getKeyStates();

            if ((keyStates & RIGHT_PRESSED) != 0) {
                baldeX += velocidadeBalde;
            } else if ((keyStates & LEFT_PRESSED) != 0) {
                baldeX -= velocidadeBalde;
            }

            /* limpar o fundo (pintar a branco) */
            g.setColor(255, 255, 255);
            g.fillRect(0, 0, getWidth(), getHeight());

            /* actualizar posição do balde e desenhá-lo */
            balde.setPosition(baldeX, getHeight()-44);
            balde.paint(g);

            /* actualizar a bola */
            bola.move(0, VELOCIDADE);
            bola.nextFrame();
            bola.paint(g);

            /* se a bola chegou ao fundo do ecrã volta a colocá-la no topo */
            if (bola.getY() > getHeight()) {
                bola.setPosition(aleatório(largura), 0);
            }

            flushGraphics();

            try {
                Thread.sleep(100);
            } catch(InterruptedException ie) {
                System.err.println(ie.getMessage());
            }
        }
    }

    private int aleatório(int máximo) {
        int r = Math.abs(random.nextInt()) % máximo;
        return r;
    }

    public void parar() {
        ligado = false;
        try {
            motor.join();
        } catch (InterruptedException ie) {
            System.err.println(ie.getMessage());
        }
    }
}
```

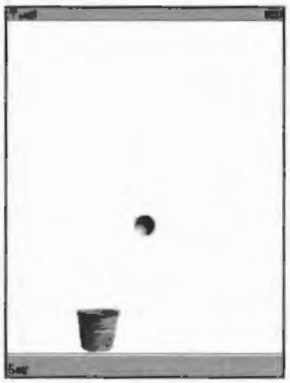


Figura 12.3 `SpriteBasicaCanvas` – Utilização de *sprites*.

Colisões

Uma operação muito importante quando se desenham jogos é a determinação de colisões entre os vários elementos visuais do jogo. A API de jogos do MIDP 2.0 fornece-nos dois mecanismos para detectar a colisão de *sprites* com outras *sprites*, imagens, ou *tiled layers* (descritas mais à frente):

- Detecção de colisão ao nível do píxel. Neste tipo de colisões, a detecção é feita comparando a posição dos píxeis opacos da *sprite* com a posição dos píxeis opacos do outro elemento.
- Detecção de colisão ao nível do rectângulo. Neste tipo de colisão, apenas se verifica a intersecção dos rectângulos de colisão da *sprite* com o rectângulo que define a fronteira do outro elemento.

Os métodos fornecidos para detecção de colisões fazem parte da classe `Sprite` e descrevem-se a seguir. Em todos estes métodos o parâmetro "`pixelLevel`" indica se pretendemos uma detecção ao nível do píxel ("`pixelLevel`" igual a *true*) ou apenas ao nível do rectângulo ("`pixelLevel`" igual a *false*):

- `boolean collidesWith(Sprite s, boolean pixelLevel)`
  Verifica se há uma colisão entre esta `Sprite` e a `Sprite` indicada.
- `boolean collidesWith(Image image, int x, int y, boolean pixelLevel)`
  Verifica se há uma colisão entre esta `Sprite` e a `Image` indicada. Os parâmetros "`x`" e "`y`" indicam a posição do canto superior esquerdo da imagem.

- `boolean collidesWith(TiledLayer t, boolean pixelLevel)`
  Verifica se há uma colisão entre esta `Sprite` e a `TiledLayer` indicada.

Em todos estes métodos está subjacente o conceito de rectângulo de colisão. O rectângulo de colisão é um rectângulo imaginário, definido sobre uma *sprite*, utilizado da seguinte forma na detecção de colisões:

- Na detecção ao nível do píxel, este rectângulo delimita os píxeis da *sprite* que são testados. Apenas os píxeis dentro da área delimitada pelo rectângulo são testados.
- Na detecção ao nível do rectângulo, é o rectângulo de colisão da *sprite* que é utilizado para verificar se existe sobreposição com o outro elemento especificado.

É possível definir o rectângulo de colisão da *sprite* através do método `defineCollisionRectangle(int x, int y, int width, int height)`. As coordenadas são dadas relativamente ao canto superior esquerdo da *sprite* (que tem coordenadas (0, 0)). O rectângulo definido pode ser maior ou menor do que a *sprite*; no caso de ser maior, assume-se que os píxeis fora da *sprite* mas dentro do rectângulo são transparentes, para efeitos de detecção de colisões ao nível do píxel.

Agora podemos melhorar o exemplo anterior com detecção de colisões entre o balde e a esfera (apenas mostro o método `run()`):

```
public void run() {
    int largura = getWidth();
    int acertos = 0, falhas = 0;
    Graphics g = getGraphics();

    /* inicializar a posição da primeira bola */
    bola.setPosition(aleatório(largura), 0);

    while (ligado) {

       /* determinar a tecla pressionada */
       int keyStates = getKeyStates();

       if ((keyStates & RIGHT_PRESSED) != 0) {
           baldeX += velocidadeBalde;
       } else if ((keyStates & LEFT_PRESSED) != 0) {
           baldeX -= velocidadeBalde;
       }

       /* limpar o fundo (pintar a branco) */
       g.setColor(255, 255, 255);
       g.fillRect(0, 0, getWidth(), getHeight());

       /* actualizar posição do balde e desenhá-lo */
       balde.setPosition(baldeX, getHeight()-44);
       balde.paint(g);

       /* actualizar a bola */
       bola.move(0, VELOCIDADE);
       bola.nextFrame();
```

```
            bola.paint(g);

            /* se a bola bateu no balde, incrementa pontuação e volta a
            colocá-la no topo
            */
            if (bola.collidesWith(balde, false)) {
                if (bola.collidesWith(balde, true)) {
                    bola.setPosition(aleatório(largura), 0);
                    acertos++;
                }
            } else if (bola.getY() > getHeight()) {
                bola.setPosition(aleatório(largura), 0);
                falhas++;
            }

            /* escrever pontuação */
            g.setColor(0, 0, 0);
            g.drawString("Acertos: "+acertos + " Falhas: " + falhas, 0, 0,
            Graphics.TOP|Graphics.LEFT);

            flushGraphics();

            try {
                Thread.sleep(100);
            } catch(InterruptedException ie) {
                System.err.println(ie.getMessage());
            }
        }
    }
```

Devem ter reparado que o teste de colisão é feito usando simultaneamente os mecanismos de detecção ao nível do rectângulo e ao nível do píxel:

```
if (bola.collidesWith(balde, false)) {
    if (bola.collidesWith(balde, true)) {
        bola.setPosition(aleatório(largura), 0);
        acertos++;
    }
}
```

Esta é uma forma de melhorar o desempenho do jogo uma vez que a detecção ao nível do píxel é uma operação computacionalmente mais pesada do que a detecção ao nível do rectângulo. Se testarmos primeiro a colisão ao nível do rectângulo e o resultado for negativo, não faz sentido testar a colisão ao nível do píxel.

Ponto de referência

O posicionamento de *sprites* que temos visto até agora é feito relativamente ao canto superior esquerdo da *sprite*. No entanto, às vezes é mais fácil posicionar a *sprite* relativamente a um determinado ponto da própria *sprite*, por possuir um significado especial, por exemplo, a mão de uma personagem.

A classe *Sprite* permite-nos definir um ponto de referência para posicionamento da *sprite* através do método `defineReferencePixel(int x, int y)`.

Os parâmetros "x" e "y" são relativos ao canto superior esquerdo da *sprite*. Depois de definirmos o ponto de referência, podemos utilizar o método

