Programação por Objectos

Java

Parte 1: Introdução

História – versões (1)

- **[1995] Versão 1.0**, denominada ***Java Development Kit* (JDK)**
  - 212 classes em 8 pacotes
  - Lento, muitos *bugs*, mas com *Applets*

- **[1997] Versão 1.1**, denominada JDK 1.1
  - 504 classes em 23 pacotes
  - Melhoria na eficiência da JVM
  - Principais extensões: classes aninhadas, JavaBeans, JDBC, RMI, ...

- **[1998] Versão 1.2**, a partir daqui denominada ***Java 2 Platform* (J2SE)**
  - 1520 classes em 59 pacotes
  - JVM da Sun com compilador JIT
  - Principais extensões: Swing, colecções, ...
  - Nome de código *Playground*

História – versões (2)

- **[2000] Versão 1.3**, denominada J2SE 1.3
  - 1842 classes em 76 pacotes
  - Melhoria na eficiência da JVM
  - Nome de código *Kestrel*

- **[2002] Versão 1.4**, denominada J2SE 1.4
  - 2291 classes em 135 pacotes
  - Melhoria na eficiência da JVM
  - Principais extensões: asserções, transferência de excepções, segurança e criptografia, ...
  - Disponibilizado em 3 plataformas:
    - Java 2 Micro Edition (J2ME), para Telemóveis e PDAs
    - Java 2 Standard Edition (J2SE), para desktop
    - Java 2 Enterprise Edition (J2EE), para aplicações empresariais

História – versões (3)

  - Desenvolvimento facilitado por ambientes poderosos: IDE (*Integrated Development Environment*)
    - NetBeans, da Sun
    - Eclipse, da IBM
  - Nome de código *Merlin*

- **[2004] Versão 5.0**, denominada J2SE 5.0
  - 3000 classes em 165 pacotes
  - Principais extensões: genéricos, enumerados, tipos primitivos e classes de embrulho, número variável de argumentos
  - **Versão anteriormente numerada 1.5**
  - Nome de código *Tiger*

