

Programando em Java

(Classes Simples e Tipos de Dados Básicos)

**Material didáctico elaborado pelas diferentes equipas de
Introdução à Programação**

Luís Caires (Responsável), Armanda Rodrigues, António Ravara, Carla Ferreira, Fernanda Barbosa, Fernando Birra, Jácome Cunha, João Araújo, Miguel Goulão, Miguel Pessoa Monteiro, e Sofia Cavaco.

Mestrado Integrado em Engenharia Informática FCT UNL

Alguns Programas Simples

- Neste capítulo, vamos programar em Java um conjunto de classes simples.



- No caminho, serão introduzidas várias noções novas e importantes:
 - Operações com valores inteiros e lógicos
 - Parâmetros de construtores e parâmetros de métodos
 - Definição nas classes de múltiplos construtores

Conta Bancária

Conta Bancária

- **Objectivo**
 - Simular uma conta bancária.
- **Descrição**
 - Uma conta bancária é um “depósito” de dinheiro (valor inteiro em cêntimos). A quantidade de dinheiro na conta chama-se “saldo”. O saldo pode ser positivo (credor ou nulo) ou negativo (devedor), e é sempre um valor inteiro em cêntimos.
- **Funcionalidades**
 - Numa conta pode-se depositar e levantar dinheiro.
 - Deve ser sempre possível consultar o saldo da conta, e verificar se a conta tem um saldo devedor.
 - Se não indicarmos nada, a conta é criada com saldo zero. Em alternativa, podemos indicar um valor inicial para o saldo.
- **Interacção com o utilizador**
 - Após criar uma conta bancária, pode invocar as operações da conta.

Conta Bancária

- **Que objecto se deve definir?**
 - Uma conta bancária (classe BankAccount) que guarda o saldo em centimos
- **Interface:**
 - void** deposit(**int** amount)
Depositar a importância amount na conta
 - void** withdraw(**int** amount)
Levantar a importância amount na conta
 - int** getBalance()
Consultar o saldo da conta
 - boolean** redZone()
Indica se a conta está devedora

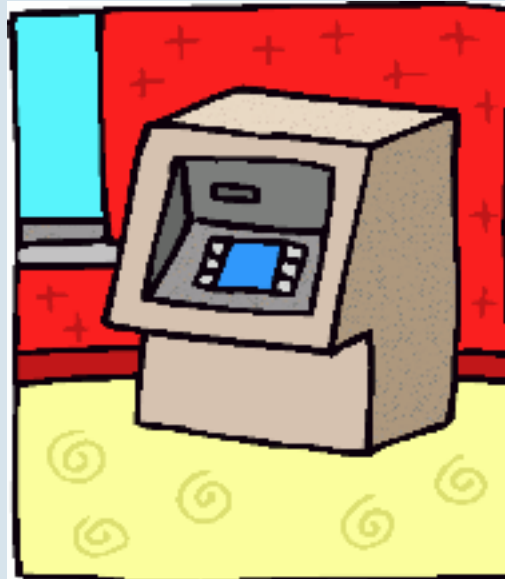
Cenário

Comportamento da Conta Bancária

```
BankAccount b1 = new BankAccount(2000);  
b1.getBalance()  
2000 (int)  
b1.deposit(2);  
b1.deposit(8);  
b1.redZone()  
false (boolean)  
b1.getBalance()  
2010 (int)  
b1.withdraw(3000);  
b1.getBalance()  
-990 (int)  
b1.redZone()  
true (boolean)
```


Conta Bancária

- Defina em Java uma classe BankAccount cujos objectos têm a funcionalidade indicada.
- Programe a sua classe no BlueJ.
- Teste um (ou vários) objectos BankAccount, e verifique se se comportam como esperado.