```
setRefPixelPosition(int x, int y)
```

para posicionar a *sprite* de forma que o seu ponto de referência fique localizado no ponto (x, y) no sistema de coordenadas do objecto `Graphics`. De forma inversa podemos determinar o ponto do ecrã onde se encontra o píxel de referência da *sprite* através dos métodos `getRefPixelX()` e `getRefPixelY()`.

A Figura 12.4 mostra um exemplo da utilização do ponto de referência, de forma a parecer que o elefante está a comer um pedaço de vegetação.

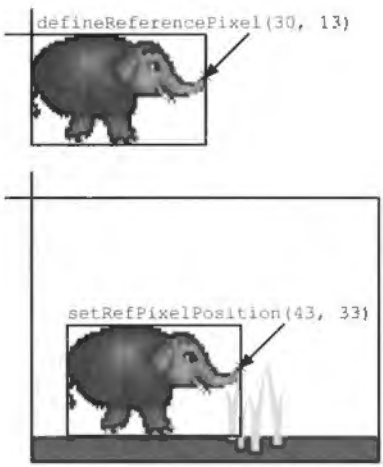


Figura 12.4 Ponto de referência de uma *sprite*.

Transformações

É comum nos jogos as *sprites* virarem-se no sentido do andamento, *i.e.*, se a personagem está a deslocar-se para a esquerda, então o seu corpo está virado para a esquerda e se estiver a deslocar-se para a direita, o seu corpo está virado para a direita.

Uma forma de conseguir isto é desenhar uma *sprite* com todas as orientações possíveis para a personagem. Esta solução tem o inconveniente de aumentar o tamanho da imagem e consumir mais memória.

Outra alternativa é utilizar as operações de transformações 2D de *sprites*. As transformações oferecidas pela API são transformações 2D muitos básicas: apenas rotações de

múltiplos de 90° e reflexões. No entanto, estas transformações combinadas com algumas *frames* diferentes para a *sprite* possibilitam um grande leque de orientações para uma *sprite*.

Para aplicar uma transformação à *sprite* utiliza-se o método `setTransform(int transform)` em que "`transform`" é uma das seguintes constantes da classe `Sprite`:

TRANS_MIRROR Faz com que a *sprite* seja reflectida sobre o eixo vertical.

TRANS_MIRROR_ROT180 Faz com que a *sprite* seja reflectida sobre o eixo vertical e depois rodada 180° no sentido horário.

TRANS_MIRROR_ROT270 Faz com que a *sprite* seja reflectida sobre o eixo vertical e depois rodada 270° no sentido horário.

TRANS_MIRROR_ROT90 Faz com que a *sprite* seja reflectida sobre o eixo vertical e depois rodada 90° no sentido horário.

TRANS_ROT180 Faz com que a *sprite* seja rodada 180° no sentido horário.

TRANS_ROT270 Faz com que a *sprite* seja rodada 270° no sentido horário.

TRANS_ROT90 Faz com que a *sprite* seja rodada 90° no sentido horário.

TRANS_NONE Nenhuma transformação é aplicada.

A Figura 12.5 mostra todas as transformações que se podem aplicar às *sprites*.

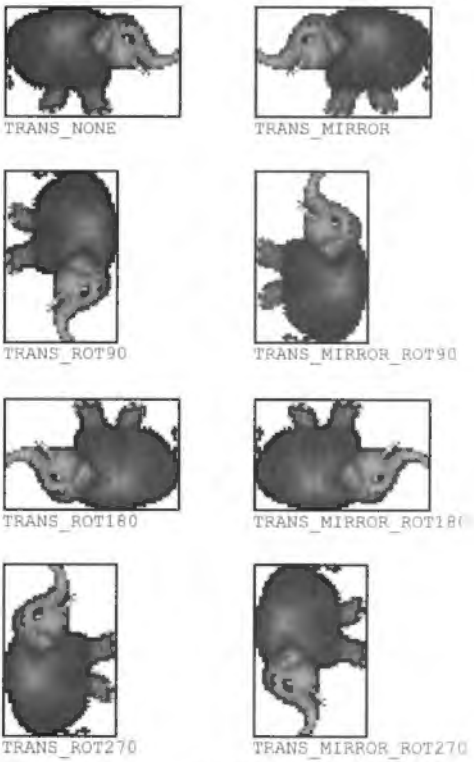


Figura 12.5 As transformações às *sprites*.

As transformações não são cumulativas, *i.e.*, invocar duas vezes `setTransform(TRANS_ROT90)` não faz com que a *sprite* seja rodada 180°. O ponto central utilizado para estas transformações é o ponto de referência da *sprite*.

Para exemplificar o uso de transformações vamos adaptar o exemplo da esfera a cair. Vamos simular a animação da esfera recorrendo apenas a duas *frames* diferentes e a transformações. As *frames* utilizadas são as primeiras duas da Figura 12.2. Para conseguirmos um efeito de rotação suave vamos combinar estas *frames* com transformações de rotação da seguinte forma:

Posição da animação	*Frame* Utilizada	Transformação Aplicada
0	0	TRANS_NONE
1	1	TRANS_NONE
2	0	TRANS_ROT90
3	1	TRANS_ROT90
4	0	RANS_ROT180
5	1	TRANS_ROT180
6	0	TRANS_ROT270
7	1	TRANS_ROT270

Para facilitar a programação vamos colocar os pares *frame*-transformação num *array* bidimensional:

```
private int frameTransformacao[][] = {{0, Sprite.TRANS_NONE},
                                      {1, Sprite.TRANS_NONE},
                                      {0, Sprite.TRANS_ROT90},
                                      {1, Sprite.TRANS_ROT90},
                                      {0, Sprite.TRANS_ROT180},
                                      {1, Sprite.TRANS_ROT180},
                                      {0, Sprite.TRANS_ROT270},
                                      {1, Sprite.TRANS_ROT270}};
```

Agora basta-nos animar a *sprite*:

```
public void run() {
    int largura = getWidth();

    /* indica a posição da animação*/
    int i = 0;
    Graphics g = getGraphics();

    /* inicializar a posição da primeira bola */
    bola.setPosition(aleatório(largura), 0);
    bola.defineReferencePixel(10, 10);
    while (ligado) {

       /* determinar a tecla pressionada */
       int keyStates = getKeyStates();

       if ((keyStates & RIGHT_PRESSED) != 0) {
           baldeX += velocidadeBalde;
       } else if ((keyStates & LEFT_PRESSED) != 0) {
           baldeX -= velocidadeBalde;
       }

       /* limpar o fundo (pintar a branco) */
       g.setColor(255, 255, 255);
       g.fillRect(0, 0, getWidth(), getHeight());

       /* actualizar posição do balde e desenhá-lo */
       balde.setPosition(baldeX, getHeight()-44);
       balde.paint(g);
```

```
            /* actualizar a bola */
            bola.move(0, VELOCIDADE);

            bola.setFrame(frameTransformacao[i][0]);
            bola.setTransform(frameTransformacao[i][1]);
            i++;
            if (i >= frameTransformacao.length) {
                i = 0;
            }
            bola.paint(g);