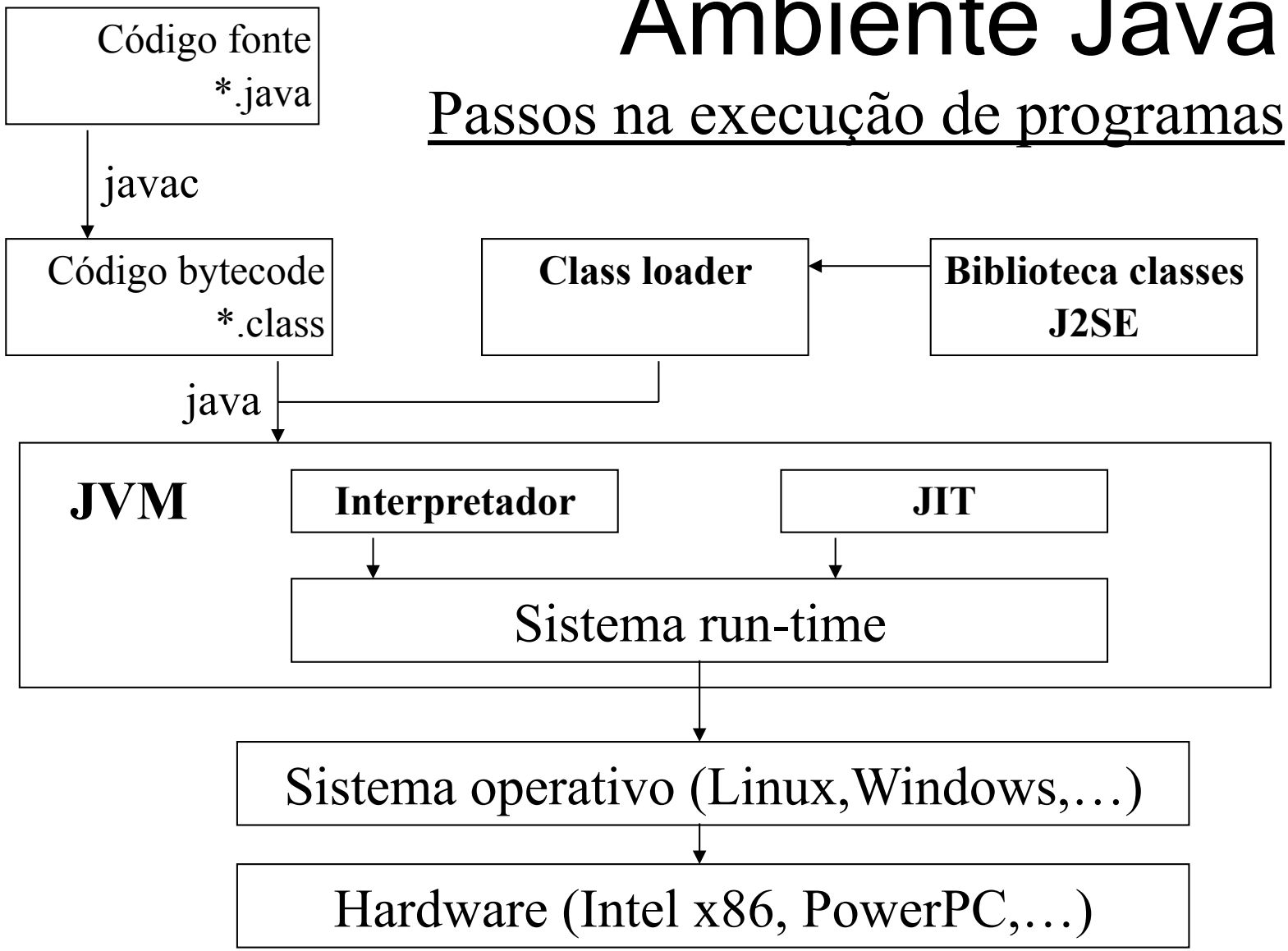
- **[2006] Versão 6.0**, denominada J2SE 6.0
  - Principais extensões: XML, web services, …
  - Nome de código *Mustang*

História – versões (4)

- A **versão J2SE 7** possui nome de código *Dolphin*, e está estimada para 2010.
  - JVM com suporte para linguagens de programação dinâmicas
  - Melhorias a nível do *garbage collector*
  - Principais extensões: computação paralela em processadores multi-core, super pacotes, ...

Ambiente Java

Passos na execução de programas



Plataforma Java (1)

- A tecnologia Java é distribuída para 3 plataformas:
  - J2EE (*Entreprise Edition*), para desenvolvimento de aplicações empresariais.
  - J2ME (*Micro Edition*), para dispositivos de capacidades limitadas (telemóveis e PDA's).
  - J2SE (*Standard Edition*), para *desktop* e servidores.

