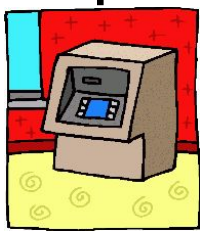
Programando em Java
(Classes Simples e Tipos de Dados Básicos)

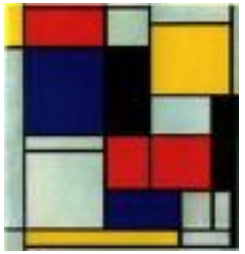
Mestrado Integrado em Engenharia Informática FCT UNL
http://ctp.di.fct.unl.pt/miei/ip/

Corpo Docente 2020/2021
António Ravara, Artur Miguel Dias, Bernardo Toninho,
Ema Vieira, Inês Fernandes, Margarida Mamede,
Miguel Monteiro, Rui Nóbrega

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, reais e lógicos
  - Parâmetros de construtores e parâmetros de métodos
  - Definição de (nomes para) constantes
  - Definição, nas classes, de múltiplos construtores
  - Chamada de outros métodos dentro do método de um objecto

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 cêntimos
- **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 isInRedZone()
```

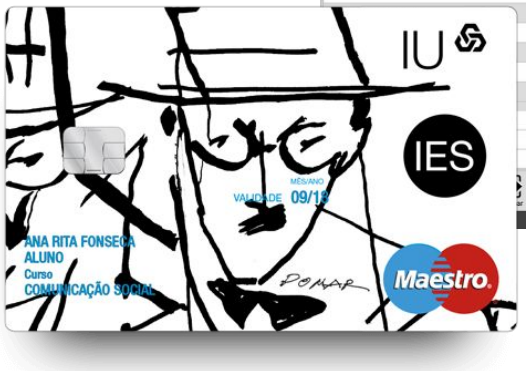
Indica se a conta está devedora

Cenário
Comportamento da Conta Bancária

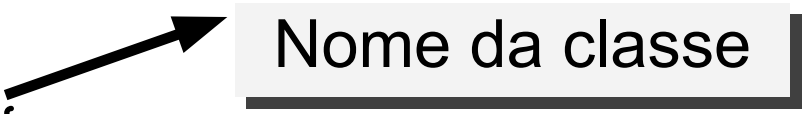
```
BankAccount b1 = new BankAccount(2000);
System.out.println(b1.getBalance());
2000
b1.deposit(2);
b1.deposit(8);
System.out.println(b1.isInRedZone());
false
System.out.println(b1.getBalance());
2010
b1.withdraw(3000);
System.out.println(b1.getBalance());
-990
System.out.println(b1.isInRedZone());
true
```

Conta Bancária

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

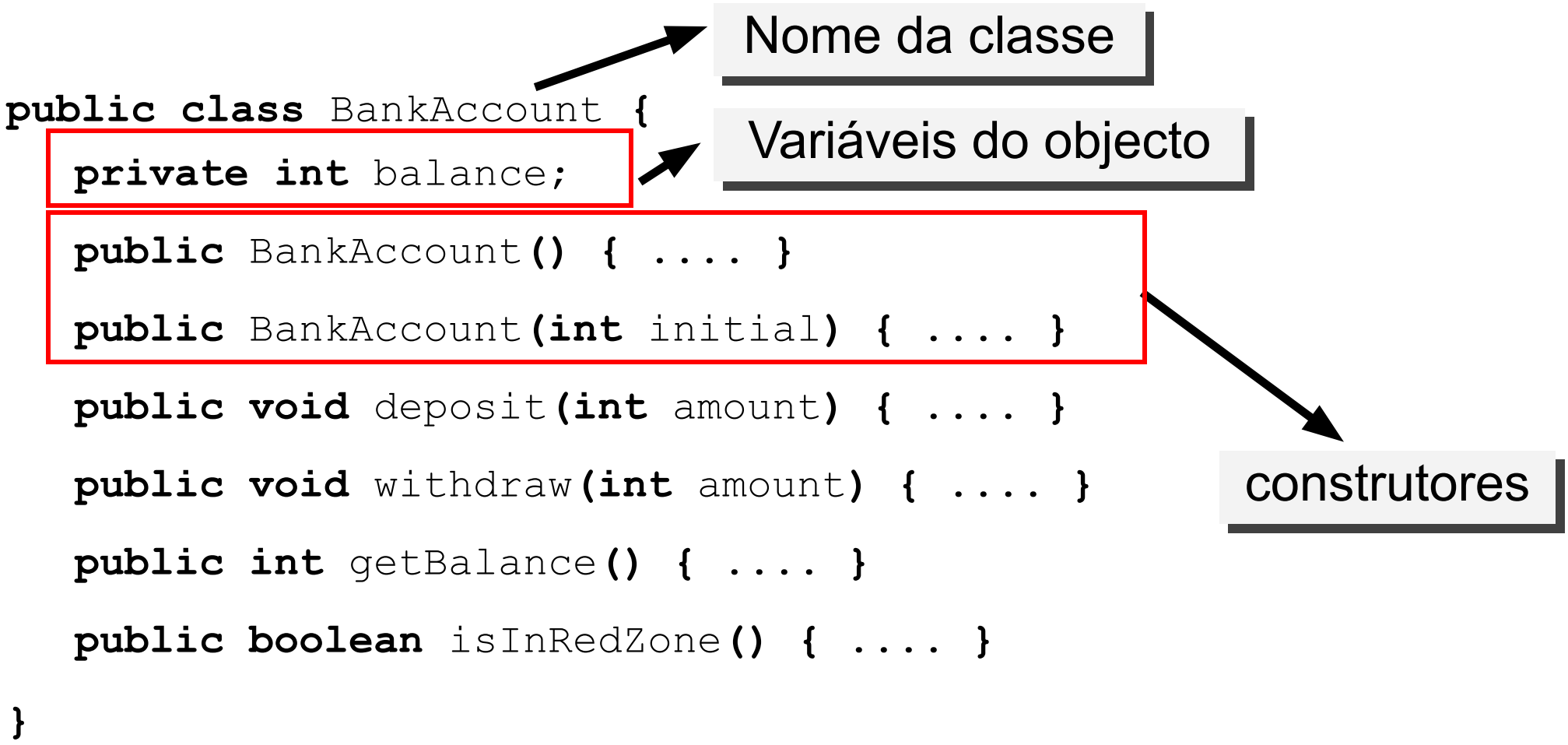


Programando a Conta Bancária




