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, Jácome Cunha, 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 Eclipse 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.

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.
```

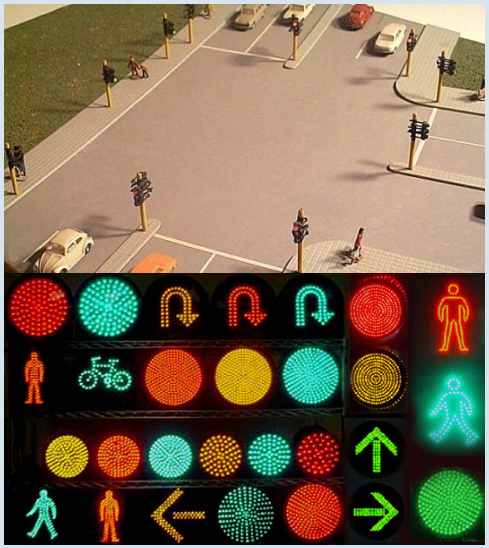
Semáforo

```
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 Eclipse.
- 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
`int` `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 Eclipse 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.

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), kilómetros 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

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), kilómetros 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:
  - 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.

	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

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();
// 0.003600000000000003   (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 Eclipse
- Teste vários objectos da classe `Speed` e verifique que se comportam tal 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: $\mathbf{P=a*(b - T)^c}$
  - [onde **T** é o Tempo em segundos;
    - e.g. 10.43 para 100 metros]
    - Saltos: $\mathbf{P=a*(M - b)^c}$
    - [onde **M** é a medida em centímetros;
    - e.g. 808 para o salto em comprimento]
    - Lançamentos: $\mathbf{P=a*(D - b)^c}$
    - [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**
  - Registar 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**
  - Registar 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)
```

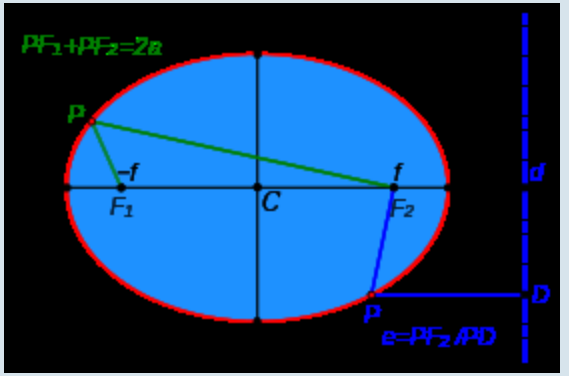
Decatlo

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


Elipse

Elipse

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



Elipse

- **Objectivo**
  - Manipular elipses no plano XY com centro na origem (0,0).
- **Descrição**

Qualquer elipse no plano XY com centro na origem define-se por dois pontos no plano (par de coordenadas XY) indicando os focos; e os comprimentos dos raios (valores reais positivos maior que zero), denotados por raios maior e menor..

- **Funcionalidades**
  - Pode-se deslocar a elipse ao longo de um dado vector <dx,dy>.
  - Deve ser sempre possível consultar o valor/módulo da abcissa e da ordenada dos focos e a dimensão dos raios maior e menor. Para além disso, deve-se consultar também a área da elipse..
  - Deve ser sempre possível verificar se um dado ponto no plano XY está no interior da elipse.

…

Elipse

- Se não indicarmos nada, a elipse será um crículo em que os focos são o centro na origem do plano XY, e os raios maior e menor são iguais e unitários. Em alternativa, pode-se:
  - Indicar:
    - Um dos focos, já que o outro é simétrico tendo como base o centro (0,0)
    - Os raios maior e menor.

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

Elipse

- Operações reconhecidas (interface do Ellipse):

```
double getArea()
double getMajorRadius()
double getMinorRadius()
double getXFocus()
double getYFocus()
boolean ptInEllipse(double x, double y)
void translate(double dx, double dy)
```

Elipse

- **Interface do Ellipse**

double `getArea()`
Devolve a área da elipse

double `getMaiorRadius()`
Devolve o comprimento do raio maior da elipse

double `getMinorRadius()`
Devolve o comprimento do raio menor da elipse

double `getXFocus()`
Devolve a abcissa (módulo) dos focos

double `getYFocus()`
Devolve a ordenada (módulo) dos focos

boolean `ptInEllipse(double x, double y)`
Indica se o ponto (x,y) se encontra no interior da elipse

void `translate(double dx, double dy)`
Desloca a elipse segundo o vector <dx,dy>

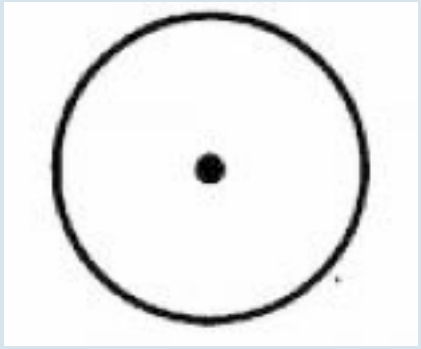
Elipse