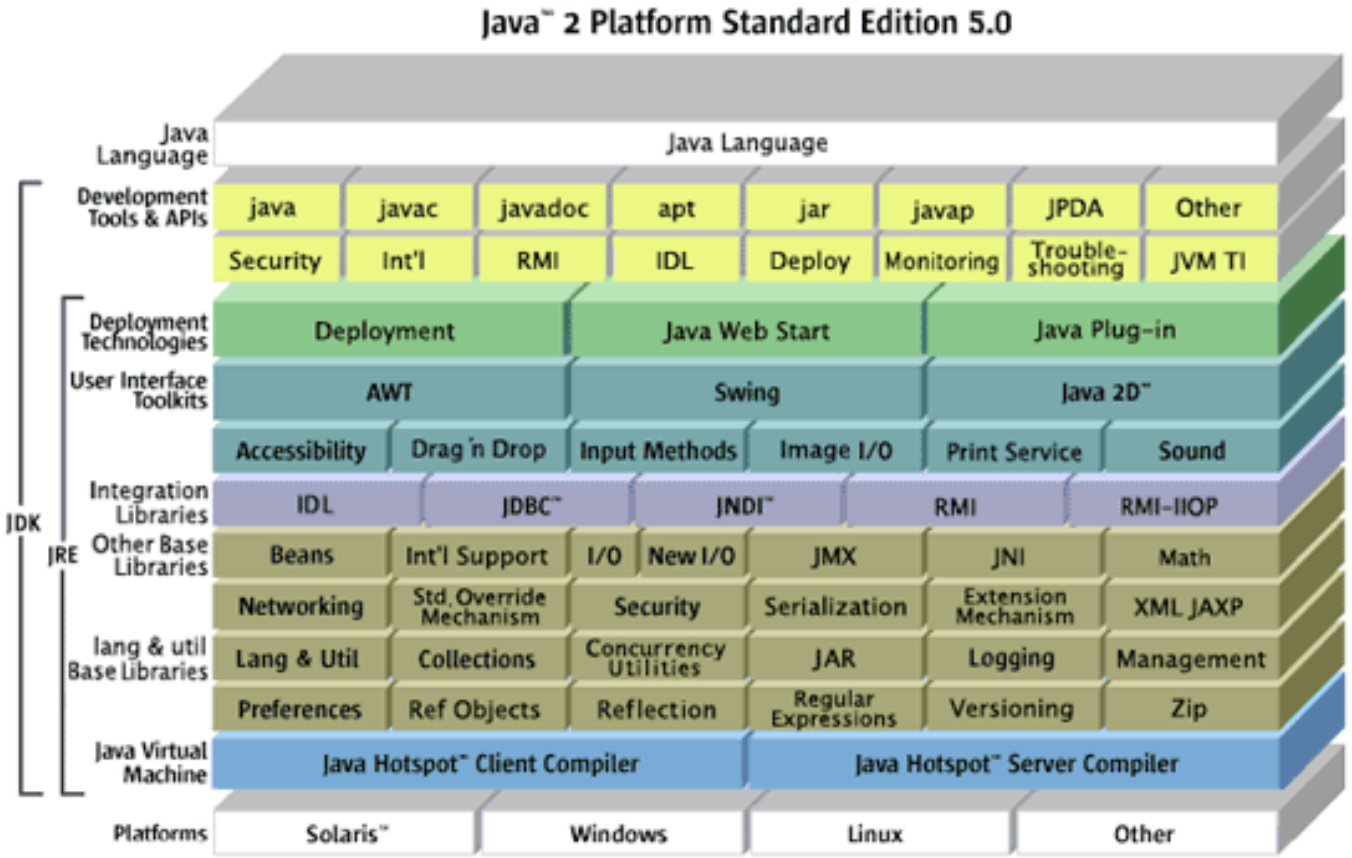
Plataforma Java (2)

- Debaixo da plataforma existem vários directórios:
  - J2xx Runtime Environment (JRE):
    - Interpretador JVM, classes de ambiente, …
    - Usado apenas para correr aplicações.
  - J2xx Development Kit (JDK):
    - JRE, compilador, classes utilitárias (Swing,…), …
    - Usado no desenvolvimento de aplicações.

Plataforma Java (3)

- A API do Java 2 consiste em diversas classes distribuídas e organizadas em pacotes e subpacotes.
- Pacotes básicos:
  - **java.lang**: classes de ambiente (importado automaticamente)
  - **java.util**: classes utilitárias (tipos de dados, etc)
  - **java.io**: classes para entrada e saída
  - java.net: classes para uso em rede (TCP/IP)
  - java.sql: classes para acesso via JDBC
  - java.awt: interface gráfica universal nativa
  - **javax.swing**: interface gráfica (mais leve que java.awt)
- Documentação:
  - http://java.sun.com/reference/api/index.html
  - J2SE 5.0: **http://java.sun.com/j2se/1.5.0/docs/api/**

Plataforma Java (4)



Linguagem Java

- Particularidades da linguagem, comparativamente com C/C++:
  - Não há apontadores, mas sim **referências**.
  - Há **`new`**, mas não há **`free`**/**`delete`** (***garbage collector***).
  - Os **parâmetros são passados por valor** aos métodos.
  - Não há *operator oveloading*.
  - Não há herança múltipla de classes, apenas de interfaces.
  - Não há *preprocessor* nem *header files*.
  - Não há variáveis globais.
  - Linguagem **fortemente tipada**.
  - Declaração de variáveis em qualquer local dentro do método.
  - Não há **`goto`**, **`typedef`**, **`union`**, **`struct`**, ou **`enum`**.
  - Pode haver mais do que um método **`main`** (mas apenas um por classe).