            /* se a bola chegou ao fundo do ecrã
            volta a colocá-la no topo */
            if (bola.getY() > getHeight()) {
                bola.setPosition(aleatório(largura), 0);
            }

            flushGraphics();

            try {
                Thread.sleep(100);
            } catch(InterruptedException ie) {
                System.err.println(ie.getMessage());
            }
        }
    }
```

Notem que defini o ponto de referência da *sprite* como sendo o centro da esfera, que é o ponto que pretendemos que seja o centro de rotação.

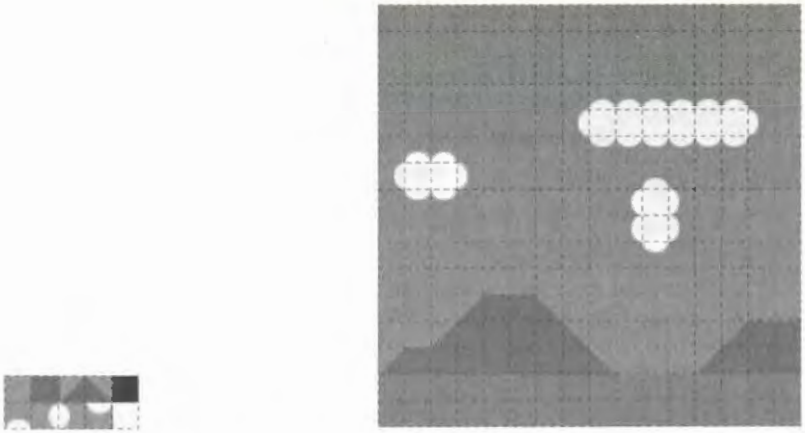
Desta forma conseguimos uma animação com oito posições diferentes recorrendo apenas a duas *frames* diferentes! Num jogo com muitos gráficos, este tipo de alternativa pode diminuir drasticamente o tamanho ocupado pela aplicação.

12.3.2. `TiledLayer`

A classe `TiledLayer` fornece-nos uma forma eficiente, do ponto de vista do número de imagens utilizadas, de construir *layers*.

Basicamente, uma `TiledLayer` é um elemento visual composto por uma grelha imaginária de células em que cada célula pode conter uma imagem – azulejo (*tile*). Isto permite-nos construir uma *layer* com uma imagem composta muito grande utilizando poucas e pequenas imagens.

Tal como com *frames* das *sprites*, os azulejos das *tiled layers* estão agrupados numa só imagem. A Figura 12.6(a) mostra uma imagem com 10 azulejos diferentes e a Figura 12.6(b) uma possível paisagem que se pode construir com esses 10 azulejos.



(a) Azulejos para uma `TiledLayer` (b) Paisagem construída com os azulejos

Figura 12.6 Construção de *layers* com azulejos.

Os azulejos da imagem são numerados de forma semelhante ao que acontece com as *sprites* (ver Figura 12.2), mas no caso dos azulejos a numeração começa em 1 em vez de começar em 0.

A composição da *tiled layer* é feita através de uma grelha, em que cada célula corresponde a um azulejo. Por exemplo, a paisagem da Figura 12.6 foi construída a partir da seguinte grelha:

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	6	6	6	6	6	6	1	1
1	1	1	1	1	1	1	7	10	10	10	10	10	10	8	1
1	6	6	1	1	1	1	1	9	9	9	9	9	9	1	1
7	10	10	8	1	1	1	1	1	1	6	1	1	1	1	1
1	9	9	1	1	1	1	1	1	7	10	8	1	1	1	1
1	1	1	1	1	1	1	1	1	7	10	8	1	1	1	1
1	1	1	1	1	1	1	1	1	1	9	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	3	2	2	4	1	1	1	1	1	1	1	1	1
1	1	3	2	2	2	2	4	1	1	1	1	1	3	2	2
3	2	2	2	2	2	2	2	4	1	1	1	3	2	2	2
5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5

A classe `TiledLayer` possui apenas um construtor:

```
TiledLayer(int columns, int rows, Image image,
    int tileWidth, int tileHeight)
```

Em que:

columns O número de colunas da grelha;

rows O número de linhas da grelha;

image A imagem que contém os azulejos;

tileWidth A largura de cada azulejo na imagem;

tileHeight A altura de cada azulejo na imagem.

Para definirmos o conteúdo de cada célula temos duas alternativas:

- `fillCells(int col, int row, int numCols, int numRows, int tileIndex)`
  Preenche uma região da grelha com um azulejo. A região a ser preenchida tem o canto superior esquerdo na intersecção da coluna "`col`" com a linha "`row`" e uma dimensão de "`numCols`" colunas por "`numRows`" linhas. "`tileIndex`" é o número do azulejo que irá ser colocado em todas as células da região.
- `setCell(int col, int row, int tileIndex)`
  Atribui um azulejo a uma célula.

No caso da paisagem que temos vindo a utilizar, o código para construir a *tiled layer* seria:

```
static int LARGURA_GRELHA = 16;
static int ALTURA_GRELHA = 16;

static int LARGURA_AZULEJO = 20;
static int ALTURA_AZULEJO = 20;

byte paisagem[][] =
 {{ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
 { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
 { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
 { 1, 1, 1, 1, 1, 1, 1, 1, 6, 6, 6, 6, 6, 6, 1, 1},
 { 1, 1, 1, 1, 1, 1, 1, 1, 7,10,10,10,10,10,10, 8, 1},
 { 1, 6, 6, 1, 1, 1, 1, 1, 9, 9, 9, 9, 9, 9, 1, 1},
 { 7,10,10, 8, 1, 1, 1, 1, 1, 1, 6, 1, 1, 1, 1, 1},
 { 1, 9, 9, 1, 1, 1, 1, 1, 1, 7,10, 8, 1, 1, 1, 1},
 { 1, 1, 1, 1, 1, 1, 1, 1, 1, 7,10, 8, 1, 1, 1, 1},
 { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 9, 1, 1, 1, 1, 1},
 { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
 { 1, 1, 1, 3, 2, 2, 4, 1, 1, 1, 1, 1, 1, 1, 1, 1},
 { 1, 1, 3, 2, 2, 2, 2, 4, 1, 1, 1, 1, 1, 3, 2, 2},
 { 3, 2, 2, 2, 2, 2, 2, 2, 4, 1, 1, 1, 3, 2, 2, 2},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5}};
```

```
try {
    Image tiles = Image.createImage("/tiles.png");
    TiledLayer fundo = new TiledLayer(LARGURA_GRELHA, ALTURA_GRELHA,
        tiles, LARGURA_AZULEJO, ALTURA_AZULEJO);

    for (int i = 0; i < ALTURA_GRELHA; i++) {
       for (int j = 0; j < LARGURA_GRELHA; j++) {
          fundo.setCell(j, i, paisagem[i][j]);
       }
    }
} catch (IOException ioe) {
}
```

As células da grelha não precisam de ter um azulejo associado. Podemos indicar células vazias colocando o valor 0 nessa célula. As células vazias são transparentes quando se desenha a *layer*.

Se quisermos, podemos mudar a imagem que serve de base à construção dos azulejos através do método

