

# Programando em Java

## (Classes Simples e Tipos de Dados Básicos)

### Lista de Exercícios

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

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

**Mestrado Integrado em Engenharia Informática FCT UNL**

# Objectivos

- O aluno deverá ser capaz de:
  - A partir de uma especificação simples, em língua natural, identificar a classe a definir e propor uma interface para essa classe .
  - Usar o BlueJ na construção de classes simples, definindo:
    - Variáveis de instância (privadas);
    - Construtor de instâncias da classe (público);
    - Operações modificadoras (públicas);
    - Operações de consulta (públicas).
  - Usar operações aritméticas simples na implementação da classe, que envolvam valores inteiros, reais e lógicos.
  - Testar cuidadosamente as classes desenvolvidas, analisando com espírito crítico os resultados produzidos pelos testes.
  - Interpretando correctamente as eventuais mensagens de erro resultantes de defeitos no código produzido.

# Sensor de movimento

# Sensor de movimento

- **Objectivo**
  - Simular um sensor de movimento.
- **Descrição**
  - Um sensor de movimento é capaz de detectar movimento. O seu estado pode ser: “em movimento” ou “parado”.
- **Funcionalidades**
  - A deteção de movimento ou não é realizada através das operações de “move” e “stop”, respectivamente.
  - Quando o sensor é criado o seu estado é “parado”.
  - Deve ser sempre possível consultar se o estado é “em movimento” com a operação “isMoving”.
- **Interacção com o utilizador**
  - Após criar um sensor, pode invocar as operações do sensor.

# Sensor de movimento

- **Que objecto se deve definir?**
  - Um sensor de movimento (classe `Sensor`).
- **Interface:**

**void** move()

Detectar movimento

**void** stop()

Detectar ausência de movimento

**boolean** isMoving()

Consultar se há movimento detectado

# Sensor de movimento

```
Sensor s1 = new Sensor();
```

```
Sensor s2 = new Sensor();
```

```
s1.isMoving()
```

```
false (boolean)
```

```
s2.isMoving()
```

```
false (boolean)
```

```
s1.move();
```

```
s1.isMoving()
```

```
true (boolean)
```

```
s2.isMoving()
```

```
false (boolean)
```

```
s2.stop();
```

```
s2.isMoving()
```

```
false (boolean)
```

```
s2.move();
```

```
s2.isMoving()
```

```
true (boolean)
```

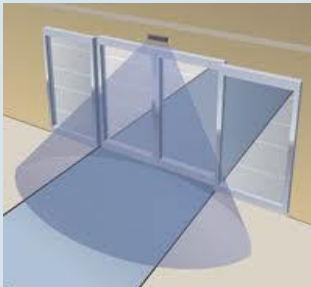
```
s2.stop();
```

```
s2.isMoving()
```

```
false (boolean)
```

# Sensor de movimento

- Defina em Java uma classe `Sensor`, cujos objectos são sensores de movimento. Estes sensores são usados, por exemplo, em edifícios inteligentes, como forma de poupar energia, abrir automaticamente portas, ou detectar intrusos, em sistemas de entretenimento , etc
- Programe a sua classe no BlueJ
- Teste vários objectos da classe `Sensor` e verifique que se comportam tal como esperado.



# FairPlay



# FairPlay

- **Objectivo**

- Simular o estado de espírito de um adepto de futebol.

- **Descrição**

- O estado de espírito de um adepto de futebol pode ser “feliz” ou “deprimido”.
- O adepto apenas está feliz na sequência da vitória da própria equipa, ou do empate ou derrota de uma equipa rival. Tudo o resto é um drama para o adepto (e um caso de polícia evidente), deixando o adepto deprimido.

- **Funcionalidades**

- O adepto é sempre informado dos resultados dos jogos da sua equipa e das equipas rivais: ganhou, empatou ou perdeu.
- Quando o adepto é criado o seu estado de espírito é “feliz”.
- Deve ser sempre possível consultar se o adepto está “feliz”.

- **Interacção com o utilizador**

- Após criar um adepto, pode invocar as operações.

# FairPlay

- **Interface** (classe FairPlay) :

**boolean** isHappy()

Devolve true se o adepto está feliz, false caso contrário.

**void** weWon()

A equipa do adepto venceu, ele fica feliz.

**void** weDrew()

A equipa do adepto empatou, ele não está feliz.

**void** weLost()

Trágica derrota, o adepto não está feliz.

# FairPlay

- **Interface** (classe **FairPlay**) :

...

**void** theyWon()

Lamentável vitória de um rival, o adepto fica triste.

**void** theyDrew()

Finalmente o rival perdeu pontos, o adepto fica contente.

**void** theyLost()

Glorioso momento, o rival está de rastos e o adepto fica feliz.

