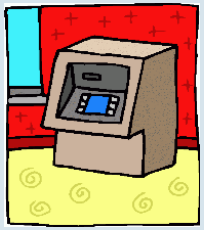
Programando em Java
(Classes Simples e Tipos de Dados Básicos)

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

- 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.

Conta Bancária

- Uma conta bancária é:
  - Um “depósito” de dinheiro
  - A quantidade de dinheiro na conta chama-se “saldo”
  - O saldo pode ser positivo (credor ou nulo) ou negativo (devedor)
- Quando a conta é criada
  - Se não indicarmos nada, o seu saldo é zero
  - Em alternativa, podemos indicar um valor inicial para o saldo
- Operações reconhecidas (interface de `BankAccount`):

```
public void deposit(int amount)
public void withdraw(int amount)
public int getBalance()
public boolean redZone()
```

Conta Bancária

- Operações reconhecidas (interface de `BankAccount`):

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

Depositar a importância `amount` na conta

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

Levantar a importância `amount` na conta

```
public int getBalance()
```

Consultar o saldo da conta

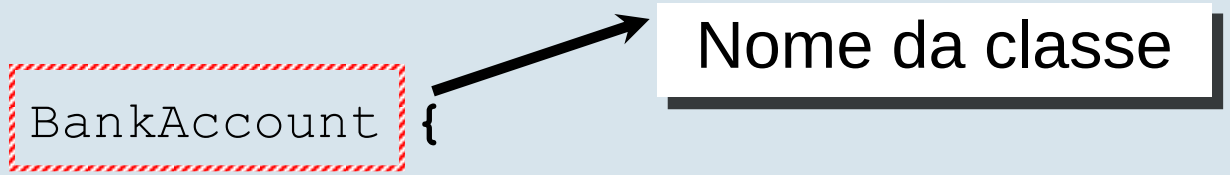
```
public boolean redZone()
```

Indica se a conta está devedora

Conta Bancária

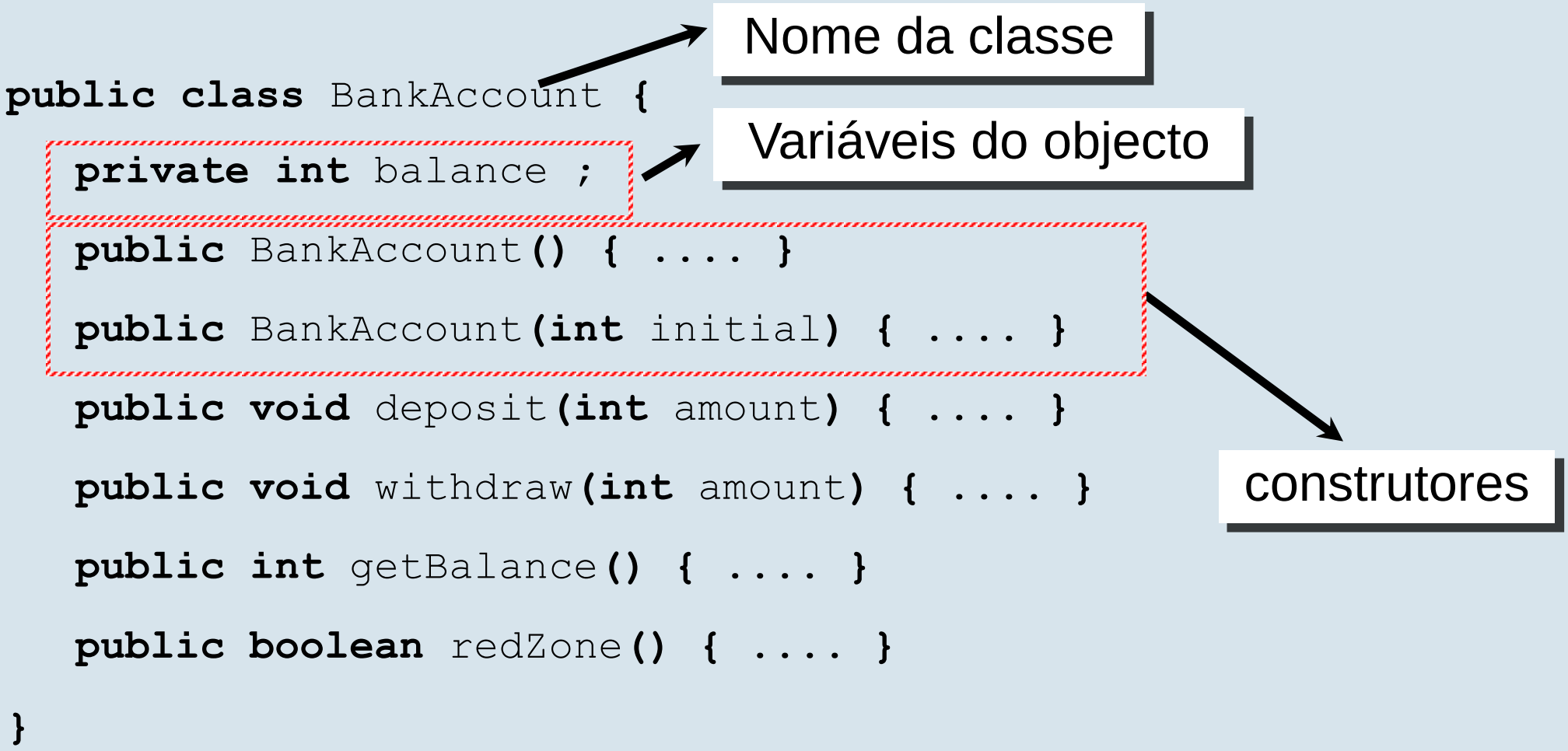
```
BankAccount b1 = new BankAccount(2000);
b1.getBalance()
2000  (int)
b1.deposit(2);
b1.deposit(8);
b1.redZone()
false  (boolean)
b1.getBalance();
2010  (int)
b1.withdraw(3000);
b1.getBalance()
-990  (int)
b1.redZone()
true  (boolean)
```