Programando a Conta Bancária

```
public class BankAccount {  
    private int balance ;  
    public BankAccount() { .... }  
    public BankAccount(int initial) { .... }  
    public void deposit(int amount) { .... }  
    public void withdraw(int amount) { .... }  
    public int getBalance() { .... }  
    public boolean redZone() { .... }  
}
```



Nome da classe

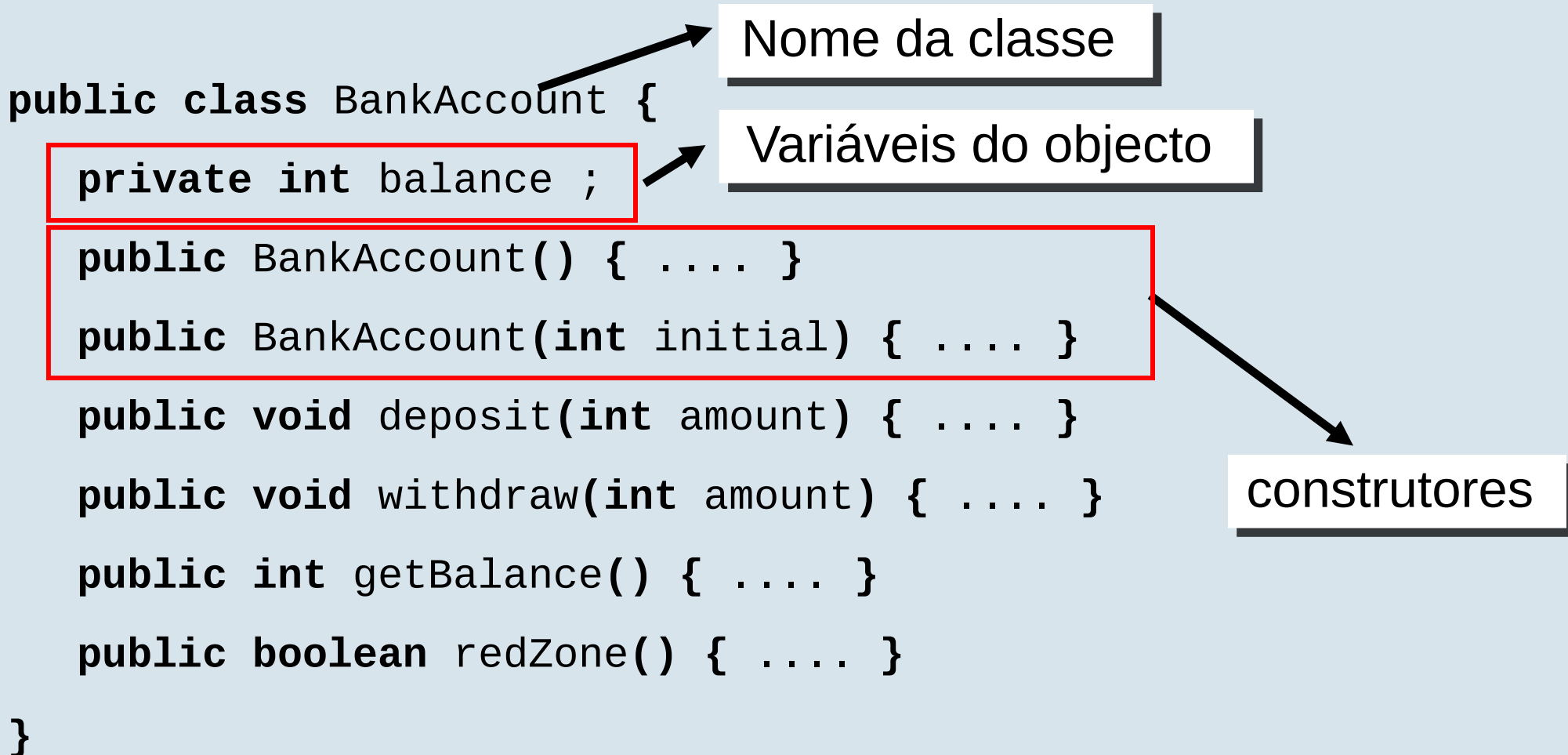
Programando a Conta Bancária

```
public class BankAccount {  
    private int balance ;  
    public BankAccount() { .... }  
    public BankAccount(int initial) { .... }  
    public void deposit(int amount) { .... }  
    public void withdraw(int amount) { .... }  
    public int getBalance() { .... }  
    public boolean redZone() { .... }  
}
```

