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

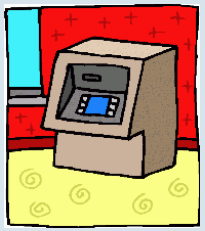
Material didáctico elaborado pelas diferentes equipas de Introdução à Programação

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, 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 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

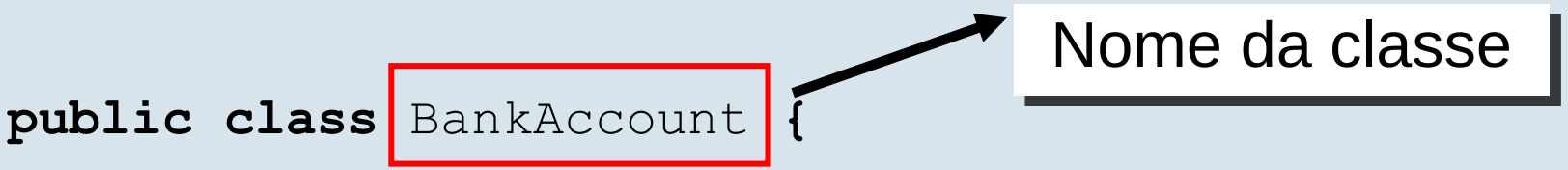
Cenário
Comportamento da Conta Bancária

```
 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

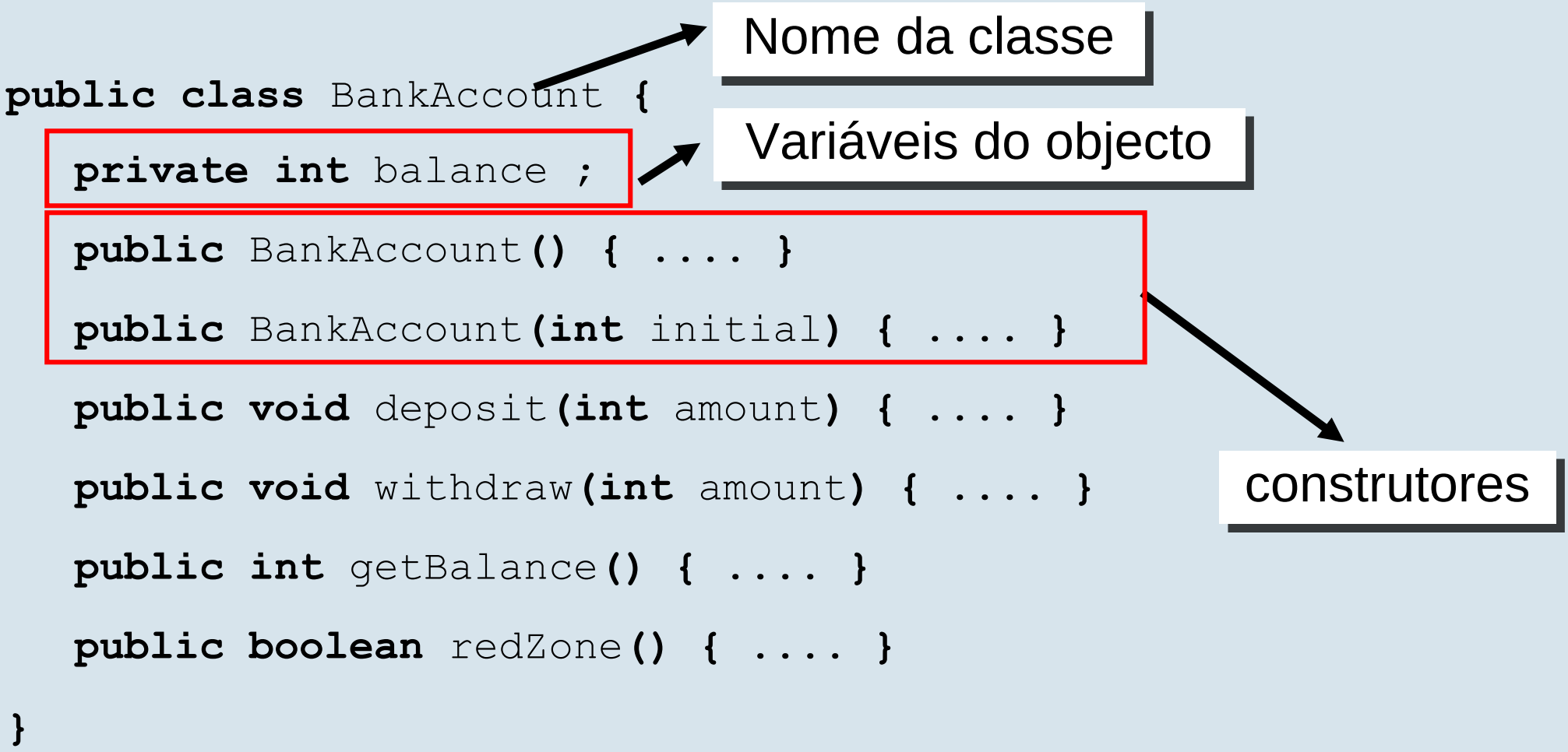
- Defina em Java uma classe BankAccount cujos objectos têm a funcionalidade indicada.
- Programe a sua classe no Eclipse.
- Teste um (ou vários) objectos BankAccount, e verifique se se comportam como esperado.

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 redZone() { .... }
}
```