```
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 isInRedZone() { .... }
}
```

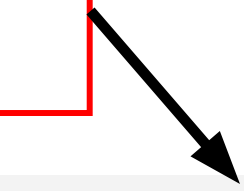
Programando a Conta Bancária




```
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 isInRedZone() { .... }
}
```

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 isInRedZone() { .... }
}
```



métodos

Múltiplos Construtores

```
public class BankAccount {
   private int balance;

   public BankAccount() { balance = 0; }

   public BankAccount(int initial) { … }

   public v
   public v
   public i
   public boolean isInRedZone() { .... }
}
```

saldo inicial

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
```

Múltiplos Construtores

Parâmetro: informação de entrada (como numa função f(x) = …)

saldo inicial

```
public class BankAccount {
    private int balance;

    public BankAccount() { balance = 0; }

    public BankAccount(int initial) { balance = initial; }

    public void deposit(int amount) { ... }

    public
    public
    public
}
```

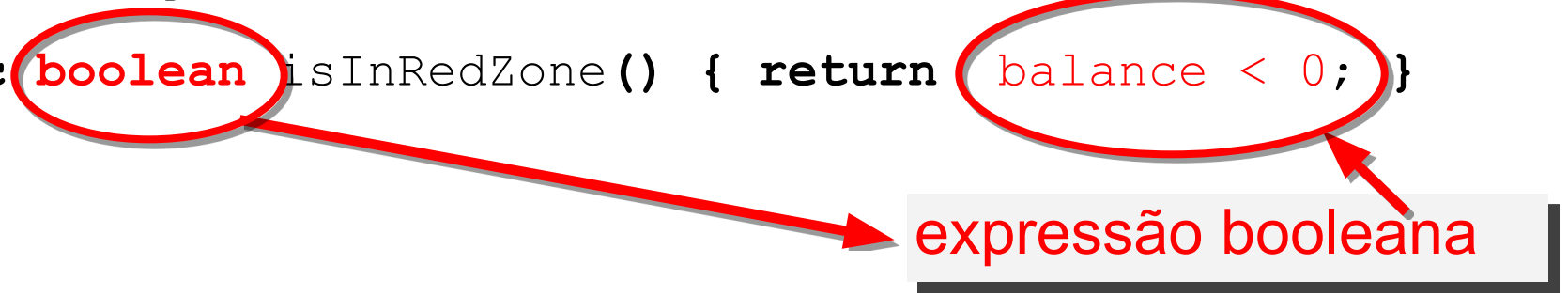
Note que definimos **dois** construtores diferentes.
Isto permite criar novos objectos de duas formas diferentes. Por exemplo:

```
new BankAccount()      // Invocação cria objecto com saldo 0
new BankAccount(20)    // Invocação cria objecto com saldo 20
```

Argumento: valor passado para substituir parâmetro

Método `isInRedZone`