```
setStaticTileSet(Image image, int tileWidth,
    int tileHeight)
```

Se o novo conjunto de azulejos tiver tantos, ou mais, azulejos que o anterior, então o conteúdo da grelha é mantido inalterado. Caso contrário a grelha é limpa.

Células animadas

É comum nas *tiled layers* a necessidade de animar alguns azulejos, por exemplo para dar a aparência de movimento da água ou do vento nas árvores.

A classe `TiledLayer` dá-nos uma forma simples de animar azulejos. Para isso é preciso indicar que determinado azulejo deve ser animado, invocando o método

```
int createAnimatedTile(int staticTileIndex).
```

Este método cria um azulejo animado e retorna o seu identificador. O índice do azulejo inicialmente associado ao azulejo animado é especificado por "`staticTileIndex`".

Depois de obtermos o identificador do azulejo animado podemos invocar `setCell()` para colocar o azulejo na grelha.

A animação propriamente dita é conseguida alterando continuamente o azulejo estático associado ao azulejo animado. Isto é feito através do método

```
setAnimatedTile(int animatedTileIndex, int staticTileIndex)
```

Todas as células que contêm o azulejo animado irão ser alteradas para mostrarem o novo azulejo estático.

O processo genérico para utilizar azulejos animados é o seguinte:

```
byte paisagem[][] =
 {{ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
 {…} };
```

```
/* define os azulejos estáticos que compõem a animação do azulejo */
byte animaçãoAzulejo[] = {2, 3, 4};

/* o identificador do azulejo animado */
int azulejoAnimado;

try {
    Image tiles = Image.createImage("/tiles.png");
    TiledLayer fundo = new TiledLayer(LARGURA_GRELHA, ALTURA_GRELHA,
        tiles, LARGURA_AZULEJO, ALTURA_AZULEJO);

    for (int i = 0; i < ALTURA_GRELHA; i++) {
       for (int j = 0; j < LARGURA_GRELHA; j++) {
           fundo.setCell(j, i, paisagem[i][j]);
       }
    }
} catch (IOException ioe) {
}

try {
    /* criar o azulejo animado e indicar que o primeiro azulejo estático
       a ser exibido é o azulejo número 2
    */
    azulejoAnimado = fundo.createAnimatedTile(2);

    /* colocar o azulejo animado na grelha */
    fundo.setCell(5, 10, azulejoAnimado);
} catch(IndexOutOfBoundsException ioobe) {}
/* ciclo do jogo */
i = 0;
while (aJogar) {
    [...]

    /* animar o azulejo */
    fundo.setAnimatedTile(azulejoAnimado, animaçãoAzulejo[i]);
    i++;
    if (i == animaçãoAzulejo.length) {
       i = 0;
    }
    [...]
}
```

Os identificadores dos azulejos animados devolvidos por `createAnimatedTile()` são números negativos, por isso não existe conflito com os azulejos estáticos.

Automatizar a criação de *Tiled Layers*

Num jogo com um cenário muito pequeno pode ser viável construir esse cenário "à mão", isto é, definir a grelha do *tiled layer*, célula a célula. No entanto, para cenários médios ou grandes é aconselhável utilizar uma ferramenta que nos permita construir o cenário visualmente, de forma a podermos experimentar mais facilmente diferentes configurações.

Não vou entrar em grandes descrições de ferramentas para este fim, porque não é esse o objectivo deste livro. Vou apenas apontar uma ferramenta, *open source*, que pode ser utilizada para construir cenários para jogos: o Tile Studio [Wie].

A Figura 12.7 mostra a janela da aplicação com um projecto aberto. O Tile Studio permite desenhar azulejos para serem usados em *sprites* e cenários e permite construir os próprios cenários (mapas, na terminologia da aplicação). Uma das vantagens deste género de aplicações é o facto de permitirem exportar os mapas para utilização directa no código do programa. O Tile Studio permite exportar para diversas linguagens. Infelizmente, não possui a opção para Java mas a versão Visual C++ serve perfeitamente. O mapa é exportado na forma (o ficheiro de código gerado contém mais do que apenas isto):

```
signed short tilesMap1MapData[16][16] =
 {{ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
 { 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2},
 { 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3},
 { 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6},
 { 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7},
 { 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8},
 { 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9},
 {10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10},
 { 1, 1, 1, 1, 1, 1, 1, 4, 1, 1, 1, 1, 1, 1, 1, 1},
 { 1, 1, 1, 1, 1, 1, 1, 4, 1, 1, 1, 1, 1, 1, 1, 1},
 { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
 { 1, 1, 1, 3, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
 { 1, 1, 1, 3, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
 { 1, 1, 1, 2, 2, 1, 5, 5, 1, 1, 1, 1, 1, 1, 1, 1}};
```

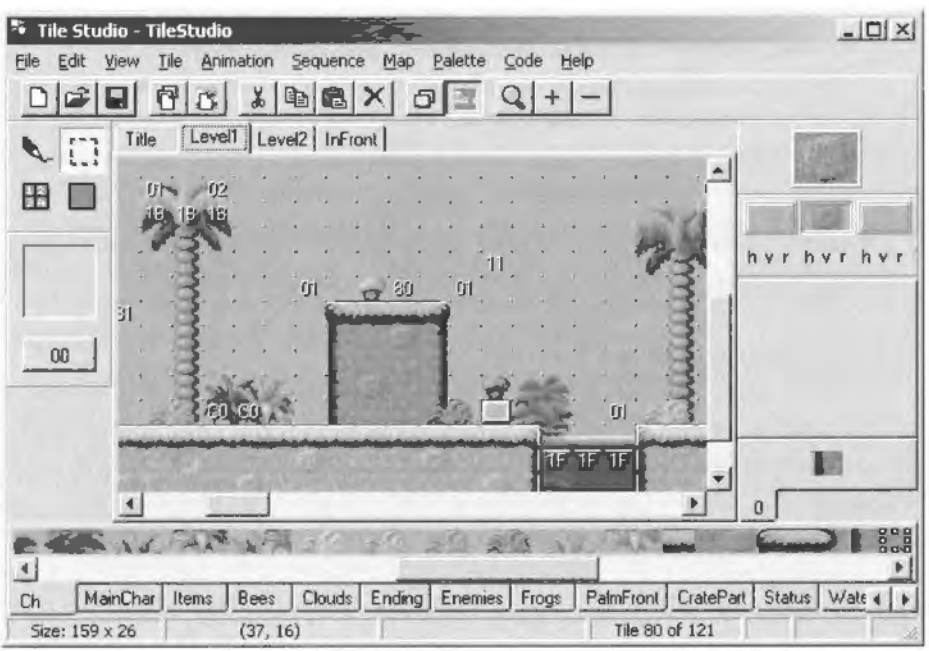


Figura 12.7 TileStudio – Um editor de *tiles*.

Pelo que basta alterar a sintaxe da declaração de *arrays* para podermos utilizar o código directamente no nosso jogo.

Para mostrar a utilidade do Tile Studio, vamos usar um dos cenários-exemplo do programa e aplicá-lo no jogo. O cenário utilizado é o do exemplo Charlie.tsp. Vamos aplicar o cenário num jogo que permite ao utilizador movimentar uma abelha (também retirada do exemplo do Tile Studio) através do cenário. A Figura 12.8 mostra o ecrã do jogo. O código do `GameCanvas` é apresentado a seguir (o mapa foi cortado uma vez que é demasiado extenso para ser apresentado na sua totalidade):

Exemplo 12.3: `TilesCanvas` – Cenário do Tile Studio