```
 Ellipse c = new Ellipse();
 c.getMajorRadius();
//  1.0   (double)
 c.getArea();
//  3.141592653589793   (double)
 c.translate(1,1);
 c.getArea();
//  3.141592653589793   (double)
 c.ptInEllipse(-1,-1);
//  false   (boolean)
 c.getXFocus();
//  1.0   (double)
 c.getYFocus();
//  1.0   (double)
```

Círculo

Círculo

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



Círculo

- **Objectivo**
  - Manipular circulos no plano XY.
- **Descrição**

Qualquer círculo no plano XY define-se por um ponto no plano (par de coordenadas XY) indicando o centro; e o comprimento do raio (valor real positivo maior que zero).

- **Funcionalidades**
  - Pode-se deslocar o círculo ao longo de um dado vector <dx,dy>.
  - Deve ser sempre possível consultar a abcissa e a ordenada do centro e a dimensão do raio. Para além disso, deve-se consultar também o perímetro e a área do círculo.
  - Deve ser sempre possível verificar se um dado ponto no plano XY está no interior do círculo.

…

Círculo

- Se não indicarmos nada, o círculo terá o centro na origem do plano XY com raio unitário. Em alternativa, pode-se:
  - Indicar:
    - O centro
    - O raio
  - Indicar
    - O raio, sendo o centro na origem do plano XY.

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

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

Interface do Circle

`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

```
  Public class Circle {
    ... ;
    public Circle() {...; }
    public Circle(double radius) { ... ; }
/* Pre: radius > 0) */
    public Circle(double cx, double cy, double radius) {
        ... ;
    }
/* Pre: radius > 0) */
    ...
  }
```

Múltiplos Construtores

```
  public class Ellipse {
    ... ;
    public Ellipse() {...; }
    public Ellipse(double fx, double fy, double majorRadius,
double minorRadius) {
       ... ;
    }
/* Pre: majorRadius > 0 && minorRadius > 0) */
    ...
  }
```

Mars Rover

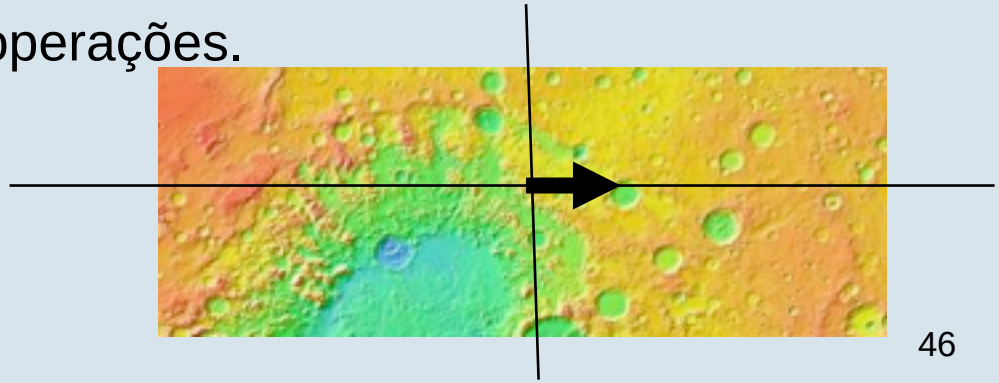
Mars Rover

- **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 (= 0º)

- **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.

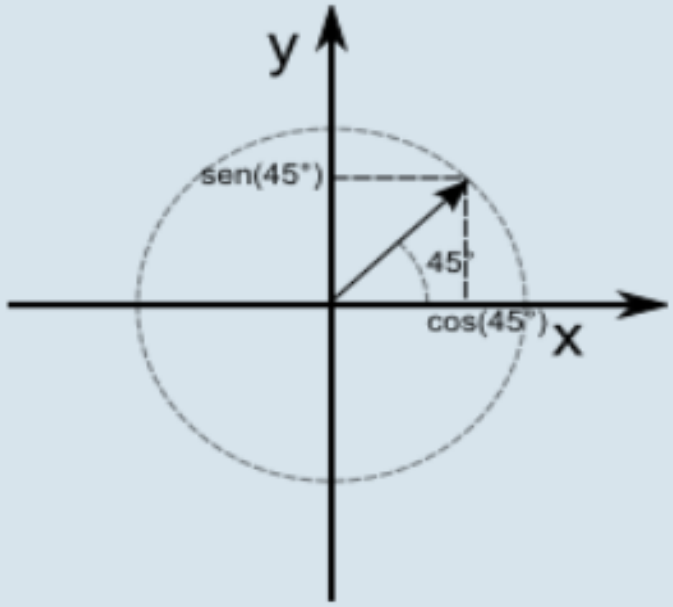
Mars Rover

- **Dicas**

  Nova posição em X = posição anterior em X + (distancia) * cos(angle);
  Nova posição em Y = posição anterior em Y + (distancia) * sin(angle);

Funções da biblioteca Math necessárias:

- Math.sin(radianos) - seno
- Math.cos(radianos) - cosseno
- Math.toRadians(graus) – converte graus em radianos



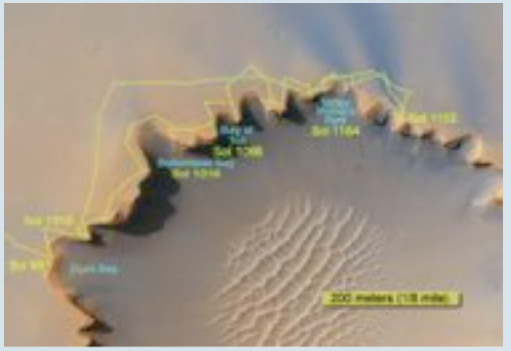
Mars Rover