# FairPlay

```
FairPlay b = new FairPlay();
```

```
b.happy()
```

```
true    (boolean)
```

```
b.weLost();
```

```
b.happy()
```

```
false   (boolean)
```

```
b.theyLost();
```

```
b.happy()
```

```
true    (boolean)
```

```
b.weLost();
```

```
b.happy()
```

```
false   (boolean)
```

```
b.weDrew();
```

```
b.happy()
```

```
false   (boolean)
```

```
b.theyDrew();
```

```
b.happy()
```

```
true    (boolean)
```

# FairPlay

- Defina em Java uma classe `FairPlay`, cujos objectos são usados para descobrir o estado de espírito de um adepto de futebol.
- Programe a sua classe no BlueJ
- Teste vários objectos da classe `FairPlay` e verifique que se comportam tal como esperado.



# Semáforo



# Semáforo

- **Objectivo**
  - Simular um semáforo.
- **Descrição**
  - O estado de um semáforo pode ser: “vermelho”, “verde”, ou “amarelo”.
- **Funcionalidades**
  - O semáforo é sempre informado para mudar de estado, sendo que a mudança ocorre sempre da seguinte forma:  
vermelho → verde → amarelo → vermelho → ...
  - É sempre possível consultar se o semáforo está vermelho, verde ou amarelo. Assim como consultar se é possível passar (estado em verde ou amarelo) e se é necessário parar (estado vermelho).
  - Quando o semáforo é criado o seu estado é “vermelho”.
- **Interacção com o utilizador**
  - Após criar um semáforo, pode invocar as operações do semáforo.

# Semáforo

- Interface (classeTrafficLight) :

**boolean** isRed()

Indica se o semáforo está vermelho.

**boolean** isGreen()

Indica se o semáforo está verde.

**boolean** isYellow()

Indica se o semáforo está amarelo.

**boolean** pass()

Indica se se pode passar.

**boolean** stop()

Indica se temos de parar.

**void** changeColor()

Muda/Troca o estado do semáforo.



# Semáforo

```
TrafficLight t1 = new TrafficLight();
```

```
t1.isGreen()
```

```
false      (boolean)
```

```
t1.isRed()
```

```
true       (boolean)
```

```
t1.pass()
```

```
false      (boolean)
```

```
t1.stop()
```

```
true       (boolean)
```

```
t1.changeColor();
```

```
t1.isYellow()
```

```
false      (boolean)
```

```
t1.pass()
```

```
true       (boolean)
```

```
t1.changeColor();
```

```
t1.isGreen()
```

```
false      (boolean)
```

```
t1.isYellow()
```

```
true       (boolean)
```

```
t1.pass()
```

```
true       (boolean)
```

```
t1.changeColor();
```

```
t1.pass()
```

```
false      (boolean)
```

# Semáforo

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



# Dieta

# Dieta

- **Objectivo**
  - Controlar uma dieta.
- **Descrição**
  - Numa dieta há ingestão de alimentos, com as odiadas calorias, e exercício físico, em que a pessoa em dieta se livra dos excessos cometidos.
- **Funcionalidades**
  - Em cada refeição, ou exercício, registam-se sempre as calorias ganhas, ou perdidas, respectivamente.
  - É necessário saber sempre as calorias retidas (ingeridas que não são perdidas). Assim como se o valor das calorias é negativo.
  - É sempre possível consultar o número médio de calorias ingeridas e perdidas. Assim como consultar o número de refeições e exercícios realizados.
  - Quando é criada, as calorias existentes são zero.
- **Interacção com o utilizador**
  - Após criar uma dieta, pode invocar as operações.

# Dieta

- Interface (classe `Diet`):

**void** eat(**int** c)

Ingere c calorias numa refeição

pre:  $c > 0$

**void** burn(**int** c)

Consome c calorias a realizar um exercício

pre:  $c > 0$

**int** eatTimes()

Indica o número de refeições realizadas

**void** burnTimes()

Indica o número de exercícios realizados

# Dieta

- Interface (classe `Diet`):

**int** `balance()`

Devolve o saldo total de calorias

**boolean** `isBalanceNegative()`

Indica se o saldo total de calorias é negativo

**float** `averageEatenCallories()`

Devolve o valor médio das calorias ingeridas

pre: `eatTimes() > 0`

**float** `averageBurntCallories()`

Devolve o valor médio das calorias consumidas

pre: `burnTimes() > 0`

# Dieta

```
Diet d = new Diet();  
d.balance()  
0    (int)  
d.eatTimes()  
0    (int)  
d.eat(50);  
d.eat(75);  
d.eatTimes()  
2    (int)  
d.burnTimes()  
0    (int)  
d.burn(20);  
d.balance()  
105  (int)  
d.eat(75);  
d.eat(100);  
d.burn(40);
```

```
d.burn(30);  
d.balance()  
210  (int)  
d.eatTimes()  
4    (int)  
d.burnTimes()  
3    (int)  
d.averageEatenCalories()  
75.0  (float)  
d.averageBurntCalories()  
30.0  (float)  
d.isBalanceNegative()  
false  (boolean)
```

# Dieta

- Defina em Java uma classe `Diet`.
- Programe a sua classe no BlueJ e forneça também um programa principal para testar a sua dieta.
- Teste no programa principal vários objectos da classe `Diet` e verifique que se comportam tal como esperado.