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

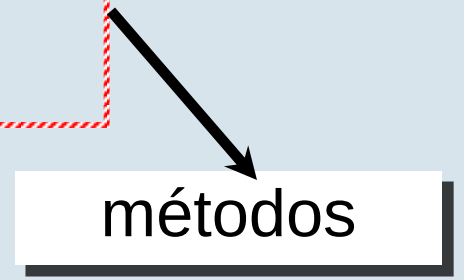
Programando a Conta Bancária




```
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) { balance = initial ; }
    public void deposit
    public void withdra
    public int getBalan
    public boolean redZ
}
```

saldo inicial

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

```
new BankAccount()      // Criação com saldo 0
new BankAccount(20)    // Criação com saldo 20
```

Método `redZone`

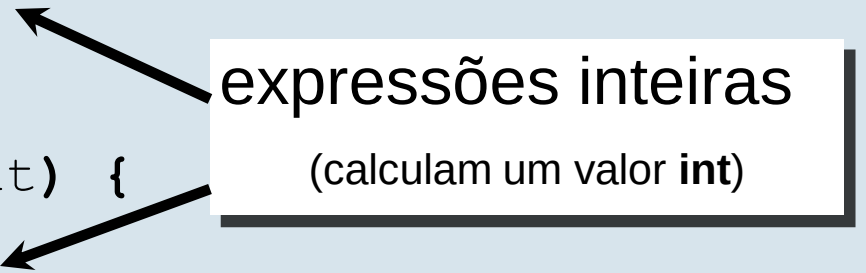
```
public class BankAccount {
   private int balance ;
   public BankAccount() { balance = 0; }
   public BankAccount(int initial) { balance = initial ;}
   public void deposit(int amount) { .... }
   public void withdraw(int amount) { .... }
   public int getBalance() { .... }
   public boolean redZone()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