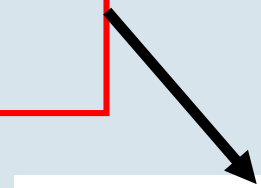
Programando a Conta Bancária




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

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



Múltiplos Construtores

```
public class BankAccount {
    private int balance ;
    public BankAccount() { balance = 0; }
    public BankAccount(int initial) { … }
    public v
    public v
    public i
    public boolean redZone() { .... }
}
```

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 void withdraw(int amount) { ... }

   public int getBalance() { ... }

   public boolean redZone() { ... }

}
```

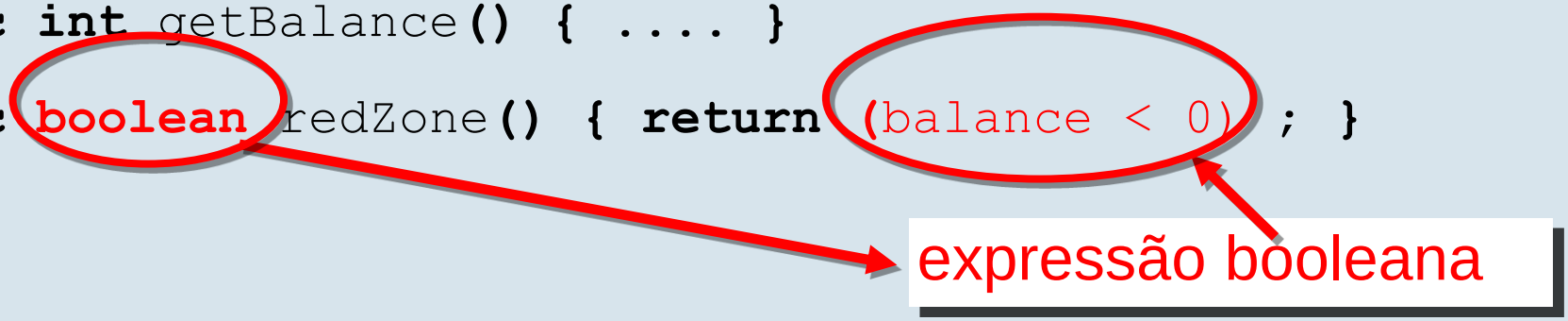
Note que definimos **dois** construtores diferentes.
Isto permite criar novos objectos de duas formas diferentes. Por exemplo:

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

Argumento: valor passado para substituir parâmetro

Método `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) ; }

}
```