# Mars Rover

# Mars Rover

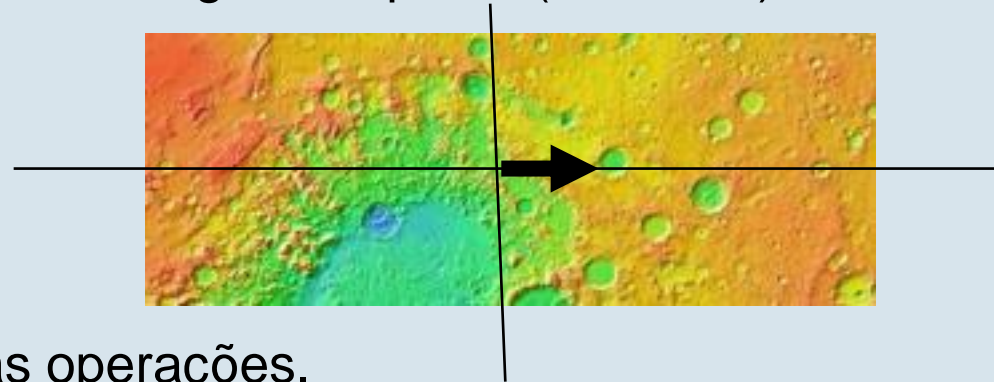
- **Objectivo**
  - Simular um veículo de exploração do planeta Marte.
- **Descrição**
  - Um veículo de exploração do planeta Marte, desloca-se num plano imaginário sobreposto à superfície do planeta, em que cada posição é indicada por um par de coordenadas (números reais).
  - Um veículo encontra-se sempre numa dada posição  $(x,y)$  e com uma dada orientação (direcção absoluta).



# Mars Rover

- **Funcionalidades**

- O veículo desloca-se em linha recta x metros apartir da posição corrente, e na direcção em que se encontra. O veículo pode também mudar de direcção.
- É sempre possível consultar as coordenadas da posição corrente do veículo, assim como a sua direcção corrente.
- Estes veículos conseguem medir distâncias (em linha recta) entre vários pontos no terreno. Mais precisamente, podem registar um dado ponto e depois calcular a distância desse último ponto registado até à posição corrente do veículo.
- Quando é criado, o veículo encontra-se na origem do plano ( $X=0, Y=0$ ), virado para leste ( $\alpha=0^\circ$ ).



- **Interacção com o utilizador**

- Após criar um veículo, pode invocar as operações.

# Mars Rover

- **Interface**

**void** moveForward(**double** distance)

O Rover avança distance unidades na direcção corrente.

pre: distance > 0

**void** setHeading(**double** angle)

O Rover vira-se para a direcção absoluta angle.

pre: angle >= 0 (unidades graus)

**double** getXPos()

Consultar o valor da coordenada X.

**double** getYPos()

Consultar o valor da coordenada Y.

**double** getHeading()

Consultar o valor da orientação do Rover.



# Mars Rover

- **Interface**

...

**void** mark()

O Rover regista o ponto corrente como ponto base, para próximo cálculo de distância.

**double** getDistance()

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

# 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()  
2.8284271247461903    (double)
```

# 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.



# Speed



# Speed

- **Objectivo**

- Manipular valores de velocidades.

- **Descrição**

- A velocidade é um valor real que pode ser expresso em diferentes unidades: metros por segundo (m/s), quilómetros por hora (km/h) e milha por hora (mph) .

- **Funcionalidades**

- É sempre possível registar e consultar o valor de velocidade em qualquer unidade. As conversões a considerar são:

	m/s	km/h	mph
1 m/s =	1	3.6	2.236936
1 km/h =	0.277778	1	0.621371
1 mph =	0.44704	1.609344	1

- Quando é criado, o valor da velocidade é de 0 m/s.

- **Interacção com o utilizador**

- Registar valores de velocidades e fazer conversões entre unidades.

# Speed

- **Interface (classe Speed)**

`double getInMS()`

**Consultar a velocidade em metros por segundo.**

`void setInMS(double speed)`

**Definir a velocidade em metros por segundo (Pre: speed >0)**

`double getInKmH()`

**Consultar a velocidade em kilómetros por hora**

`void setInKmH(double speed)`

**Definir a velocidade em kilómetros por hora (Pre: speed >0)**

`double getInMpH()`

**Consultar a velocidade em milhas por hora**

`void setInMpH(double speed)`

**Definir a velocidade em milhas por hora (Pre: speed >0)**

# Speed