```
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); }
}
```

Parâmetro do método

Métodos com Parâmetros

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

acesso tipo identificadorMétodo **()** { ***comandos*** }

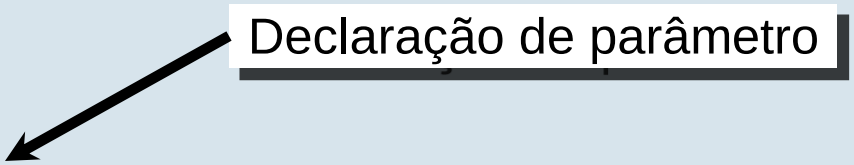
- Para fornecermos informação de entrada adicional a uma operação de um objecto, podemos introduzir parâmetros nos métodos

acesso tipo identificadorMétodo **(*tipo*** *parâmetro,* ***tipo*** *parâmetro, ...* **)** { ***comandos*** }

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, ...* **)** { ***comandos*** }

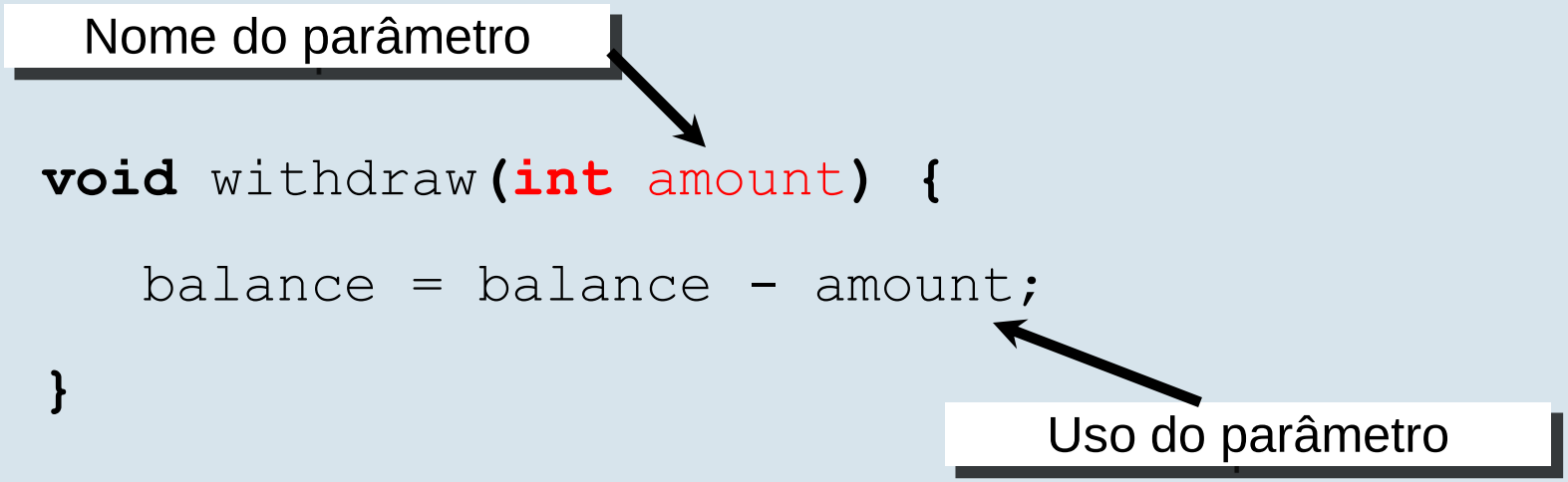



```
public void withdraw(int amount) {
    balance = balance - amount;
}
```

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, ...* **)** { ***comandos*** }



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)
```

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

Argumento da chamada

Quando o corpo do método `withdraw` é 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

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)

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**.

Tipos float / 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)
  - O tipo **`double`** (números de vírgula flutuante de precisão dupla)
- Operações que produzem um valor real (de tipo **`float`** ou **`double`**) a partir de valores reais (de tipo **`float`** ou **`double`**)

expr **`+`** *expr*	adição
expr **`-`** *expr*	subtracção
expr **`*`** *expr*	multiplicação
expr **`/`** *expr*	divisão

Tipos float / double e Operações Associadas

- 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`**
- 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`**

Tipos float / double e Operações Associadas

- A biblioteca **`Math`**, disponível no ambiente Java, oferece um conjunto bastante completo de constantes e operações para números reais.
- Constantes úteis
  - **`Math.PI`** π
  - **`Math.E`** *e*
- Operações úteis sobre valores de tipo **`double`**
  - **`Math.round(`** *expr* **`)`** arrendonda 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, na zona da documentação sobre a linguagem e as bibliotecas do ambiente Java.