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); }
}
```

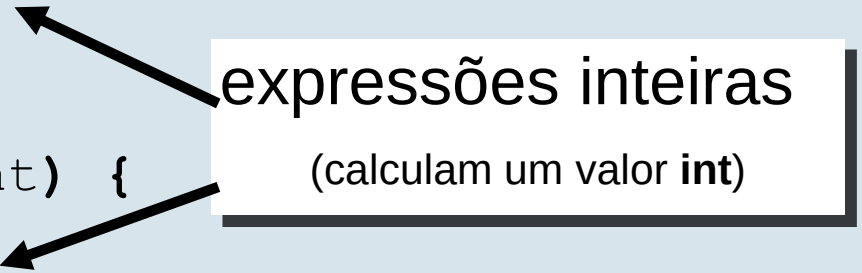
Métodos com Parâmetros

```
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); }
}
```



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 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
}
```

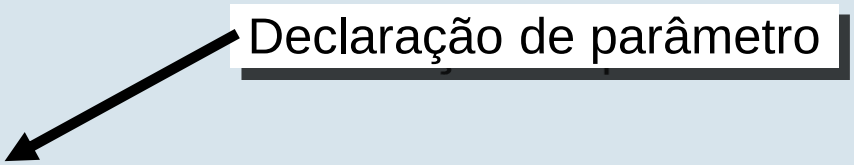
- Para fornecermos informação de entrada adicional a uma operação de um objecto, podemos introduzir parâmetros nos métodos

```
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*** }




```
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 (quociente)
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`**)

const	constante
var	variável
expr1 **`+`** *expr2*	adição
expr1 **`-`** *expr2*	subtracção
expr1 **`*`** *expr2*	multiplicação
expr1 **`/`** *expr2*	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*
    - *pi*
  - 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
```

Rectângulo

Rectângulo

- **Objectivo**
  - Manipular rectângulos no plano XY.
- **Descrição**
  - Qualquer rectângulo está no plano XY, e tem os seus lados paralelos aos eixos. As coordenadas que o caracterizam devem ser valores reais de precisão muito grande.
- **Funcionalidades**
  - Pode-se deslocar o rectângulo ao longo de um dado vector <dx,dy>, e rodar o rectângulo 90º para a direita.
  - Deve ser sempre possível consultar a abcissa dos cantos esquerdos e direitos e a ordenada dos cantos superiores e inferiores do rectângulo. Para além disso, deve-se consultar também a altura, a largura, o perímetro, a área, e a ordenada e abcissa do centro do rectângulo.
  - Deve ser sempre possível verificar se um dado ponto ou um dado rectângulo no plano XY está no interior do rectângulo.

  …

Rectângulo

- Se não indicarmos nada, o rectâgulo será um quadrado de lado 1, com centro na origem do plano XY. Em alternativa, pode-se:
  - Indicar:
    - A abcissa do canto superior esquerdo (left)
    - A ordenada do canto superior esquerdo (top)
    - A abcissa do canto inferior direito (right)
    - A ordenada do canto inferior direito (bottom)
  - Indicar:
    - A abcissa do canto superior esquerdo
    - A ordenada do canto superior esquerdo
    - A dimensão do lado (length) – neste caso o rectângulo é um quadrado!

- **Interacção com o utilizador**
  - Após criar rectângulos, pode invocar as operações.

Rectângulo

- **Que objecto definir?**
  - Um rectângulo (classe Rect)
- **Interface:**

As coordenadas do rectângulo devem ser valores reais de precisão muito grande. → tipo `double`

`double` `getLeft()`

consulta a abcissa dos cantos esquerdos

`double` `getRight()`

consulta a abcissa dos cantos direitos

`double` `getTop()`

consulta a ordenada dos cantos superiores

`double` `getBottom()`

consulta a ordenada dos cantos inferiores

Rectângulo

double `getXCenter()`

consulta a abcissa do centro do rectângulo

double `getYCenter()`

consulta a ordenada do centro do rectângulo

double `getWidth()`

consulta (calcula) a largura do rectângulo

double `getHeight()`

consulta a altura do rectângulo

double `getPerimeter()`

consulta o perímetro do rectângulo

double `getArea()`

consulta a área do rectângulo

Rectângulo

boolean `ptInRect(double x, double y)`

verifica se o ponto (`x`,`y`) está dentro do rectângulo

O parâmetro do método é um objecto rectângulo.

boolean `rectInRect(Rect r)`

verifica se o rectângulo `r` está dentro do rectângulo

void `translate(double dx, double dy)`

desloca o rectângulo ao longo do vector <`dx`,`dy`>

void `turn()`

roda o rectângulo 90º para a direita

Múltiplos Construtores