```
Speed caracol = new Speed();
caracol.setInMS(0.001);
caracol.getInMS()
0.001    (double)
caracol.getInKmH()
0.003600000000000000003    (double)
caracol.getInMpH()
0.002236936    (double)
Speed obikwelu = new Speed();
obikwelu.setInKmH(36.0);
obikwelu.getInMS()
10.0    (double)
obikwelu.getInKmH()
36.0    (double)
obikwelu.getInMpH()
22.36936    (double)
obikwelu.setInMpH(24.0);
obikwelu.getInKmH()
38.624261042783516    (double)
```

# Speed

- Defina em Java uma classe `Speed`, cujos objectos representam o conceito de velocidade.
- Programe a sua classe no BlueJ
- Teste vários objectos da classe `Speed` e verifique que se comportam tal como esperado.



# Mapa do Tesouro

# Mapa do Tesouro

- **Objectivo**
  - Descobrir o tesouro.
- **Descrição**
  - O tesouro é um cofre enterrado algures numa ilha, numa dada posição no plano XY. Existe sempre um pirata, numa dada posição que procura o tesouro.
  - Este mapa funciona num tablet equipado com GPS em que as coordenadas são dadas em passos (valores inteiros). 😊
- **Funcionalidades**
  - O pirata pode sempre mover-se n passos em qualquer das quatro direcções: norte, sul, este e oeste.
  - O pirata pode sempre escavar o terreno na posição corrente de modo a saber se encontrou o tesouro.

# Mapa do Tesouro

...

- É sempre possível consultar as coordenadas da posição do pirata, assim como a sua direcção.
- Quando é criado, o pirata encontra-se na posição (0,0) virado para norte, e é indicada a posição do tesouro.
- **Interacção com o utilizador**
  - Após criar o mapa do tesouro, mover o pirata para encontrar o tesouro.

# Mapa do Tesouro

- Interface (classe do TreasureMap):

```
void walkNorth (int steps)
```

O pirata move-se **steps** passos (unidades) na direcção norte

**pre:** steps > 0

```
void walkEast (int steps)
```

O pirata move-se **steps** passos (unidades) na direcção este

**pre:** steps > 0

```
void walkSouth (int steps)
```

O pirata move-se **steps** passos (unidades) na direcção sul

**pre:** steps > 0

```
void walkWest (int steps)
```

O pirata move-se **steps** passos (unidades) na direcção oeste

**pre:** steps > 0



# Mapa do Tesouro

- Interface (classe do `TreasureMap`):

...

`boolean dig()`

O pirata escava na posição corrente e verifica se está o tesouro. Devolve `true`, se estiver precisamente no local do tesouro, ou `false`, caso contrário.

`int getXPos()`

Consulta a abcissa da posição corrente do pirata.

`int getYPos()`

Consulta a ordenada da posição corrente do pirata.

`int getCurrentDirection()`

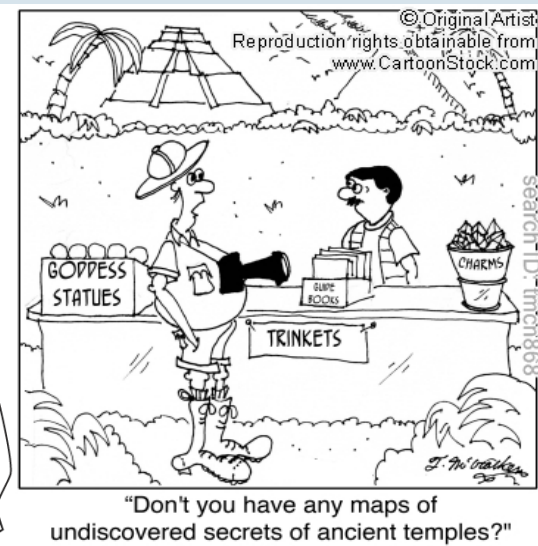
Consulta a direcção corrente do pirata. Devolve 0 = Norte; 1 = Este; 2 = Sul; 3 = Oeste.

# Mapa do Tesouro