Ordem de Precedência dos Operadores

- Para desambiguar a ordem de aplicação dos operadores em expressões complicadas, 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
```

Conversor

Conversor

- Defina em Java uma classe CurrencyConverter cujos objectos guardam uma importância numa determinada moeda que convertem em Euros, Dólares, e Libras.
- Programe a sua classe no BlueJ.
- Teste um (ou vários) objectos CurrencyConverter, e verifique se estes se comportam como esperado.

Conversor

- Um conversor de câmbio permite transformar uma importância expressa numa moeda noutra moeda.
  - Neste problema vamos considerar apenas conversões de e para Euros (EUR), Libras (GPB) e Dólares (USD)
  - Considere as taxas de conversão
    - 1 USD = 0.70 EUR
    - 1 EUR = 0.69 GBP
- Operações (interface de CurrencyConverter):

```
public int getInEuros()
public void setInEuros(int amount)
public int getInDollars()
public void setInDollars(int amount)
public int getInPounds()
public void setInPounds(int amount)
```

Conversor

```
public int getInEuros()
```

Consultar a importância em euros

```
public void setInEuros(int amount)
```

Definir a importância em Euros

```
public int getInDollars()
```

Consultar a importância em dólares

```
public void setInDollars(int amount)
```

Definir a importância em dólares

```
public int getInPounds()
```

Consultar a importância em libras

```
public void setInPounds(int amount)
```

Definir a importância em libras

Conversor

```
CurrencyConverter c1 = new CurrencyConverter();
c1.setInEuros(100);
c1.getInEuros()
100  (int)
c1.getInDollars();
143  (int)
c1.getInPounds()
69  (int)
c1.setInPounds(200);
c1.getInEuros()
290  (int)
c1.getInDollars()
414  (int)
c1.getInPounds()
200  (int)
```

Declaração de Constantes

- Em certos problemas, como no de programar o Conversor, 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)
- Exemplo: o identificador **`PI`** é uma constante, de tipo **`double`**, definida na classe **`Math`**.
- Para declarar numa classe uma constante, utiliza-se a forma

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

A expressão ***expr*** só pode mencionar outras constantes. Por exemplo,

```
public static final double PI2 = 2*Math.PI;
```

Declaração de Constantes

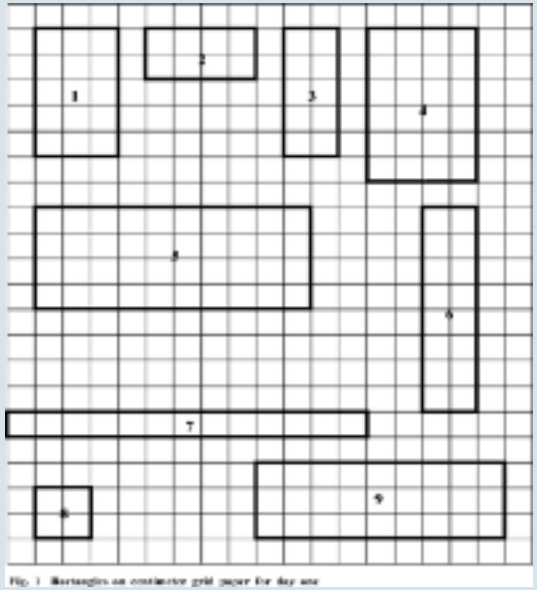
- Note que **não é possível reafectar** o valor associado a uma constante (experimente ver o que acontece no BlueJ)
- Tal como as variáveis, as constantes são referidas por um identificador, escolhido pelo programador
- Por convenção, é costume usar-se um identificador iniciado por maiúscula, ou mesmo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

```
public class CurrencyConverter {
    public static final float USD2EUR = 0.70f ;
    ...
}
```

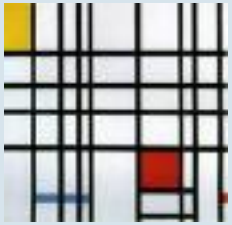
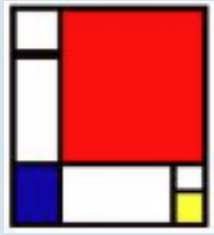
Rectângulo

Rectângulo

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





Rectângulo

- Cada objecto Rect
  - Representa um rectângulo no plano, com lados paralelos aos eixos
  - As coordenadas devem ser de precisão muito grande
- Pretende-se construir objectos Rect de **várias** maneiras:
  - Não indicando nada (o rectâgulo será um quadrado com centro na origem do plano)
  - Indicando:
    - A abcissa do canto superior esquerdo
    - A ordenada do canto superior esquerdo
    - A abcissa do canto inferior direito
    - A ordenada do canto inferior direito
  - Indicando
    - A abcissa do canto superior esquerdo
    - A ordenada do canto superior esquerdo
    - A dimensão do lado (neste caso o rectângulo é um quadrado)

Rectângulo

- Operações reconhecidas (interface de Rect):