```
public class Rect {
    ... ; /* Definição de constantes e variáveis de instância */
    /* Um quadrado de lado 1, com centro na origem do plano */
    public Rect() { ...; }
    /* Pre: (left < right) && (top > bottom) */
    public Rect(double left, double top, double right, double bottom)
    { ... ; }
    /* Pre: length > 0 */
    public Rect(double left, double top, double length)
    { ... ; }
    ...  /* Definição dos restantes métodos da classe */
}
```

Múltiplos Construtores

```
public class Rect {

    ... ;

    public Rect() { ...; }

    public Rect(double left, double top, double right, double bottom)

    { ... ; }

    public Rect(double left, double top, double length)

    { ... ; }

    ...

}
```

- É possível definir na mesma classe quantos construtores quisermos, para cobrir as várias maneiras pretendidas de criar os objectos.
- Todos os construtores têm o mesmo nome (o nome da classe) mas são distinguidos pelos parâmetros que possuem

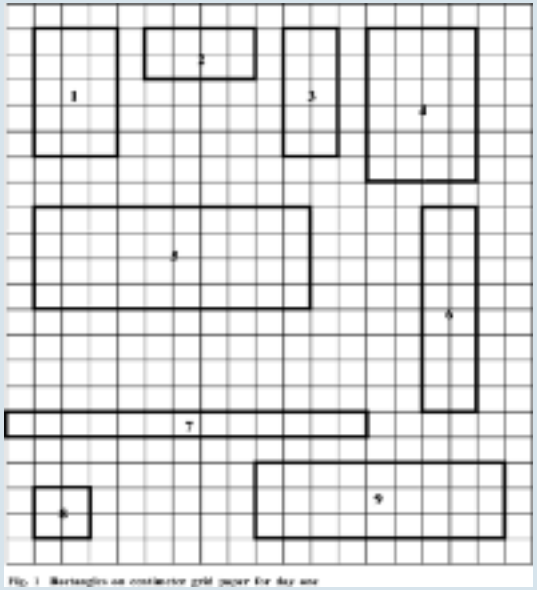
Rectângulo