```
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)
```

Mars Rover

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

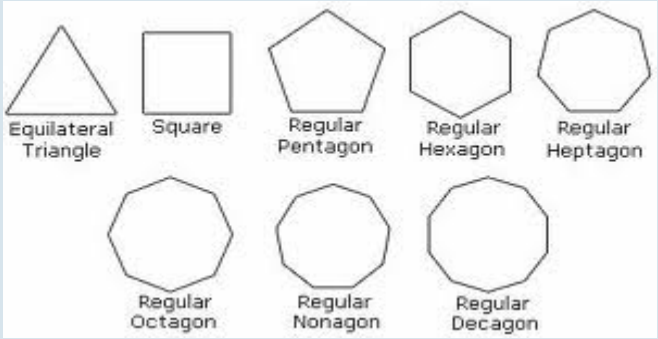


Outros exercícios

Polígonos regulares

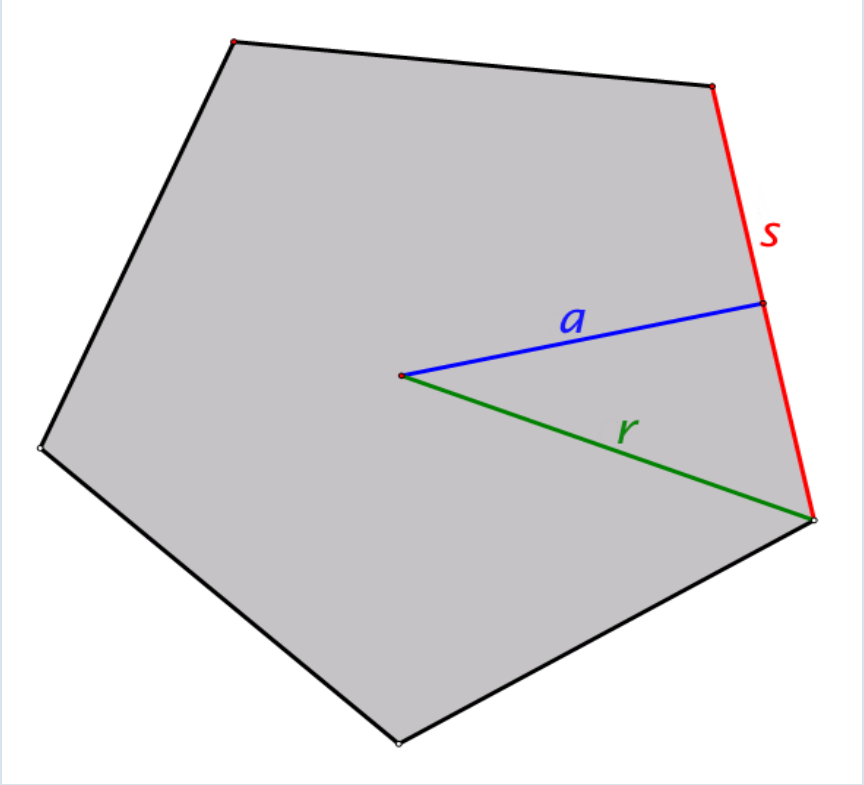
Polígonos Regulares

- 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 Eclipse
- 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:

$A = ½ * a * s * n$

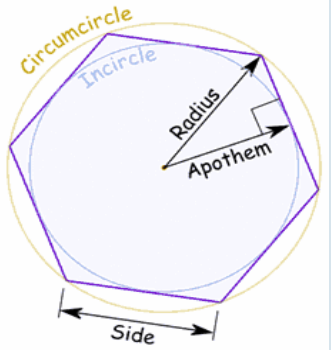
em que:

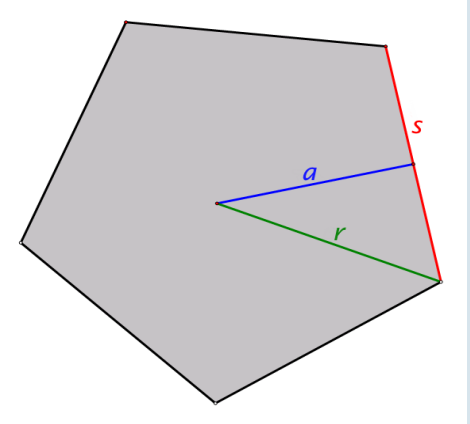
a = apótema (raio do círculo inscrito no polígono)

Apótema = Raio × cos(π/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 Eclipse.
- 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`):