```
public double getLeft()
```
consulta a abcissa dos cantos esquerdos

```
public double getRight()
```
consulta a abcissa dos cantos direitos

```
public double getTop()
```
consulta a ordenada dos cantos superiores

```
public double getBottom()
```
consulta a ordenada dos cantos inferiores

Rectângulo

- Operações reconhecidas (interface de Rect):

```
public double getXCenter()
```

consulta a abcissa do centro do rectângulo

```
public double getYCenter()
```

consulta a ordenada do centro do rectângulo

```
public double getWidth()
```

consulta (calcula) a largura do rectângulo

```
public double getHeight()
```

consulta a altura do rectângulo

```
public double getPerimeter()
```

consulta o perímetro do rectângulo

Rectângulo

- Operações reconhecidas (interface de Rect):

`public double` `getArea()`

consulta a área do rectângulo

`public boolean` `ptInRect(double x, double y)`

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

`public void` `translate(double dx, double dy)`

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

`public void` `turn()`

roda o rectângulo 90º para a direita

Múltiplos Construtores

```
public class Rect {
    ... ; /* Definição de constantes e variáveis de instância */
    public Rect() { ...; }
    public Rect(double left, double top, double right, double bottom)
    { ... ; }
    public Rect(double top, double left, double length)
    { ... ; }
    ...  /* Definição dos restantes métodos da classe */
}
```

- 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

Chamada dos Próprios Métodos

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

```
public double getArea() {
   return this.getWidth() * this.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 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 mais tarde).

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 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

Declaração e utilização de variáveis locais

- A declaração de variáveis locais é bastante parecida com a declaração de variáveis de instância, mas acontece **dentro** de um método
- Por exemplo, na classe Rect, podemos querer definir, dentro do método turn, a variável local centerX:

```
public void turn(){
    double centerX = this.getXCenter();
    ... /* resto do método aqui */
}
```

Rectângulo

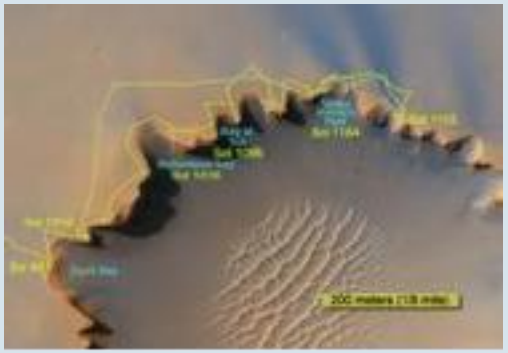
```
Rect r = new Rect(10,50,120,20);
r.getArea()
3300.0   (double)
r.getPerimeter()
280.0   (double)
r.ptInRect(30,40)
true   (boolean)
r.ptInRect(130,40)
false   (boolean)
r.translate(10,-30);
r.ptInRect(130,40)
false   (boolean)
```

```
r.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)
```

Mars Rover

Mars Rover

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



Mars Rover

- O MarsRover
  - É um veículo de exploração do planeta Marte.
  - Desloca-se num plano imaginário sobreposto à superfície do planeta, em que cada posição é indicada por um par de coordenadas (números reais).
  - Uma das funcionalidades do MarsRover é medir as distâncias (em linha recta) entre vários pontos no terreno.

Mars Rover

- Cada objecto MarsRover
  - Conhece a sua posição no terreno:
    - A posição é indicada por um par de coordenadas X, Y
  - Conhece a sua direcção de deslocação
    - A direcção é indicada por um ângulo α
- Operações reconhecidas (interface do MarsRover):

```
public void moveForward(double distance)
public void setHeading(double angle)
public void mark()
public double getXPos()
public double getYPos()
public double getHeading()
```

Mars Rover

`public void moveForward(double distance)`

O Rover avança `distance` unidades na direcção corrente

`public void setHeading(double angle)`

O Rover vira-se para a direcção absoluta `angle`.

O ângulo deverá ser indicado em graus.

`public double getXPos()`

Consultar o valor da coordenada `X`.

`public double getYPos()`

Consultar o valor da coordenada `Y`.

`public double getHeading()`

Consultar o valor da orientação do Rover.