```
Rect r = new Rect(10,50,120,20);
Rect r1 = new Rect();
r.getArea();
// 3300.0   (double)
r.getPerimeter();
// 280.0   (double)
r.ptInRect(30,40);
// true   (boolean)
r.ptInRect(130,40);
// false   (boolean)
r.translate(10,-30);
r.ptInRect(130,40);
// false   (boolean)
r.rectInRect(r1);
// false            (boolean)
```

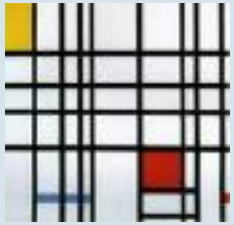
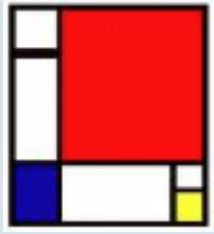
```
r.ptInRect(30,40);
// false   (boolean)
r.turn();
r.getLeft();
// 60.0   (double)
r.getTop();
// 60.0   (double)
r.getRight();
// 90.0   (double)
r.getBottom();
// -50.0   (double)
```

Rectângulo

- Defina em Java uma classe Rect cujos objectos representam rectângulos num plano.
- Programe a sua classe no Eclipse.
- Teste um (ou vários) objectos Rect, e verifique se se comportam como esperado.



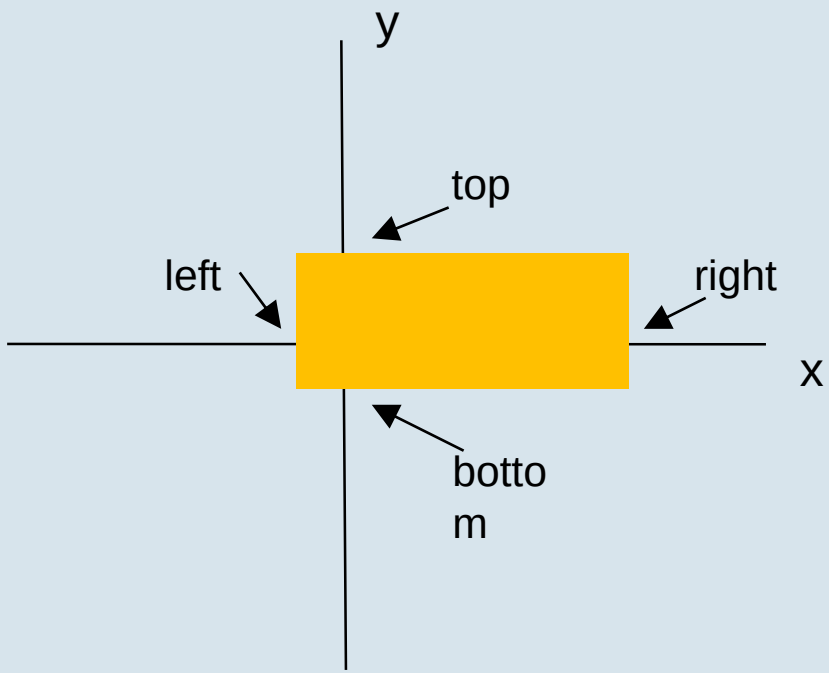


Programando o Rectângulo

```
public class Rect {
    private double left, top, right, bottom;
    ...
}
```

Variáveis de instância



Programando o Rectângulo

```
public class Rect {
   private double left, top, right, bottom;
   ...
   public Rect(){
      left = …
      top = …
      right = …
      bottom = …
   }
   ...
}
```

Variáveis de instância

Necessitamos de ter várias instruções para poder inicializar todas as variáveis de instância do objecto

constructores

Composição sequencial: ideia central

- Temos visto desde o início que um programa é uma sequência de ordens/instruções (recorde o exemplo do robot, do operador, etc)
- Nos exemplos de interacção com classes definidos no Eclipse, usaram-se sequências de ordens: primeiro criar objecto, depois chamar modificador e, finalmente, fazer consulta (para ver estado)
- Em Java usa-se o símbolo '`;`' para encadear instruções: é um operador binário que, dadas duas instruções `c1` e `c2`, constrói uma nova instrução que é a sequencialização `c1;c2`
- O comportamento da composição sequencial `c1;c2` é o seguinte: primeiro executa `c1` e, quando este acabar, executa `c2`

```
BankAccount b = new BankAccount();
b.deposit(10000);
```

Composição sequencial: onde surge, numa classe Java?

- Até agora, nos exemplos apresentados, o corpo de um método é simplesmente uma instrução (atribuição ou return), dita básica
- É útil fazer composições destas instruções básicas, por exemplo para alterar mais do que uma variável de instância – vamos ver corpos de métodos compostos
- Vimos também que uma classe Java tem vários "ingredientes": *constantes*, *variáveis*, *construtores* e *métodos*; a sua declaração **em sequência** recorre ao operador de composição sequencial, compondo assim as várias instruções para se obter o corpo da classe

Programando o Rectângulo