```
TreasureMap map = new TreasureMap(8, -4);
map.walkNorth(1);
map.walkWest(3);
map.getXPos()
-3      (int)
map.getYPos()
1       (int)
map.dig()
false   (boolean)
map.walkSouth(6);
map.walkEast(10);
map.walkNorth(1);
map.dig()
false   (boolean)
map.walkEast(1);
map.dig()
true     (boolean)
```

# Mapa do Tesouro

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



# Decatlo

# Decatlo

- **Objectivo**

- Manipular os resultados de um atleta de Decatlo.

- **Descrição**

- O decatlo consiste em 10 provas realizadas em dois dias: (1) 100 metros, salto em comprimento, tiro, salto em altura, e 400 metros; (2) 110 metros barreiras, lançamento do disco, salto com vara, lançamento do dardo e 1500 metros.
- Cada prova tem uma pontuação (valor real). Há 3 tipos de provas, cuja pontuação ( $P$ ) se calcula do seguinte modo:
  - Corridas:  $P=a*(b - T)^c$   
[onde  $T$  é o Tempo em segundos;  
e.g. 10.43 para 100 metros]
  - Saltos:  $P=a*(M - b)^c$   
[onde  $M$  é a medida em centímetros;  
e.g. 808 para o salto em comprimento]
  - Lançamentos:  $P=a*(D - b)^c$   
[onde  $D$  é a distância em metros;  
e.g. 16.69 para a prova de tiro]

# Decatlo

- As constantes a, b e c, para cada prova, definem-se de acordo com a seguinte tabela.

Unidade	Prova	a	b	c
Segundos (double)	100m	25.4347	18.00	1.81
Segundos (double)	400m	1.53775	82.00	1.81
Segundos (double)	1500m	0.03768	480.00	1.85
Segundos (double)	110m Barreiras	5.74352	28.50	1.92
Centímetros (int)	High Jump	0.8465	75.00	1.42
Centímetros (int)	Pole Vault	0.2797	100.00	1.35
Centímetros (int)	Long Jump	0.14354	220.00	1.40
Metros (double)	Shot	51.39	1.50	1.05
Metros (double)	Discus	12.91	4.00	1.10
Metros (double)	Javelin	10.14	7.00	1.08

# Decatlo

- A pontuação global do atleta é a parte inteira do valor resultante da soma de todas as pontuações obtidas nas diferentes provas do decatlo.

- **Funcionalidades**

- Registrar o desempenho (tempo em segundos, medida em centímetros ou distância em metros) do atleta nas diferentes provas do decatlo (corridas, saltos ou lançamentos).
- É sempre possível consultar a pontuação do atleta numa dada prova do decatlo, assim como a pontuação global (soma da pontuação de todas as provas).

- **Interacção com o utilizador**

- Registrar e calcular pontuações das provas realizadas por um atleta no decatlo.

# Decatlo

- **Interface (classe Decathlon):**

```
// Sets com o resultado na respectiva prova
// Pre: t > 0, m > 0, d > 0 (nos respectivos métodos)
void set100Meters(double t)
void set400Meters(double t)
void set1500Meters(double t)
void set110MetersHurdles(double t)
void setHighJump(int m)
void setLongJump(int m)
void setPoleVaultJump(int m)
void setShot(double d)
void setDiscus(double d)
void setJavelin(double d)

void reset() // deita fora todos os resultados
```



# Decatlo

```
// gets com a pontuação na respectiva prova
double get100Meters()
double get400Meters()
double get1500Meters()
double get110MetersHurdles()
double getHighJump()
double getLongJump()
double getPoleVaultJump()
double getShot()
double getDiscus()
double getJavelin()
/* calcula pontuação total, como somatório da pontuação
   de cada prova, arredondado para baixo */
int getPoints()
```

# Decatlo

```
Decathlon d = new Decathlon();  
d.set100Meters(10.4f);  
d.getPoints()  
999    (int)  
d.reset();  
d.getPoints()  
0      (int)  
d.set100Meters(10.4f);  
d.setLongJump(736);  
d.getPoints()  
1900   (int)
```

```
d.setShot(13.53f);  
d.getPoints()  
2600   (int)  
d.setHighJump(210);  
d.getPoints()  
3497   (int)
```

# Decatlo

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

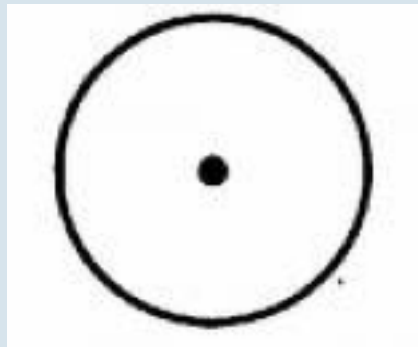


# Outros exercícios

# 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 (Pre: raio  $> 0$ )
- 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):

```
double getPerimeter()
```

```
double getArea()
```

```
double getRadius()
```

```
double getXCenter()
```

```
double getYCenter()
```

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

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



# Círculo

**double** getPerimeter()

Devolve o perímetro do círculo

**double** getArea()

Devolve a área do círculo

**double** getRadius()

Devolve o comprimento do raio do círculo

**double** getXCenter()

Devolve a abcissa do centro

**double** getYCenter()

Devolve a ordenada do centro

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

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

**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

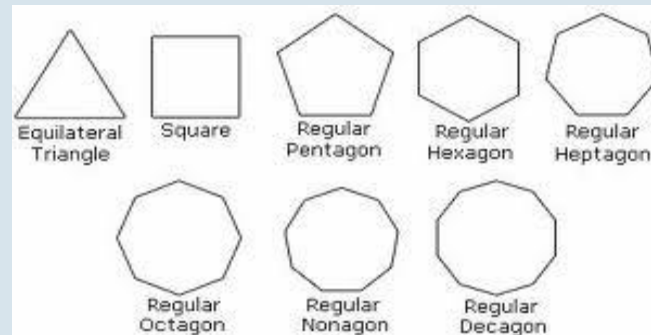
```
class Circle {  
    ... ;  
  