```
import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;
import java.io.*;
import java.util.*;

public class TilesCanvas extends GameCanvas implements Runnable {
    private static final int LARGURA_GRELHA = 159;
    private static final int ALTURA_GRELHA = 26;

    /* A sprite da abelha */
    private Sprite abelha;

    /* O fundo do jogo (gerado pelo TileStudio) */
    private TiledLayer fundo;
    private int[][] grelha =
    {{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    [resto da grelha obtida do Tile Studio aqui]
    };

    /* A thread do motor do jogo */
    private Thread motor;
    /* A thread executa enquanto ligado == true */
    private boolean ligado = true;

    public TilesCanvas() {
       super(false);

       /* criar a sprite e o fundo */
       try {

           Image abelhaImg = Image.createImage("/bees.png");
           abelha = new Sprite(abelhaImg, 20, 20);

           Image tiles = Image.createImage("/ch.png");
           fundo = new TiledLayer(LARGURA_GRELHA, ALTURA_GRELHA, tiles,
               20, 16);

           for (int i = 0; i < ALTURA_GRELHA; i++) {
               for (int j = 0; j < LARGURA_GRELHA; j++) {
                   fundo.setCell(j, i, grelha[i][j]);
               }
           }
       } catch (IOException ioe) {
           System.err.println("Impossivel criar layer: " +
               ioe.getMessage());
```

```
    }

    /* iniciar o motor do jogo */
    motor = new Thread(this);
    motor.start();
}

public void run() {
    int largura = getWidth();
    int altura = getHeight();

    int moveX = 0, moveY = 0;

    Graphics g = getGraphics();

    /* colocar a layer de forma a coincidir com o fundo do ecrã */
    fundo.setPosition(0, getHeight()-fundo.getHeight());

    /* colocar a abelha no centro do ecrã */
    abelha.setPosition(largura/2, altura/2);
    /* ponto de referência no centro da abelha uma vez que vamos usar
       transformações */
    abelha.defineReferencePixel(10, 10);
    while (ligado) {
        /* determinar a tecla pressionada */
        int keyStates = getKeyStates();

        moveX = 0;
        moveY = 0;
        if ((keyStates & RIGHT_PRESSED) != 0) {
            abelha.setTransform(Sprite.TRANS_NONE);
            moveX = -5;
        } else if ((keyStates & LEFT_PRESSED) != 0) {
            abelha.setTransform(Sprite.TRANS_MIRROR);
            moveX = 5;
        } else if ((keyStates & UP_PRESSED) != 0) {
            abelha.setTransform(Sprite.TRANS_ROT270);
            moveY = 5;
        }else if ((keyStates & DOWN_PRESSED) != 0) {
            abelha.setTransform(Sprite.TRANS_ROT90);
            moveY = -5;
        }
        fundo.move(moveX, moveY);
        /* se colidir, desfazer o movimento */
        if (abelha.collidesWith(fundo, false) ) {
            if (abelha.collidesWith(fundo, true) ) {
                fundo.move(-moveX, -moveY);
            }
        }

        /* limpar o fundo (pintar a azul - céu) */
        g.setColor(170, 255, 255);
        g.fillRect(0, 0, largura, altura);

        fundo.paint(g);

        abelha.nextFrame();
        abelha.paint(g);

        flushGraphics();

        try {
            Thread.sleep(50);
        } catch(InterruptedException ie) {
```

```
                System.err.println(ie.getMessage());
            }
        }
    }

    public void parar() {
        ligado = false;
        try {
            motor.join();
        } catch (InterruptedException ie) {
            System.err.println(ie.getMessage());
        }
    }
}
```

Figura 12.8 `Tiles` – Um jogo implementado com *tiles*.

12.4. O Gestor de *Layers*

Falta apenas mencionar mais uma funcionalidade da API de jogos MIDP 2.0: o gestor de *layers*.

Para facilitar a organização do jogo quando este é composto por várias *layers*, existe uma classe – `LayerManager` – que simplifica algum do trabalho do programador.

Não há muito a dizer sobre esta classe. Basicamente, serve para automatizar o processo de desenho das várias *layers*. O gestor de *layers* mantém uma lista ordenada de *layers* de forma que para desenharmos todas as *layers* do nosso jogo basta pedir ao gestor de *layers* – a invocação do `paint()` de cada *layer* é feita automaticamente pelo gestor e pela ordem correcta. Para além disso, o gestor de *layers* permite-nos definir uma janela de visualização sobre as layers. A Figura 12.9 exemplifica o conceito da janela de visualização.

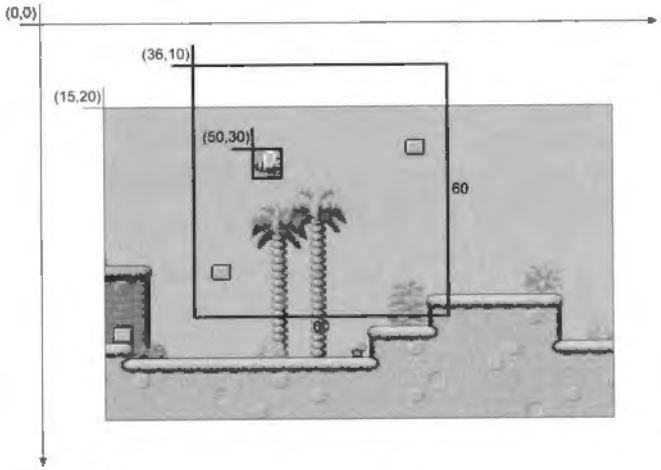


Figura 12.9 Janela de visualização do gestor de *layers*.

Apenas a área no interior da janela de visualização é desenhada. A posição da janela de visualização pode ser ajustada continuamente no jogo para dar a ilusão de *scrolling* pelo cenário.

Os métodos fornecidos pela `LayerManager` são os seguintes:

- `void append(Layer l)`
  Adiciona uma *layer* ao gestor. As *layers* são adicionadas atrás das já existentes, *i.e.*, devemos adicionar as *layers* começando nas que estão mais próximas do utilizador. É atribuído um índice a cada *layer* adicionada, começando em zero.
- `Layer getLayerAt(int index)`
  Retorna a *layer* com o índice especificado.
- `int getSize()`
  Retorna o número de *layers* geridas pelo `LayerManager`.
- `void insert(Layer l, int index)`
  Insere uma *layer* no índice especificado. Se a *layer* tiver sido inserida anteriormente, é removida primeiro.
- `void remove(Layer l)`
  Remove a *layer* do gestor.
- `void setViewWindow(int x, int y, int width, int height)`
  Define a janela de visualização do gestor de *layers*. As coordenadas são dadas no sistema de coordenadas do gestor de *layers*. Quando se utiliza o gestor de *layers* a posição das *layers* é calculada relativamente ao sistema de coordenadas do gestor e não do `Graphics`.

- `void paint(Graphics g, int x, int y)`
  Desenha o conteúdo da janela de visualização no ecrã na posição (x, y).

Para ilustrar o uso do `LayerManager` vamos alterar o Exemplo 12.3 para utilizar o gestor de *layers*:

Exemplo 12.4: `TilesCanvas` – Utilização de `LayerManager`