Mars Rover

`public void mark()`

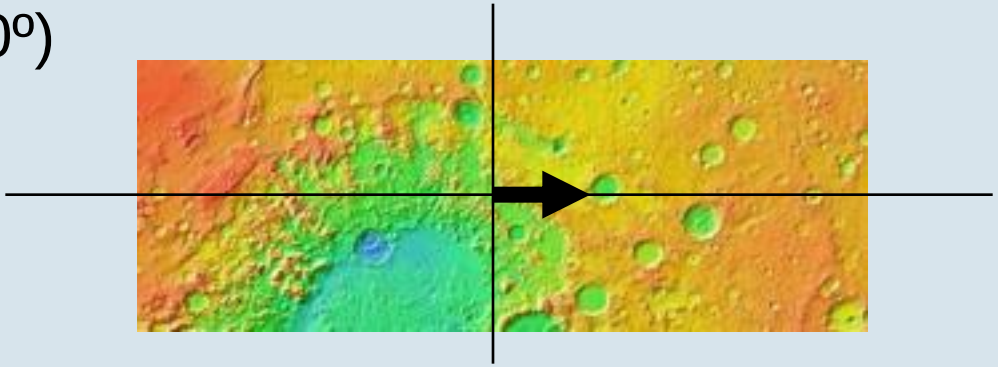
O Rover regista o ponto corrente como ponto base

`public double getDistance()`

O Rover indica a distância (em linha recta) entre o último ponto base marcado e a sua posição corrente.

Quando é criado, cada objecto MarsRover

- Encontra-se na origem do plano (X=0,Y=0)
- Encontra-se virado para leste (α=0º)



Mars Rover

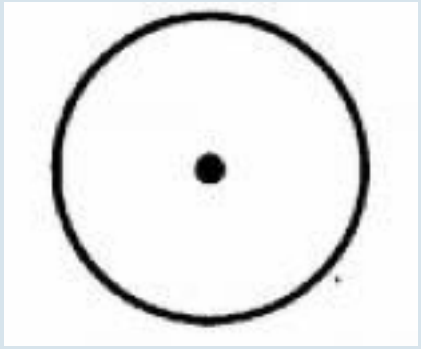
```
MarsRover r = new MarsRover();
r.getXPos()
0.0   (double)
r.getYPos()
0.0   (double)
r.setHeading(90);
r.moveForward(1.0);
r.getXPos()
6.123233995736766E-17 (double)
r.getYPos()
1.0   (double)
r.moveForward(-2);
```

```
r.getYPos()
-1.0   (double)
r.getXPos()
-6.123233995736766E-17 (double)
r.mark();
r.moveForward(2);
r.getDistance()
2.0   (double)
r.setHeading(0);
r.moveForward(2);
r.getDistance()
2.828427124746l903   (double)
```

Círculo

Círculo

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



Círculo

- Um círculo define-se por
  - Um ponto no plano (par de coordenadas X, Y) indicando o centro
  - O comprimento do raio
- Pretende-se construir círculos de várias maneiras:
  - Dados o seu centro e raio
  - Dado o seu raio
    - centro será na origem do plano
  - Sem indicar nada
    - centro será na origem do plano e o raio será unitário

Círculo

- Operações reconhecidas (interface do Circle):

```
public double getPerimeter()
public double getArea()
public double getRadius()
public double getXCenter()
public double getYCenter()
public boolean ptInCircle(double x, double y)
public void translate(double dx, double dy)
```

Círculo

public double `getPerimeter`**()**

Devolve o perímetro do círculo

public double `getArea`**()**

Devolve a área do círculo

public double `getRadius`**()**

Devolve o comprimento do raio do círculo

public double `getXCenter`**()**

Devolve a abcissa do centro

public double `getYCenter`**()**

Devolve a ordenada do centro

public boolean `ptInCircle`**(double** `x`**, double** `y`**)**

Indica se o ponto (x,y) se encontra no círculo

public void `translate`**(double** `dx`**, double** `dy`**)**

Desloca o círculo segundo o vector <dx,dy>

Círculo

```
Circle c = new Circle();
c.getRadius()
1.0   (double)
c.getArea()
3.141592653589793   (double)
c.translate(1,1);
c.getArea()
3.141592653589793   (double)
c.ptInCircle(-1,-1)
false   (boolean)
c.getXCenter()
1.0   (double)
c.getYCenter()
1.0   (double)
```

```
Circle d = new Circle(1,1,10);
d.getArea()
314.1592653589793   (double)
d.translate(1,2);
d.getXCenter()
2.0   (double)
d.getYCenter()
3.0   (double)
d.getArea()
314.1592653589793   (double)
Circle e = new Circle(5);
e.getRadius()
5.0   (double)
```