```
public class Rect {
   private double left, top, right, bottom;
   ...
   public Rect(){
     left = …;
     top = …;
     right = …;
     bottom = …;
   }
   ...
}
```

Variáveis de instância

Operador de sequência “;”

constructores

Programando o Rectângulo

```
public class Rect {
   private double left, top, right, bottom;
   public static final double DEFAULT_VALUE = 0.5;
   public Rect(){
     left = - DEFAULT_VALUE;
     top = DEFAULT_VALUE;
     right = DEFAULT_VALUE;
     bottom = - DEFAULT_VALUE;
   }
   ...
}
```

Constante

constructores

Declaração de Constantes

- Em certos problemas é conveniente definir o valor de certas **constantes globais**
- Porque dizemos “constantes globais”? Porque:
  - Globais, pois são usadas por **todos** os objectos de uma certa classe
  - Constantes, pois o seu valor nunca poderá ser alterado (é fixo)

- Para declarar numa classe uma constante, utiliza-se a forma

```
public static final tipo constantIdentifier = expr ;
```

A expressão ***expr*** só pode usar valores constantes. Por exemplo:

```
public static final int HOURS_IN_DAY = 24;
public static final int HOUR_TO_MINUTES = 60;
public static final int MINUTES_IN_DAY = HOURS_IN_DAY * HOUR_TO_MINUTES;
```

Declaração de Constantes

- Note que **não é possível reafectar** o valor associado a uma constante (experimente ver o que acontece no Eclipse)
- Tal como as variáveis, as constantes são referidas por um identificador, escolhido pelo programador
- Por convenção, é costume usar-se um identificador iniciado por maiúscula, ou mesmo constituído apenas por maiúsculas
- Por exemplo:
  - **MAXSIZE**
  - **MONTHS_IN_YEAR**

```
public static final int MONTHS_IN_YEAR =
12;
```

Declaração de constantes

`public` especifica que o valor desta constante pode ser usado fora desta classe.

`static` especifica que o valor da constante é partilhado por todos os objectos da classe.

`final` especifica que estamos a declarar uma constante.

```
public static final tipo constantIdentifier = expr ;
```

O valor guardado na constante é do tipo ***tipo***.

A constante é identificada por ***constantIdentifier***.

A constante assume o valor de ***expr***.

Programando o Rectângulo

```
public class Rect {
   private double left, top, right, bottom;
   ...
   public Rect(double left,double top,
               double right,double bottom){
     this.left = left;
     this.top = …;
     this.right = …;
     this.bottom = …;
   }
   ...
}
```

Variáveis de instância

Os parâmetros do constructor e as variáveis de instância da classe tem o mesmo nome. Como diferenciar o parâmetro **`left`** e a variável de instância **`left`**?

O "pronome" **`this`** refere o próprio objecto. Usa-se como prefixo na variável de instância, distinguindo-a do parâmetro.

this: referência ao objecto corrente

constructores

Programando o Rectângulo

```
public class Rect {
   private double left, top, right, bottom;
   ...
   public double getXCenter(){
     return (left+right)/2;
   }
   public double getWidth(){
     return right - left;
   }
   ...
}
```

métodos

Chamada dos Próprios Métodos

- Nos métodos de um objecto é possível invocar (chamar / utilizar) operações também definidas no mesmo objecto.
- Por exemplo, no caso da classe `Rect` podemos definir:

```
public double getArea() {
    return this.getWidth() * this.getHeight()
}
```

retorna a multiplicação dos valores **retornados** pelas chamadas aos métodos `getWidth()` e `getHeight()`

- Note que neste caso usamos o "pronome" **`this`** para referir o próprio objecto ao qual a operação deve ser aplicada.
- No exemplo, quando um objecto da classe `Rect` processa a operação `getArea()` vai calcular a área a partir das operações `getWidth()` e `getHeight()`, aplicadas a "si próprio" (**`this`**).
- **`this`** significa "eu" ou "a mim". É uma palavra reservada em Java.
- É sempre necessário indicar qual é o objecto ao qual o método deve ser aplicado, seja ele **`this`** ou outro (veremos no método `rectInRect`).

Programando o Rectângulo