Nome da classe

Variáveis do objecto

construtores



Programando a Conta Bancária

```
public class BankAccount {  
    private int balance ;  
    public BankAccount() { .... }  
    public BankAccount(int initial) { .... }  
    public void deposit(int amount) { .... }  
    public void withdraw(int amount) { .... }  
    public int getBalance() { .... }  
    public boolean redZone() { .... }  
}
```



métodos

Múltiplos Construtores

```
public class BankAccount {
```

```
    private int balance ;
```

```
    public BankAccount() { balance = 0; }
```

```
    public BankAccount(int initial) { ... }
```

Note que definimos 2 construtores diferentes.
Isto permite criar novos objectos de duas formas diferentes. Por exemplo:

```
    public void main() {  
        new BankAccount() // Invocação cria objecto com saldo 0
```

```
    }
```

```
    public boolean redZone() { .... }
```

```
}
```

saldo inicial

Múltiplos Construtores

Parâmetro: informação de entrada (como numa função $f(x) = \dots$)

```
public class BankAccount {  
    private int balance ;  
    public BankAccount() { balance = 0; }  
    public BankAccount(int initial) { balance = initial ; }  
    public boolean redzone() { ..... }  
}
```

Note que definimos 2 construtores diferentes.

Isto permite criar novos objectos de duas formas diferentes. Por exemplo:

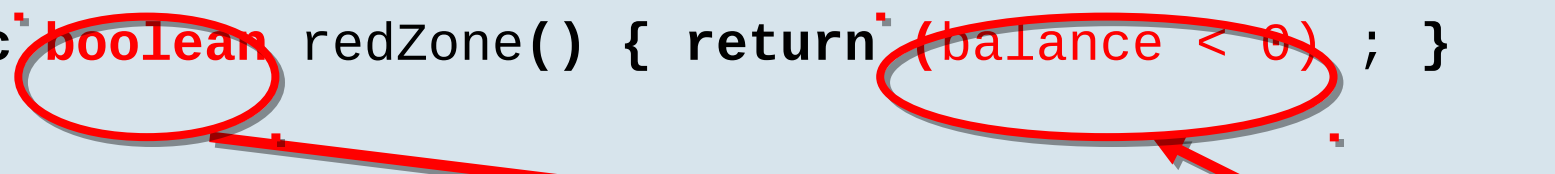
`new BankAccount()` // Invocação cria objecto com saldo 0

`new BankAccount(20)` // Invocação cria objecto com saldo 20

Argumento: valor passado para substituir parâmetro

Método redZone

```
public class BankAccount {  
    private int balance ;  
    public BankAccount() { balance = 0; }  
    public BankAccount(int initial) { balance = initial ;}  
    public void deposit(int amount) { .... }  
    public void withdraw(int amount) { .... }  
    public int getBalance() { .... }  
    public boolean redZone() { return (balance < 0) ; }  
}
```




expressão booleana

Método getBalance

```
public class BankAccount {  
    private int balance ;  
    public BankAccount() { balance = 0; }  
    public BankAccount(int initial) { balance = initial;}  
    public void deposit(int amount) { .... }  
    public void withdraw(int amount) { .... }  
    public int getBalance() { return balance; }  
    public boolean redZone() { return(balance < 0); }  
}
```

Métodos com Parâmetros

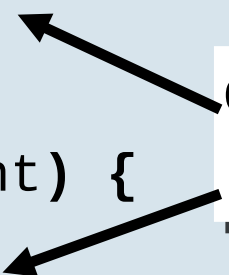
```
public class BankAccount {  
    private int balance ;  
    public BankAccount() { balance = 0; }  
    public BankAccount(int initial) { balance = initial ; }  
    public void deposit(int amount) {  
        ...  
    }  
    public void withdraw(int amount) {  
        ...  
    }  
    public int getBalance() { return balance; }  
    public boolean redZone() { return (balance < 0); }  
}
```