    Circle() {...; }  
  
    Circle(double radius) { ... ; }  
  
    Circle(double cx, double cy, double radius) {  
        ... ;  
    }  
  
    ...  
}
```

- 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

# Polígonos regulares

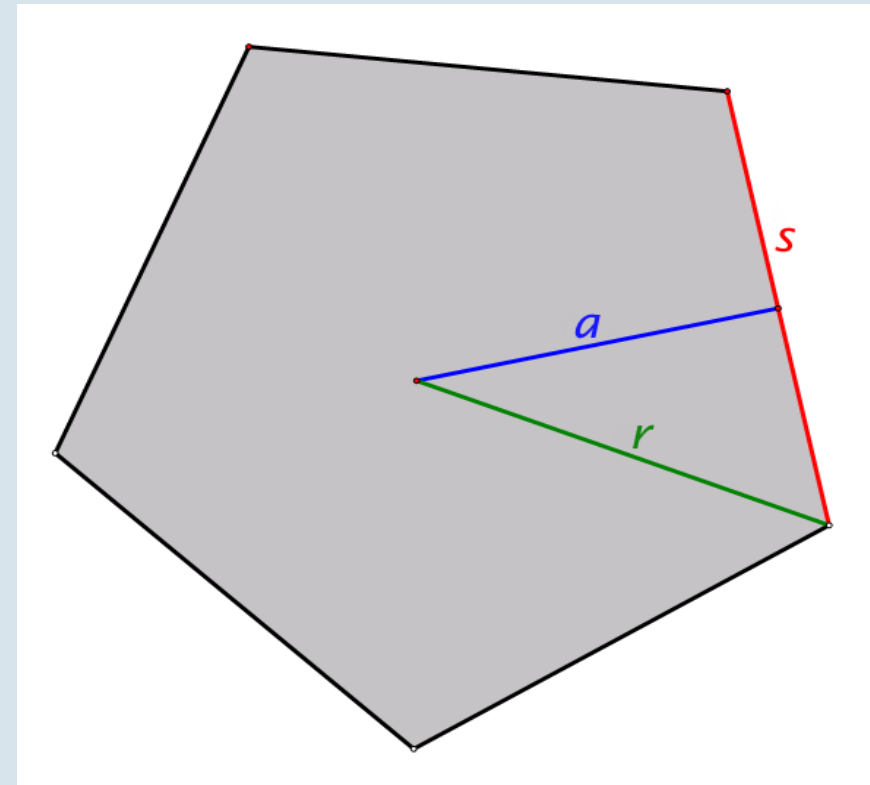
# Polígonos Regulares

- Defina em Java uma classe `RegularPolygon`, cujos objectos representam polígonos regulares. A sua classe deverá permitir a construção de um polígono regular com um número arbitrário de lados, bem como o cálculo da sua área e perímetro.
- Programe a sua classe no BlueJ
- Teste vários objectos da classe `RegularPolygon` e verifique que se comportam tal como esperado.



# Polígonos regulares

- Todos os vértices de um polígono regular estão inscritos numa circunferência de raio  $r$
- Todos os lados e ângulos do polígono regular são iguais
- A apótema mede a distância mais curta do centro geométrico ao lado do polígono
- Qualquer polígono regular pode ser caracterizado por 3 medidas:
  - Lado ( $s$ )
  - Raio da circunferência envolvente ( $r$ )
  - Apótema ( $a$ )
- Pre:
  - $r > 0$
  - $a > 0$
  - $s > 0$



# Polígonos regulares

- O perímetro  $P$  de um polígono regular de  $N$  lados de comprimento  $L$  é dado por:

$$P = n * s$$

- A área é dada por:

$$A = \frac{1}{2} * a * s * n$$

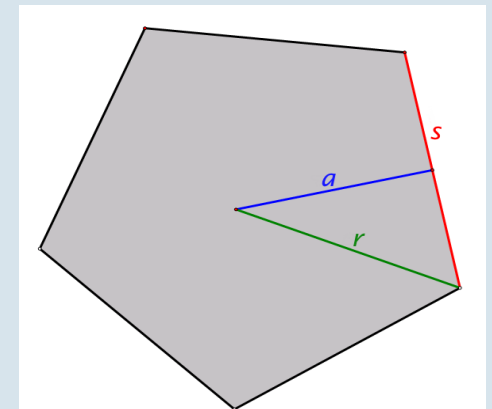
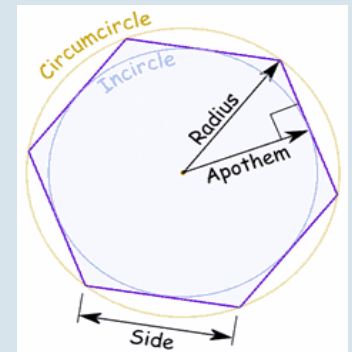
em que:

$a$  = apótema (raio do círculo inscrito no polígono)

$$\text{Apótema} = \text{Raio} \times \cos(\pi/n)$$

$s$  = comprimento do lado

$r$  = raio do círculo circunscrito



# Polígonos regulares