```
import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;
import java.io.*;
import java.util.*;

public class TilesCanvas extends GameCanvas implements Runnable {
    private static final int LARGURA_GRELHA = 159;
    private static final int ALTURA_GRELHA = 26;

    /* A sprite da abelha */
    private Sprite abelha;

    /* O fundo do jogo (gerado pelo TileStudio) */
    private TiledLayer fundo;
    private int[][] grelha =
    {{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    [resto da grelha obtida do Tile Studio aqui]
    };

    /* O gestor de layers */
    LayerManager gestor;

    /* A thread do motor do jogo */
    private Thread motor;
    /* A thread executa enquanto ligado == true */
    private boolean ligado = true;

    public TilesCanvas() {
       super(false);

       /* criar a sprite e o fundo */
       try {

           Image abelhaImg = Image.createImage("/bees.png");
           abelha = new Sprite(abelhaImg, 20, 20);

           Image tiles = Image.createImage("/ch.png");
           fundo = new TiledLayer(LARGURA_GRELHA, ALTURA_GRELHA, tiles,
               20, 16);

           for (int i = 0; i < ALTURA_GRELHA; i++) {
               for (int j = 0; j < LARGURA_GRELHA; j++) {
                   fundo.setCell(j, i, grelha[i][j]);
               }
           }
       } catch (IOException ioe) {
           System.err.println("Impossivel criar layer: " +
               ioe.getMessage());
       }
```

```
        gestor = new LayerManager();
        gestor.append(abelha);
        gestor.append(fundo);

        /* iniciar o motor do jogo */
        motor = new Thread(this);
        motor.start();
    }

    public void run() {
        int larguraEcrã = getWidth();
        int alturaEcrã = getHeight();

        int alturaFundo = fundo.getHeight();

        int moveX = 0, moveY = 0;

        Graphics g = getGraphics();
        /* inicializar a janela de visualização */
        gestor.setViewWindow(0, (alturaFundo-alturaEcrã)/2,
            larguraEcrã-10, alturaEcrã-10);

        /* colocar a abelha no centro do ecrã */
        abelha.setPosition((larguraEcrã-10)/2,
            (alturaEcrã-10)/2+(alturaFundo-alturaEcrã)/2);

        /* ponto de referência no centro da abelha uma vez que vamos usar
        transformações */
        abelha.defineReferencePixel(10, 10);
        while (ligado) {

            /* determinar a tecla pressionada */
            int keyStates = getKeyStates();

            moveX = 0;
            moveY = 0;
            if ((keyStates & RIGHT_PRESSED) != 0) {
                abelha.setTransform(Sprite.TRANS_NONE);
                moveX = -5;
            } else if ((keyStates & LEFT_PRESSED) != 0) {
                abelha.setTransform(Sprite.TRANS_MIRROR);
                moveX = 5;
            } else if ((keyStates & UP_PRESSED) != 0) {
                abelha.setTransform(Sprite.TRANS_ROT270);
                moveY = 5;
            }else if ((keyStates & DOWN_PRESSED) != 0) {
                abelha.setTransform(Sprite.TRANS_ROT90);
                moveY = -5;
            }
            fundo.move(moveX, moveY);

            /* se colidir, desfazer o movimento */
            if (abelha.collidesWith(fundo, false) ) {
                if (abelha.collidesWith(fundo, true) ) {
                    fundo.move(-moveX, -moveY);
                }
            }

            /* limpar o fundo (pintar a azul - céu) */
            g.setColor(170, 255, 255);
            /* deixamos uma borda a branco à volta do jogo */
            g.fillRect(5, 5, larguraEcrã-10, alturaEcrã-10);
```

```
                abelha.nextFrame();

                gestor.paint(g, 5, 5);
                flushGraphics();

                try {
                    Thread.sleep(50);
                } catch(InterruptedException ie) {
                    System.err.println(ie.getMessage());
                }
            }
        }