```
public class Rect {

   private double left, top, right, bottom;

   ...

   public boolean ptInRect(double x, double y){

     return left <= x && right >= x

            && bottom <= y && top >= y;

   }

   ...

}
```

métodos

Programando o Rectângulo

```
public class Rect {

   ...

   public boolean rectInRect(Rect r){

     return r.getLeft() >= left && r.getRight() <= right

      && r.getBottom() >= bottom && r.getTop() <= top ;

   }

   ...

}
```

O parâmetro do método é um objecto rectângulo, que já foi criado.

Invocação dos métodos no objecto rectângulo passado como parâmetro (`r`)

Programando o Rectângulo

```
public class Rect {
    …
    public boolean rectInRect(Rect r){
      return r.getLeft() >= left && r.getRight() <= right
       && r.getBottom() >= bottom && r.getTop() <= top ;
    }
    ..
}
```

O parâmetro do método é um objecto rectângulo, que já foi criado.

Quando temos uma variável para guardar a referência de um objecto (caso `r`), como saber que o objecto ainda não foi criado?

Com que valor inicializamos? Algum dos valores dos tipos vistos serve?

- Em Java, há um valor especial → **`null`**.

Exemplos de declarações e inicializações de variáveis

```
Rect r = null; // Ainda não existe o objecto
Rect r = new Rect(); // O objecto é criado e a sua referência guardada
                     // na variável r
```

Rectângulo

- Enriquece-se a interface com um contrato de utilização.
- Na interface coloca-se a pré-condição:

boolean `rectInRect(Rect r)`

verifica se o rectângulo `r` está dentro do rectângulo

Pre: r != null

O parâmetro r é um objecto rectângulo, que já foi criado.

Programando o Rectângulo

```
public class Rect {

    private double left, top, right, bottom;

    ...

    public void turn(){
      double cx = this.getXCenter();
      double cy = this.getYCenter();
      double halfWidth = this.getWidth() / 2;
      double halfHeight = this.getHeight() / 2;
      left = cx - halfHeight;
      right = cx + halfHeight;
      bottom = cy - halfWidth;
      top = cy + halfWidth;
    }

    ...

}
```

Variáveis locais ao método

A declaração de variáveis locais é bastante parecida com a declaração de variáveis de instância, mas acontece **dentro** de um método.

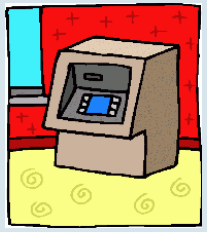
DescritorDeTipo Identificador

Variáveis locais

- Por vezes, para implementar um método, temos de salvaguardar temporariamente alguns valores, para garantir que os resultados finais da execução do método são correctos
- Dentro de um método, é possível definir **variáveis locais**, também conhecidas como **variáveis temporárias**
- A existência das variáveis locais é confinada entre a sua declaração e o fim da execução do método:
  - Quando o método terminar, a variável temporária deixa de existir
  - Se o método for invocado de novo, a variável é criada novamente, ou seja, esta variável não guarda memória do seu valor entre chamadas sucessivas ao método em que está definida
- Não confunda variáveis locais com variáveis de instância!
  - As variáveis locais existem apenas durante a execução do método
  - As variáveis de instância existem durante toda a vida do objecto

Alguns Programas Simples

- Vimos como programar em Java várias classes simples.



- Foram introduzidos vários aspectos 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 métodos dentro do método de um objecto
- Certifique-se de que **compreendeu bem** cada aspecto, não só no contexto do problema, mas também em geral