```
RegularPolygon t = new RegularPolygon(1.0, 3);  
t.perimeter()  
5.196152422706632 (double)  
t.area();  
1.2990381056766582 (double)  
RegularPolygon q = new RegularPolygon(1.0, 4);  
t.perimeter()  
6.92820323 (double)  
t.area()  
2.0 (double)
```



# 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.



USA	1 USD		6.430
EURO	1 EUR		8.368
SVERIGE	100 SEK		92.440
DANMARK	100 DKK		112.29
STORBRIANNIA	1 GBP		120.47
SVEITS	100 CHF		53.130
JAPAN	100 JPY		60.900
AUSTRALIA	1 AUD		
CANADA			



# 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 (GBP) e Dólares (USD)
  - Considere as taxas de conversão
    - 1 USD = 0.70 EUR
    - 1 EUR = 0.69 GBP
- Operações (interface de `CurrencyConverter`):
  - `int` `getInEuros()`
  - `void` `setInEuros(int amount)`
  - `int` `getInDollars()`
  - `void` `setInDollars(int amount)`
  - `int` `getInPounds()`
  - `void` `setInPounds(int amount)`

# Conversor

**int** getInEuros ()

Consultar a importância em euros

**void** setInEuros (**int** amount)

Definir a importância em Euros

**int** getInDollars ()

Consultar a importância em dólares

**void** setInDollars (**int** amount)

Definir a importância em dólares

**int** getInPounds ()

Consultar a importância em libras

**void** setInPounds (**int** amount)

Definir a importância em libras

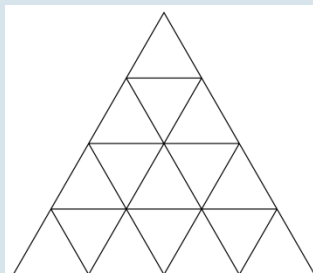
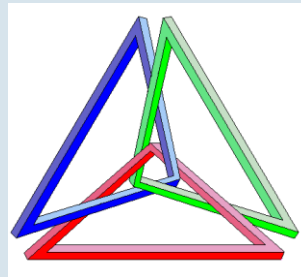
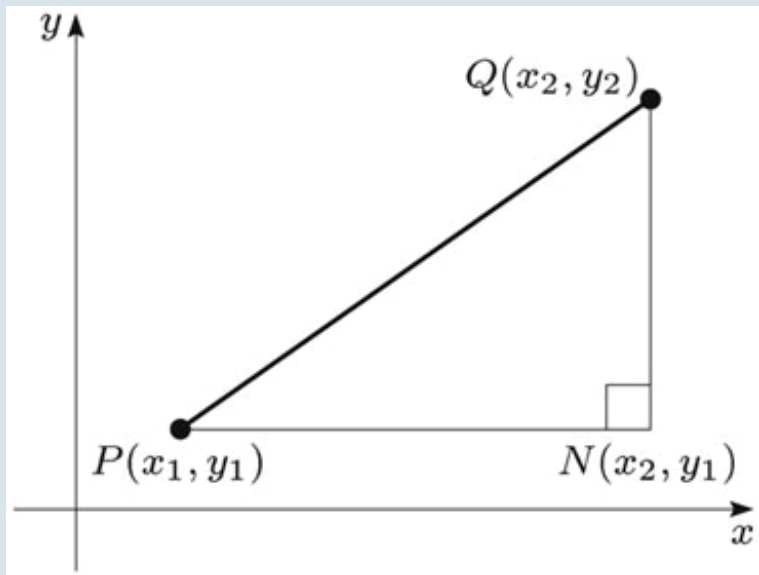
# Conversor

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

# Triângulo

# Triângulo

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



# Triângulo

- Cada objecto Triang
  - Representa um triângulo no plano, definido pelos seus três vértices
  - As coordenadas devem ser de precisão muito grande
- Pretende-se construir objectos Triang de **duas** maneiras:
  - Indicando:
    - A abcissa do primeiro vértice Px
    - A ordenada do primeiro vértice Py
    - A abcissa do segundo vértice Qx
    - A ordenada do segundo vértice Qy
    - A abcissa do terceiro vértice Rx
    - A ordenada do terceiro vértice Ry
  - Não indicando nada:  
o triângulo é criado com os seguintes vértices por defeito: (1, 1), (4, 4), (5, 1)  
(conveniente para testes)



# Triângulo

- Métodos de acesso (*getter methods*) de Triang:

**double** getPx()

devolve a abcissa do primeiro vértice

**double** getPy()

devolve a ordenada do primeiro vértice

**double** getQx()

devolve a abcissa do segundo vértice

**double** getQy()

devolve a ordenada do primeiro vértice

**double** getRx()

devolve a abcissa do terceiro vértice

**double** getRy()

devolve a ordenada do terceiro vértice

# Triângulo

- Operações fornecidas por Triang:

**double** sideLength1 ()

devolve o comprimento do 1º lado (P-Q)

**double** sideLength2 ()

devolve o comprimento do 2º lado (Q-R)

**double** sideLength3 ()

devolve o comprimento do 3º lado (R-P)

**double** perimeter ()

devolve o perímetro

**double** area ()

devolve a área, segundo a fórmula de Heron

# Triângulo

- Operações fornecidas por Triang

$$Lado_{P-Q} = \sqrt{(P_X - Q_X)^2 + (P_Y - Q_Y)^2}$$

$$Semiperímetro: S = \frac{Lado_1 + Lado_2 + Lado_3}{2}$$

$$Área = \sqrt{S(S - Lado_1)(S - Lado_2)(S - Lado_3)}$$

# Triângulo

- Operações fornecidas por Triang

Pode haver outras operações que seriam convenientes existirem em Triang, para evitar repetições dos mesmos cálculos em pontos distintos

Nas situações em detecte uma dessas operações, implemente-a e utilize-a na implementação de outras operações de modo a evitar repetição de código

# Triângulo