Parâmetro do método

Métodos deposit e withdraw

```
public class BankAccount {  
    private int balance ;  
    public BankAccount() { balance = 0; }  
    public BankAccount(int initial) { balance = initial ; }  
    public void deposit(int amount) {  
        balance = balance + amount;  
    }  
    public void withdraw(int amount) {  
        balance = balance - amount;  
    }  
    public int getBalance() { return balance; }  
    public boolean redZone() { return (balance < 0); }  
}
```



expressões inteiras
(calculam um valor int)

Métodos deposit e withdraw

```
public class BankAccount {
```

Faz sentido o valor amount não ser positivo?



```
...
```

```
public void deposit(int amount) {
```

```
    balance = balance + amount;
```

```
}
```

```
public void withdraw(int amount) {
```

```
    balance = balance - amount;
```

```
}
```

```
...
```

```
}
```

Conta Bancária

Interface

- Enriquece-se a interface com **contrato de utilização**.

- **Interface:**

void deposit(**int** amount)

Depositar a importância amount na conta

Pre: amount > 0

void withdraw(**int** amount)

Levantar a importância amount na conta

Pre: amount > 0

int getBalance()

Consultar o saldo da conta

boolean redZone()

Indica se a conta está devedora

O utilizador deve respeitar o contrato (as pré-condições dos métodos).

➤ E se não respeitar?

Métodos com Parâmetros

- Já conhecemos a forma geral dos métodos mais simples sem parâmetros

```
acesso tipo identificadorMétodo( ) {  
    corpo do método  
}
```

- Para fornecermos informação de entrada adicional a uma operação de um objecto, podemos introduzir parâmetros nos métodos

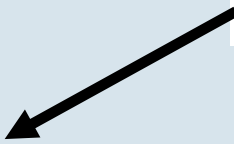
```
acesso tipo identificadorMétodo( tipo parâmetro, tipo parâmetro, ... ) {  
    corpo do método  
}
```

Métodos com Parâmetros

- Os parâmetros de cada método são declarados entre os () logo após o nome do método

acesso tipo identificadorMétodo (tipo parâmetro, tipo parâmetro, ...) { corpo }

Declaração de parâmetro



```
public void withdraw(int amount) {  
    balance = balance - amount;  
}
```

Métodos com Parâmetros

- Os parâmetros de cada método são declarados entre os () logo após o nome do método

acesso tipo identificadorMétodo (tipo parâmetro, tipo parâmetro, ...) { corpo }

Nome do parâmetro: deve ser um identificador permitido em Java

```
public void withdraw(int amount) {  
    balance = balance - amount;  
}
```

Uso do parâmetro

Métodos com Parâmetros

```
BankAccount b1 = new BankAccount(2000);
```

```
b1.getBalance()
```

```
2000 (int)
```

```
b1.deposit(2);
```

```
b1.deposit(8);
```

```
b1.redZone()
```

```
false (boolean)
```

```
b1.getBalance()
```

```
2010 (int)
```

```
b1.withdraw(3000);
```

```
b1.getBalance()
```

```
-990 (int)
```

```
b1.redZone()
```

```
true (boolean)
```

```
public void withdraw(int amount) {  
    balance = balance - amount;  
}
```

Argumento da chamada

Quando o corpo do método `withdraw` for executado, o parâmetro `amount` vai conter o valor inteiro 3000

Tipo int e Operações Associadas

- Operações que produzem um valor inteiro (`int`) a partir de valores inteiros

const constante – uma constante é uma expressão

var variável – uma variável é uma expressão

expr + expr adição

expr - expr subtracção

*expr * expr* multiplicação

expr / expr divisão inteira

expr % expr módulo (resto da divisão inteira)

var ++ devolve o valor de *var* e incrementa *var* (depois)

var -- devolve o valor de *var* e decrementa *var* (depois)