        public void parar() {
            ligado = false;
            try {
                motor.join();
            } catch (InterruptedException ie) {
                System.err.println(ie.getMessage());
            }
        }
    }
```

Neste exemplo, utilizamos o `LayerManager` para definir uma janela menor do que o ecrã de forma a colocarmos uma borda vazia à volta da área do jogo, como se pode ver na Figura 12.10. Uma vez que não alteramos a posição inicial da *layer* de fundo, esta é posicionada, por omissão, em (0, 0), no sistema de coordenadas do gestor de *layers*.

Figura 12.10 Ecrã do jogo com gestor de *layers*.

Glossário

***Abstract Windowing Toolkit* (AWT)** O *Abstract Windowing Toolkit* (AWT) é o conjunto de classes básicas do Java SE para a construção de interfaces gráficas com o utilizador. O AWT permite construir objectos gráficos como janelas, caixas de diálogo, botões, etc.

***Application Management Software* (AMS)** O *Application Management Software* (AMS) é o software residente nos dispositivos MID, responsável por gerir a instalação, actualização e remoção das aplicações Java e por gerir o seu ciclo de vida.

***Compact Virtual Machine* (CVM)** A *Compact Virtual Machine* (CVM) é a máquina virtual usada na configuração CDC. É uma máquina virtual completa, optimizada para dispositivos utilizados na electrónica de consumo.

***Connected Device Configuration* (CDC)** A *Connected Device Configuration* (CDC) é uma das configurações do Java ME. É usada em dispositivos com mais capacidade (de memória e processamento) do que os dispositivos-alvo da CLDC. Exemplos de dispositivos que podem ser incluídos nesta categoria são: *set-top boxes* para televisores, alguns PDA, sistemas de navegação para automóveis, etc.

***Connected Limited Device Configuration* (CLDC)** A *Connected Limited Device Configuration* (CLDC) é uma das configurações do Java ME. Esta configuração é orientada para dispositivos muito limitados, i.e., telemóveis e PDA, com capacidade de ligação à rede.

***Foundation Profile* (FP)** O *Foundation Profile* (FP) é um conjunto de API que suportam dispositivos com recursos limitados, sem um sistema de interface gráfica *standard*. Combinado com a configuração *Connected Device Configuration* (CDC), o FP fornece um ambiente aplicacional completo para dispositivos de electrónica de consumo e embebidos.

***Generic Connection Framework* (GCF)** A *Generic Connection Framework* (GCF) é uma hierarquia de classes desenhadas para fornecer uma abstracção comum aos vários tipos de conexões. Originalmente desenhada para a plataforma CLDC do Java ME, foi entretanto adoptada também para a CDC.

***Information Module Profile* (IMP)** O *Information Module Profile* (IMP) é um perfil para a CLDC, desenhado para dispositivos que não necessitam de interface gráfica com o utilizador. O IMP é um subconjunto do MIDP.

***JAR* (JAR)** O ficheiro JAR (diminutivo de *Java ARchive*) é um ficheiro ZIP utilizado para distribuir um conjunto de classes Java, ficheiros de recursos e metadados, que constituem uma aplicação Java.

***Java Application Descriptor* (JAD)** O *Java Application Descriptor* (JAD) é um ficheiro de texto que acompanha, normalmente, as MIDlets e que descreve a aplicação em termos de

recursos necessários, entre outras coisas, por forma que o dispositivo seja capaz de determinar se é capaz de a executar antes de instalar a MIDlet.

***Java Community Process* (JCP)** O *Java Community Process* (JCP) é o processo instituído pela Sun Microsystems para a criação de novas API Java e para a evolução das já existentes. Neste processo participam normalmente organizações e empresas directamente relacionadas e interessadas na forma como as API são desenhadas. Neste processo também podem participar indivíduos e o público em geral.

***Java Native Interface* (JNI)** O *Java Native Interface* (JNI) é uma norma que define como um programa escrito em Java pode invocar funções ou programas escritos em linguagens como o C.

***Java Platform, Enterprise Edition* (Java EE)** O *Java Platform, Enterprise Edition* (Java EE), anteriormente designado por Java 2 *Platform, Enterprise Edition* (J2EE), é a plataforma Java orientada para o desenvolvimento de aplicações empresariais. O Java EE define o *standard* para desenvolvimento de aplicações de várias camadas e baseadas em componentes.

***Java Platform, Micro Edition* (Java ME)** O *Java Platform, Micro Edition* (Java ME), anteriormente designado por *Java 2 Platform, Micro Edition* (J2ME), é um ambiente de desenvolvimento Java orientado para a electrónica de consumo. É a mais recente tecnologia Java destinada a dispositivos com limitações de memória e de processamento quando comparados com os PC de secretária.

***Java Platform, Standard Edition* (Java SE)** O *Java Platform, Standard Edition* (Java SE), anteriormente designado por *Java 2 Platform, Standard Edition* (J2SE), é o ambiente de desenvolvimento de aplicações em *desktops* e servidores.

***Java Specification Request* (JSR)** Os JSR são as descrições das propostas de especificação e especificações finais para a plataforma Java desenvolvidas através do *Java Community Process* (JCP). Em qualquer momento, existem inúmeros JSR a passar pelo processo de revisão e aprovação.

***Java Virtual Machine* (JVM)** A Java Virtual Machine (JVM), às vezes também chamada de *Java Interpreter* ou *Java Runtime*, converte bytecodes em código-máquina. A JVM executa sobre um sistema operativo e governa a execução de um programa Java, gerindo a memória, *threads*, IO, etc.

***Java Virtual Machine Debugger Interface* (JVMDI)** A *Java Virtual Machine Debugger Interface* (JVMDI) é uma interface de programação utilizada por *debuggers* e outras ferramentas de programação. A JVMDI permite inspeccionar o estado e controlar a execução de aplicações a executar numa *Java Virtual Machine*. Esta interface tornou-se obsoleta na nova versão do Java SE e foi substituída pela *JVM Tool Interface* (JVM TI).

***Java Virtual Machine Profiler Interface* (JVMPI)** A *Java Virtual Machine Profiler Interface* (JVMPI) é uma interface de programação que permite o desenvolvimentos de programas para

efectuar o *profiling* de aplicações Java. A JVMPI é uma interface de dois sentidos: por um lado, notifica o agente de vários eventos como alocação de memória, início de *threads*, etc.; por outro, permite que o agente emita pedidos para obter mais informação sobre a execução da aplicação. Esta interface tornou-se obsoleta na nova versão do Java SE e foi substituída pela *JVM Tool Interface* (JVMTI).

***JVM Tool Interface* (JVMTI)** A *JVM Tool Interface* (JVMTI) é uma nova interface de programação para ser utilizada por ferramentas de *debugging* e *profiling*. A JVMTI veio substituir a *Java Virtual Machine Profiler Interface* (JVMPI) e a *Java Virtual Machine Debug Interface* (JVMDI).

***Kilo Virtual Machine* (KVM)** A *Kilo Virtual Machine* (KVM) é uma máquina virtual Java desenhada especificamente para dispositivos muito limitados ao nível da memória e poder de processamento. A KVM é a máquina virtual Java que serve de base à configuração *Connected Limited Device Configuration* (CLDC).

KToolbar O KToolbar é uma ferramenta do *Sun Java Wireless Toolkit* que permite compilar, pré-verificar e executar programas MIDP.

MIDlet *MIDlet* é o nome dado às aplicações escritas para o perfil *Mobile Information Device Profile* (MIDP) do Java ME e que correm em dispositivos como telemóveis e *Personal Digital Assistants* (PDA).

***Mobile 3D Graphics API* (M3G)** A *Mobile 3D Graphics API* (M3G) é um pacote opcional para a CLDC 1.1 que permite trabalhar com gráficos tridimensionais.

***Mobile Information Device* (MID)** Um *Mobile Information Device* (MID) é um dispositivo de informação móvel. Inserem-se nesta categoria dispositivos como telemóveis, *Personal Digital Assistants* (PDA) ou *pagers*.

***Mobile Information Device Profile* (MIDP)** O *Mobile Information Device Profile* (MIDP) é um perfil Java ME para a configuração *Connected Limited Device Configuration* (CLDC). É o perfil mais comum da tecnologia Java ME para dispositivos móveis como telemóveis, *pagers*, *Personal Digital Assistants* (PDA), etc. O MIDP fornece API para trabalhar com a interface com o utilizador, armazenamento persistente, conexões à rede, jogos e áudio.

***Mobile Media API* (MMAPI)** A *Mobile Media API* (MMAPI) é um pacote opcional para o Java ME que fornece uma interface simples e flexível para trabalhar com dispositivos multimédia. Esta API permite reproduzir e capturar áudio e vídeo. Parte deste pacote opcional (respeitante ao áudio) foi integrado no perfil *Mobile Information Device Profile* versão 2.0 (MIDP 2.0).

***Musical Instrument Digital Interface* (MIDI)** O *Musical Instrument Digital Interface* (MIDI) é uma norma que permite que instrumentos electrónicos como sintetizadores, sequenciadores, etc., comuniquem entre si. O MIDI é também um formato de ficheiro musical

que especifica os instrumentos e as notas tocadas por cada instrumento ao longo do tempo.