Java: referências (1)

- A linguagem Java não tem apontadores:
  - Há referências, na realidade **as referências são apontadores implícitos.**
  - **Não há aritmética de ponteiros**, os apontadores implícitos nunca são utilizados explicitamente como no C/C++.
  - Todos os objectos em Java encontram-se na *heap*.

Java: referências (2)

- Em Java os tipos primitivos (**`char`**, **`int`**, **`long`**, etc) são tratados de forma diferente dos objectos:
  - **Tipos primitivos**:

```
int iVar;
```

    - Variável inteira denominada **`iVar`**.
    - O valor da variável é de facto guardado num endereço de memória denominado **`iVar`**.
    - Antes de qualquer atribuição guarda um valor dado por omissão: 0.

Java: referências (3)

- **Objectos**:

```
ContaBancaria cbVar;
```

  - **`cbVar`** é uma **referência para um objecto** de tipo **`ContaBancaria`**.
  - O endereço de memória denominado **`cbVar`** não guarda o objecto, mas sim uma referência para um objecto desse tipo que é por sua vez guardado algures em memória.
  - Antes de qualquer atribuição guarda a referência para um objecto especial: **null**.

Java: operador **new**

- Todos os objectos em Java são criados com o operador **new**:

```
ContaBancaria cbVar1;
cbVar = new ContaBancaria();
```

  - O **new** retorna uma referência (não um apontador).
    - O programador não sabe qual o endereço onde o objecto se encontra.
  - Não é necessário libertar memória.
    - O Java verifica periodicamente cada bloco de memória obtido através de um **new** para verificar se ainda existe uma referência válida para ele (***garbage collector***).
    - Evita ***memory leaks***.

Java: atribuição

- Na atribuição de referências passam a existir duas referências distintas para o mesmo objecto:

```
ContaBancaria cbVar1, cbVar2;
cbVar1 = new ContaBancaria();
cbVar2 = cbVar1;
```

  - Ambas as variáveis referenciam exactamente o mesmo objecto.
  - Se de ambas as variáveis se fizer um levantamento de 1000€ então no final a conta bancária em questão terá menos 2000€ do que inicialmente.

Java: igualdade/equivalência (1)

- **Tipos primitivos**: **==**
  - Diz se as duas variáveis comparadas têm valores iguais, tal como em C/C++.

```
int iVar1 = 27;
int iVar2 = iVar1;
if (iVar1==iVar2)
   System.out.println("As variáveis são iguais!");
```

Java: igualdade/equivalência (2)

- **Objectos: igualdade** com **==**

```
ContaBancaria cbVar1 = new ContaBancaria();
ContaBancaria cbVar2 = cbVar1;
If (cbVar1==cbVar2)
   System.out.println("Os objectos são iguais!");
```

  - O operador == diz respeito à igualdade entre referências, i.e., se as referências referenciam o mesmo objecto (em C++ verifica se os dois objectos têm os mesmo dados).

Java: igualdade/equivalência (3)

- **Objectos: equivalência** com **`equals`**

```
ContaBancaria cbVar1 = new ContaBancaria();
ContaBancaria cbVar2 = new ContaBancaria();
If (cbVar1.equals(cbVar2))
 System.out.println("Os objectos são equivalentes!");
```

  - O método **`equals`** diz respeito à equivalência entre objectos, i.e., se os dois objectos têm os mesmo dados.
  - Por omissão o método **`equals`** devolve o mesmo que o operador ==, mas deve ser redefinido para devolver equivalência entre objectos sempre que tal for necessário.