Tipos float e double e Operações Associadas

- A linguagem Java oferece dois tipos de dados para representar valores reais
 - O tipo **float** (números de vírgula flutuante de precisão simples, com 32 bits)
 - O tipo **double** (números de vírgula flutuante de precisão dupla, com 64 bits)
- Constantes de tipo **float** (números de vírgula flutuante de precisão simples)
 - Para indicar constantes de tipo **float** deve usar o sufixo **f**
 - **1.2f**
 - **200.23f**
- Constantes de tipo **double** (números de vírgula flutuante de precisão dupla)
 - Para indicar constantes de tipo **double** pode usar o ponto decimal
 - **3.14**
 - **289822.23**

Tipos float e double e Operações Associadas

- Operações que produzem um valor real (de tipo `float` ou `double`) a partir de valores reais (de tipo `float` ou `double`)

<i>const</i>	constante
<i>var</i>	variável
<i>expr1</i> + <i>expr2</i>	adição
<i>expr1</i> - <i>expr2</i>	subtracção
<i>expr1</i> * <i>expr2</i>	multiplicação
<i>expr1</i> / <i>expr2</i>	divisão

Quando realizamos operações entre números reais, o resultado tem o tipo da expressão que tiver maior precisão. Por exemplo, somar um `float` com um `double` tem como resultado um `double`.

O mesmo se aplica a operações com inteiros e reais: o resultado é real. Por exemplo, multiplicar um inteiro por um `float` produz um `float` como resultado.

Tipos float e double e Operações Associadas

- A linguagem Java é complementada por um vasto conjunto de bibliotecas que permitem “aumentar” as capacidades da linguagem mantendo-a relativamente simples
- A biblioteca `Math`, disponível no ambiente Java, oferece um conjunto bastante completo de constantes e operações para números reais:
 - Constantes úteis:
 - e
 - π
 - Operações
 - Exponencial
 - Logaritmo
 - Raiz quadrada
 - Funções trigonométricas
 - e muito, muito mais...

Tipos float e double e Operações Associadas

- Constantes úteis
 - `Math.PI` π
 - `Math.E` e
- Operações úteis sobre valores de tipo `double`
 - `Math.round(expr)` arredonda o valor de `expr` ao inteiro mais próximo
 - `Math.sin(expr)` seno (argumento em radianos)
 - `Math.cos(expr)` coseno (argumento em radianos)
 - `Math.sqrt(expr)` raiz quadrada
 - `Math.log(expr)` logaritmo de base e
- Para obter a lista completa consulte a página da disciplina no clip, na zona da bibliografia.

Tipo boolean e Operações Associadas

- Operações que produzem um valor booleano (**true** ou **false**) a partir de valores escalares (de tipo **int**, **float** ou **double**)
- Para já, vamos assumir que *expr1* e *expr2* têm que ser **int**, **float** ou **double**)

expr1 > *expr2* maior que

expr1 >= *expr2* maior ou igual

expr1 < *expr2* menor

expr1 <= *expr2* menor ou igual

expr1 == *expr2* igualdade

expr1 != *expr2* desigualdade

- **Importante:** Estes operadores chamam-se **operadores relacionais**.

Tipo boolean e Operações Associadas

- Operações que produzem um valor booleano (de tipo `boolean`) a partir de valores booleanos (de tipo `boolean`)

- Aqui, vamos assumir que *expr1* e *expr2* têm que ser `boolean`

expr1 `&&` *expr2* conjunção lógica

expr1 `||` *expr2* disjunção lógica

`!` *expr* negação lógica

`true` valor lógico “verdade”


`false` valor lógico “falso”

- **Importante:** Estes operadores chamam-se **operadores lógicos**.

Ordem de Precedência dos Operadores

- Para desambiguar a ordem de aplicação dos operadores em expressões compostas, podem ser usados os parêntesis (e).
 - Por exemplo, $(2+x)*2$ em vez $2+x*2$ (que é interpretado como $2+(x*2)$)
- Ordem de precedência (a começar nos operadores mais fortes)

++	--	maior prisão (maior precedência)	
!			
*	/	%	
+	-		
<	<=	>	>=
==	!=		
&&			
			menor prisão (menor precedência)



Exemplo: a expressão

`(x++*2>4)&& false || true`

é interpretada como

`((((x++)*2)>4)&& false) || true`