***Over The Air* (OTA)** A expressão *Over The Air* (OTA) significa a capacidade de descarregar e instalar aplicações (MIDlets) através de uma rede sem fios, a pedido do utilizador. Os dispositivos conformes com a especificação *Mobile Information Device Profile* versão 2.0 (MIDP 2.0) são obrigados a suportar este mecanismo de instalação de MIDlets.

***Personal Basis Profile* (PBP)** O *Personal Basis Profile* (PBP) é um perfil para a configuração *Connected Device Configuration* (CDC) que suporta dispositivos com recursos limitados, mas com interface gráfica com o utilizador (ao contrário do perfil *Foundation Profile* (FP) que não suporta interface gráfica). Basicamente, o perfil PBP adiciona, às classes do FP, suporte para os componentes *lightweight* AWT.

***Personal Digital Assistant* (PDA)** Um *Personal Digital Assistant* (PDA) é um computador de mão ou um organizador de informação pessoal.

***Personal Profile* (PP)** O *Personal Profile* (PP) é um perfil para a *Connected Device Configuration* (CDC) que adiciona, ao perfil *Personal Basis Profile* (PBP), suporte para o modelo de aplicação *applet* e compatibilidade total com a API AWT.

***Portable Network Graphics* (PNG)** O *Portable Network Graphics* (PNG) é um formato de imagem desenvolvido pelo consórcio W3C. O PNG é um substituto, sem patentes e mais flexível, ao formato GIF. O formato PNG é o formato adoptado pela especificação *Mobile Information Device Profile* (todos os dispositivos conformes com a especificação devem suportar este formato).

***Record Management System* (RMS)** O *Record Management System* (RMS) é uma API do perfil Mobile *Information Device Profile* (MIDP) que fornece uma interface para um sistema de armazenamento persistente no dispositivo móvel.

***Remote Method Invocation* (RMI)** O *Remote Method Invocation* (RMI) é um protocolo que permite que as aplicações Java acedam a objectos Java, através de rede, como se se tratassem de objectos locais.

***Scaloble Polyphony MIDI* (SP-MIDI)** O *Scalable Polyphony MIDI* (SP-MIDI) é um formato MIDI adaptado para dispositivos móveis. Basicamente, esta especificação permite que os dispositivos adaptem a reprodução do som aos recursos que possuem (normalmente eliminando algumas partes da música).

***Sun Java Wireless Toolkit* (WTK)** O *Sun Java Wireless Toolkit* (WTK) é um conjunto de ferramentas para desenvolver aplicações baseadas no perfil *Mobile Information Device Profile*. O WTK inclui um ambiente de emulação de dispositivos móveis, ferramentas para optimizar o desempenho da aplicação, documentação e exemplos de aplicações.

***Uniform Resource Indicator* (URI)** Um *Uniform Resource Identifier* (URI) é uma forma genérica de identificar recursos na Internet. Existem três tipos de URI: *Uniform Resource Classification* (URC), *Uniform Resource Locator* (URL) e *Uniform Resource Name* (URN).

***Uniform Resource Locator* (URL)** Um *Uniform Resource Locator* (URL) é um tipo de *Uniform Resource Indicator* (URI) que consiste num endereço de um recurso que pode ser interpretado por um servidor Web.

***Web Services API* (WSA)** A *Web Services API* (WSA) é um pacote opcional para a plataforma Java ME que fornece uma API que permite que os dispositivos Java ME sejam clientes de *web services*. O modelo de programação é consistente com a plataforma normalizada de *web services*.

***Wireless Messaging API* (WMA)** A *Wireless Messaging API* (WMA) é um pacote opcional para a plataforma Java ME que fornece um mecanismo para enviar mensagens *Short Message Service* (SMS).

Xlet A *xlet* é um modelo de aplicação semelhante à *applet* ou MIDlet. A *xlet* é suportada pelos perfis *Personal Basis Profile* (PBP) e *Personal Profile* (PP).

Lista de Atributos das MIDlets

- (mf) Significa que o atributo pode aparecer no ficheiro de manifesto
- **(mf)** Significa que tem de aparecer no ficheiro de manifesto
- (jad) Significa que o atributo pode aparecer no descritor da aplicação.
- **(jad)** Significa que o atributo tem de aparecer no descritor da aplicação

Gerais

MIDlet-Name (mf, jad) O nome da MIDlet Suite.

MIDlet-Version (mf, jad) O número de versão da MIDlet Suite. O formato do número é maior.menor.micro, e.g., 1.0.1. O número de versão é usado pelo AMS para determinar se é uma actualização ou instalação e também para informar o utilizador.

MIDlet-Vendor (mf, jad) O nome da organização que fornece a MIDlet Suite.

MIDlet-Icon (mf, jad) O nome de um ficheiro PNG usado como ícone da MIDlet Suite

MIDlet-Description (mf, jad) A descrição da MIDlet Suite.

MIDlet-Info-URL (mf, jad) Um URL que aponta para mais informação que descreve a MIDlet Suite.

MIDlet-<n> (mf) O nome, ícone e classe principal da MIDlet número <n> no ficheiro JAR, separados por vírgulas. <n> tem de começar em 1 e devem ser usados números consecutivos.

MIDlet-Jar-URL (jad) O URL de onde o ficheiro JAR pode ser descarregado.

MIDlet-Jar-Size (jad) O número de bytes ocupados pelo ficheiro JAR

MIDlet-Data-Size (mf, jad) O número mínimo de bytes de armazenamento persistente que a MIDlet necessita. O valor por omissão é zero.

MicroEdition-Profile (mf) O perfil Java ME necessário. Por exemplo, MIDP-2.0.

MicroEdition-Configuration (mf) A configuração Java ME necessária. Por exemplo, CLDC-1.0.

MIDlet-Install-Notify (jad) O URL para o qual um pedido HTTP POST é enviado para relatar o resultado da instalação. O URL não pode exceder os 256 caracteres.

MIDlet-Delete-Notify (jad) O URL para o qual um pedido HTTP POST é enviado para relatar que a MIDlet Suite foi apagada do dispositivo. O URL não pode exceder os 256 caracteres.

MIDlet-Delete-Confirm (jad) A mensagem que o sistema mostra ao utilizador para confirmar a remoção da MIDlet Suite.

Segurança

MIDlet-Permissions (mf, jad) Lista de permissões necessárias separadas por vírgula. Neste atributo devem ser listadas as permissões, sem as quais a MIDlet não pode funcionar.

MIDlet-Permissions-opt (mf, jad) Lista de permissões opcionais separadas por vírgula. Neste atributo devem ser listadas as permissões que não são absolutamente necessárias para o funcionamento da aplicação.

Push Registry

MIDlet-Push-<n> (jad) Indica um registo *push*. Cada entrada deve obedecer ao formato: <URL de Conexão>, <Nome da Classe>, <Filtro>. O valor <n> deve começar em 1 e devem ser usados números consecutivos para cada uma das entradas.

Referências

[GJS96] James Gosling, Bill Joy and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[Gos] James Gosling. *A Brief History of the Green Project*. `http://today.java.net/jag/old/green`.

[Gos95] James Gosling. *Java: an Overview*. `http://today.java.net/jag/old/OriginalJavaWhitepaper.pdf`, Fevereiro de 1995.

[jT03] *jScience Technologies*. MathFP, 2003. `http://www.jscience.net`.

[Knu03] Jonathan Knudsen. *Wireless Java*. Apress, 2nd edition, 2003.

[Lam94] Leslie Lamport. LATEX – *A Document Preparation System*. Addison-Wesley Professionaly, 2nd edition, 1994.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[Mah02] Qusay H. Mahmoud. *Learning Wireless Java*. O'Reilly, 1st edition, 2002.

[Mic00] Sun Microsystems. *Mobile Information Device Profile 1.0a*. Especificação, Sun Microsystems, Dezembro de 2000.

[Mic02] Sun Microsystems. *Mobile Information Device Profile 2.0*. Especificação, Sun Microsystems, Novembro de 2002.

[MR05] Luís Miguens and Pedro Remelhe. *Aplicações Móveis com J2ME*. FCA, 1st edition, 2005.

[Sun99] Sun. *Code Conventions for the Java Programming Language*. `http://java.sun.com/docs/codeconv/`, Abril de 1999.

[Sun00] Sun. *Connected Limited Device Configuration – Version 1.0a*. Especificação, Sun Microsystems, Maio de 2000.

[Wie] Mike Wiering. *Tile Studio*. `http://tilestudio.sourceforge.net`.

Índice Remissivo

D

G

H

N

O

P

S

Jorge Cardoso é licenciado em Engenharia Informática e Computação pela Faculdade de Engenharia da Universidade do Porto.
É docente e investigador na Escola das Artes da Universidade Católica Portuguesa desde 2003, onde lecciona disciplinas na área das tecnologias, programação de computadores e interfaces.

Jorge Cardoso

JAVA PARA TELEMÓVEIS
MIDP 2.0

FEUP edições