```
Triang t = new Triang();  
t.getX1()  
1.0      (double)  
t.getY2()  
4.0      (double)  
t.sideLength1()  
4.242640687119285      (double)  
t.sideLength2()  
3.1622776601683795      (double)  
t.semiPerimeter()  
5.702459173643832      (double)  
t.area()  
5.9999999999999999      (double)
```

# Cálculo do IVA

# Cálculo do IVA

- Defina em Java uma classe `VATCalculator` cujos objectos permitem o cálculo do IVA para as taxas existentes (em tempos) em Portugal.
- Programe a sua classe no BlueJ.
- Teste um (ou vários) objectos `VATCalculator`, e verifique se se comportam como esperado.



VAT Calculator

☒ Stay on top

To calculate VAT (Value Added Tax) at its current rate of 17.5%

**Calculate VAT**

£ 100 Calculate £ 14.89 VAT £ 85.11 Net

**Add VAT**

£ 100 Calculate £ 17.50 VAT £ 117.50 Inc VAT

This software is freeware and not for resale.

<http://www.alexnolan.net>



# Cálculo de IVA

- Cada objecto ou serviço transaccionado em Portugal implica a cobrança Imposto sobre o valor acrescentado (IVA).
- Um empresário que fornece serviços deve cobrar IVA aos seus clientes, que depois entrega ao estado.
- Se, no processo dos seus negócios, adquirir bens ou serviços, pode descontar o IVA gasto naquele que tem de entregar ao estado.
- É por isso importante saber o valor de IVA gasto ao longo de um determinado período.
- A cada valor líquido transaccionado aplica-se uma taxa de IVA, dependendo do tipo do produto ou serviço em causa
  - O IVA é calculado multiplicando o valor líquido pela taxa
- O valor bruto da transacção calcula-se somando o valor líquido ao valor do IVA



# Cálculo de IVA

- Cada objecto VATCalculator
  - Representa uma calculadora de IVA segundo três taxas: mínima, média e máxima
- Um objecto VATCalculator é criado:
  - Indicando a taxa mínima, média e máxima de IVA e aplicar e cada categoria de transacção
  - Sem indicar nada, o que implica a utilização das taxas conhecidas (5%, 12% e 20%)

# Cálculo de IVA

- Operações reconhecidas (interface de VATCalculator) :

**void** addValueMin(**double** netAmount)

Adição de valor gasto líquido (sem IVA) dentro da taxa mínima.

Pre: netAmount > 0

**void** addValueMed(**double** netAmount)

Adição de valor gasto líquido (sem IVA) dentro da taxa média.

Pre: netAmount > 0

**void** addValueMax(**double** netAmount)

Adição de valor gasto líquido (sem IVA) dentro da taxa máxima.

Pre: netAmount > 0

# Cálculo de IVA

- Operações reconhecidas (métodos de consulta) :

**double** getVATTotal()

Método que devolve o valor total de IVA calculado.

**double** getNetTotal()

Método que devolve o valor total líquido transaccionado.

**double** getGrossTotal()

Método que devolve o valor total bruto transaccionado.

**double** getVATMin()

Método que devolve o valor de IVA dentro da taxa mínima.

**double** getVATMed()

Método que devolve o valor de IVA dentro da taxa média.

**double** getVATMax()

Método que devolve o valor de IVA dentro da taxa máxima.

# Cálculo de IVA

```
VATCalculator v=new VATCalculator(0.05,0.12,0.20);  
v.addValueMin(253);  
v.addValueMed(3100);  
v.addValueMax(500);  
v.getVATMin()  
12.65      (double)  
v.getVATMed()  
372.0      (double)  
v.getVATMax()  
100.0      (double)  
v.getVATTot()  
484.65     (double)  
v.getNetTotal()  
3853.0     (double)  
  
v.getGrossTotal()  
4337.65    (double)  
v.addValueMed(405);  
v.getVATTot()  
533.25     (double)  
v.getNetTotal()  
4258.0     (double)  
v.getGrossTotal()  
4791.25    (double)
```