Java: parâmetros

- Em Java, os parâmetros são sempre passados por valor.
  - O objecto nunca é copiado, existe apenas cópia da referência, referenciando ambas o mesmo objecto.

```
void metodo1() {
   ContaBancaria cbVar = new ContaBancaria();
   metodo2();
}
void metodo2(ContaBancaria cbArg) {}
```

  - As referências **`cbVar`** e **`cbArg`** referenciam o mesmo objecto.
- Em C/C++ os argumentos também são passados por valor, mas existe cópia integral do objecto. São utilizados ponteiros para fazer apenas cópia da referência, apontando ambas as referências para o mesmo objecto.
  - Em C++ **`cbArg`** seria um novo objecto, copiado de **`cbVar`**.

Java: entradas/saídas (1)

- Saídas:
  - Qualquer tipo primitivo (números e caracteres), assim como objectos do tipo **`String`**, pode ser impresso no terminal da seguinte forma:

```
System.out.print(var);
System.out.println(var);
```

    - O método **`print`** imprime o valor de **`var`**.
    - O método **`println`** imprime o valor de **`var`** e muda de linha.
    - Podem ainda ser utilizadas várias variáveis/literais separadas pelo operador **`+`**:

```
System.ou.println("A resposta é" + var);
```

      - O resultado seria: "A resposta é 15" (se o valor de **`var`** fosse 15).

Java: entradas/saídas (2)

- Entradas:
  - É necessário colocar no início do ficheiro fonte:

```
import.java.io.*;
```

  - É necessário adicionar **`throws IOException`** em todos os métodos que façam leitura do teclado e/ou de um ficheiro (inclusivé no método **`main`**).
  - Da entrada é lido um objecto do tipo **`String`**. Se se pretende qualquer outro tipo, p.e., um caracatere ou um número, tem de se converter o objecto do tipo **`String`** para o tipo desejado.

Java: entradas/saídas (3)

- **Leitura de uma `String` do teclado**:

```
public static String getString() throws IOException {
    InputStreamReader isr = new
            InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    String s = br.readLine();
    return s;
}
```

Java: entradas/saídas (4)

- **Leitura de um `char` do teclado**:

```
public static char getChar() throws IOException {
    String s = getString();
    return s.charAt(0);
}
```

  - O método **`charAt`** retorna o caractere pedido do objecto **`String`**.
  - Neste exemplo é pedido o primeiro caractere (no índice 0).
  - A leitura da cadeia de caracteres, e o retorno de apenas o primeiro, evita que caracteres lidos fiquem no tampão de entrada (que poderiam causar problemas em leituras posteriores).

Java: entradas/saídas (5)

- **Leitura de um `int`/`long` do teclado**:

```
public static int getInt() throws IOException {
    String s = getString();
    return Integer.parseInt(s);
}

public static long getLong() throws IOException {
    String s = getString();
    return Long.parseLong(s);
}
```

  - O método **`parseInt`**/**`parseLong`**, da classe **`Integer`**/**`Long`**, converte um objecto **`String`** num **`int`**/**`long`**.

Java: entradas/saídas (6)

- **Leitura de um `double`/`float` do teclado**:

```
public static int getDouble() throws IOException {
    String s = getString();
    Double d = Double.valueOf(s);
    return d.doubleValue();
}
public static float getFloat() throws IOException {
    String s = getString();
    Float f = Float.valueOf(s);
    return f.floatValue();
}
```

  - O método **`ValueOf`**, da classe **`Double`**/**`Float`**, converte um objecto **`String`** num objecto **`Double`**/**`Float`**.
  - O método **`doubleValue`**/**`floatValue`**, da classe **`Double`**/**`Float`**, converte um objecto **`Double`**/**`Float`** num objecto **`double`**/**`float`**.

Java: main

- Todas as classe numa aplicação podem ter um método **`main`**.

```
public class HelloWorld {
    public static void main(String[] args) {
            System.out.println("Hello, world!");
    }
}
```

- Cada ficheiro tem apenas uma classe publica, e o nome do ficheiro deve ser exactamente o nome da classe com extensão .java.
- O interpretador *Java virtual machine* (JVM) executa sempre o **método main** da classe indicada na linha do comando.

```
> javac HelloWorld.java
> java HelloWorld
> Hello world!
```