```
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 isInRedZone() { return balance < 0; }
}
```



Método `getBalance`

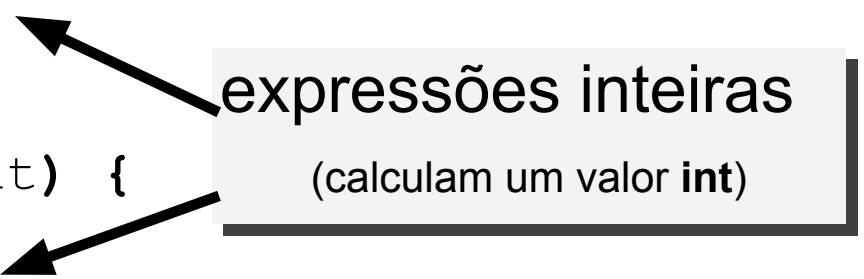
```
public class BankAccount {

   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 isInRedZone() { 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 isInRedZone() { return balance < 0; }
}
```

Métodos `deposit` e `withdraw`

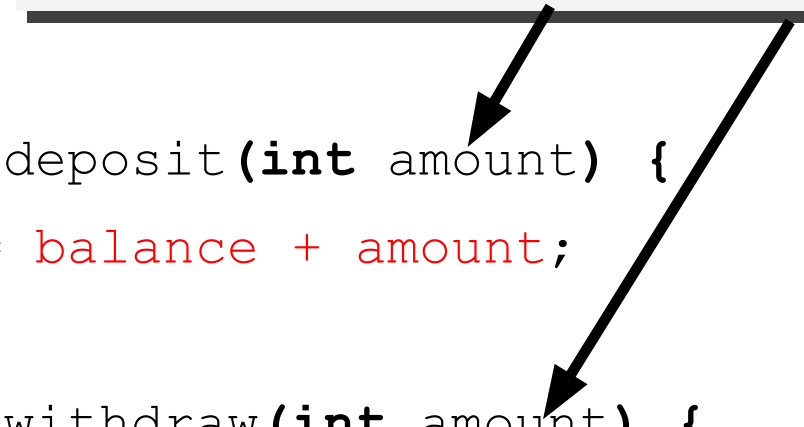
```
public class BankAccount {
   private int balance;
   public BankAccount() { balance = 0; }
   public BankAccount(int initial) { balance = initial; }
   public void deposit(int amount) {
      balance = balance + amount;
   }
   public void withdraw(int amount) {
      balance = balance - amount;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



Métodos `deposit` e `withdraw`

```
public class BankAccount {

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

Faz sentido o valor `amount` não ser positivo?



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 isInRedZone()
```

Indica se a conta está devedora

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

➢E se não respeitar?

Métodos `deposit` e `withdraw`

```
public class BankAccount {
   private int balance;
   public BankAccount() { balance = 0; }
   public BankAccount(int initial) { balance = initial; }
   /* Pre: amount > 0 */
   public void deposit(int amount) {
      balance = balance + amount;
   }
   /* Pre: amount > 0 */
   public void withdraw(int amount) {
      balance = balance - amount;
   }
   public int getBalance() { return balance; }
   public boolean isInRedZone() { return balance < 0; }
}
```

Pré-condições

- A execução correta do método é garantida apenas se `amount > 0`
- São inseridas na classe como comentário, para avisar o programador que deve considerá-las no resto do código

Métodos com Parâmetros

```
BankAccount b1 = new BankAccount(2000);
System.out.println(b1.getBalance());
2000
b1.deposit(2);
b1.deposit(8);
System.out.println(b1.isInR
false
System.out.println(b1.getBalance());
2010
System.out.println(b1.withdraw(3000);
b1.getBalance());
-990
System.out.println(b1.isInRedZone());
true
```

```
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

Métodos com Parâmetros

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

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

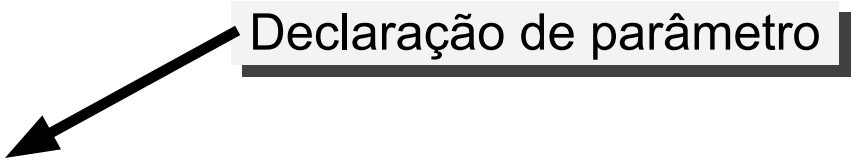
- Para fornecermos informação de entrada adicional a uma operação de um objecto, podemos introduzir parâmetros nos métodos

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

Métodos com Parâmetros

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

acesso tipo identificadorMétodo **(** ***tipo*** *idParâmetro*, ***tipo*** *idParâmetro*, ... **)** { ***corpo*** }




```
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*** *idParâmetro*, ***tipo*** *idParâmetro*, ... **)** { ***corpo*** }

Nome do parâmetro: deve ser um identificador permitido em Java

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

Uso do parâmetro