Múltiplos Construtores

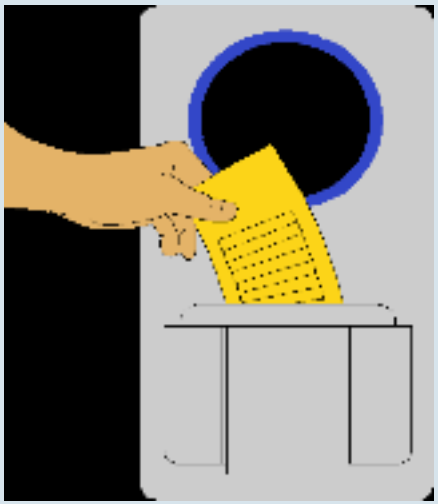
```
class Circle {
 .... ;
    Circle() {...; }
    Circle(double radius) { ... ; }
    Circle(double cx, double cy, double radius) { ... ; }
 ...
}
```

- 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

Tempo

Tempo

- Defina em Java uma classe Time cujos objectos representam uma quantidade de tempo.
- Programe a sua classe no BlueJ.
- Teste um (ou vários) objectos Time, e verifique se se comportam como esperado.



Tempo

- Cada objecto Time
  - Representa uma quantidade de tempo
  - A precisão é ao nível do segundo
- Pretende-se construir um objecto Time de duas formas
  - Indicando nada (significa 0 segundos)
  - Indicando uma quantidade de horas, minutos e segundos

Tempo

- Operações reconhecidas (interface de Time):

```
public int getSeconds()
public int getMinutes()
public int getHours()
public void addSeconds(int s)
public void addMinutes(int m)
public void addHours(int h)
public void subtractSeconds(int s)
public void subtractMinutes(int m)
public void subtractHours(int h)
public String asString()
```

Tempo

```
Time t = new Time(10,50,3);
t.asString()
"10 horas 50 minutos 3 segundos"   (String)
t.addSeconds(40)
t.asString()
"10 horas 50 minutos 43 segundos"   (String)
t.addHours(5)
t.asString()
"15 horas 50 minutos 43 segundos"   (String)
t.addHours(19)
t.asString()
"34 horas 50 minutos 43 segundos"   (String)
t.subtractMinutes(100)
t.asString()
"33 horas 10 minutos 43 segundos"   (String)
```

Tipo `String`

- Os objectos Time fornecem uma operação que permite consultar o seu valor como uma sequência de caracteres

  "10 horas 50 minutos 43 segundos"

  – Em Java as sequências de caracteres são representadas como valores do tipo **`String`**

  – As strings representam "bocados" de texto, que podem ser manipulados pelos programas através de operações especiais

  – Os valores de tipo **`String`** escrevem-se como sequências de caracteres arbitrários (menos as "'s) entre " ".

  Por exemplo:

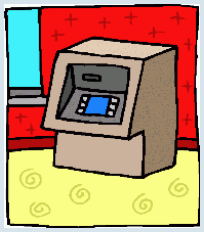
  - `"As armas e os barões assinalados"`
  - `"%&Ghg1j9892bnsabdGHJbsu?*"`
  - `"this is a recording, please hang up!"`
  - `"class"`

Operações básicas sobre o tipo `String`

- A operação mais básica que se pode efectuar com strings é a concatenação (justaposição) **+**.
  - `"hello"` **+** `" "` **+** `"world"` produz `"hello world"`
  - `"X"` **+** `"Y"` produz `"XY"`
- É possível concatenar também inteiros (**`int`**) com strings, nesse caso o inteiro é convertido na sua representação textual (por exemplo `12` é convertido em `"12"`).
  - `"Altura="` **+** `height` produz `"Altura=10"`
  - `"Horas "` **+** `mi` **+** `" minutos"` produz `"Horas 0 minutos"`
  - `"Ping"` **+** `(2-1)` produz `"Ping1"`
  - Note que o contrário não é possível. Uma string não pode ser guardada numa variável de tipo **`int`**, mesmo que represente um número (por exemplo `"23"`)

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 outros 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