```
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

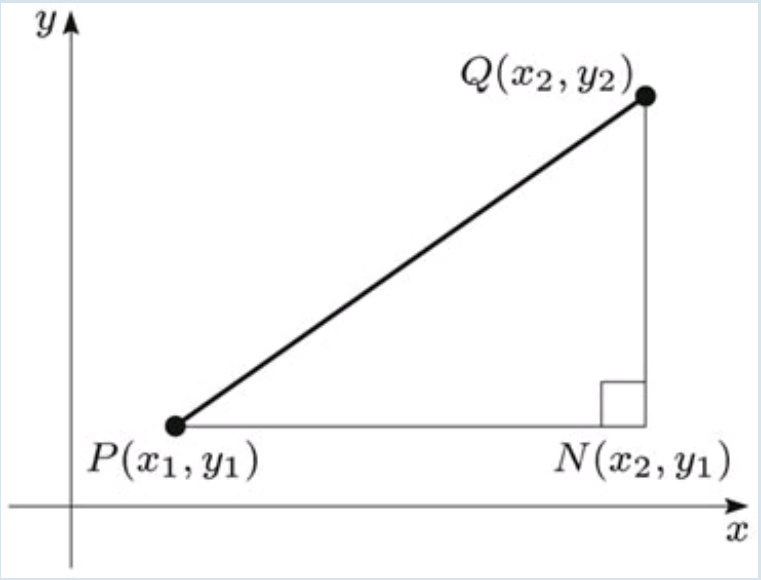
Conversor

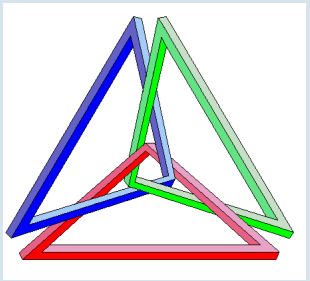
```
 CurrencyConverter c = new CurrencyConverter();
 c.setInEuros(100);
 c.getInEuros();
//  100  (int)
 c.getInDollars();
//  143  (int)
 c.getInPounds();
//  69  (int)
 c.setInPounds(200);
 c.getInEuros();
//  290  (int)
 c.getInDollars();
//  414  (int)
 c.getInPounds();
//  200  (int)
```

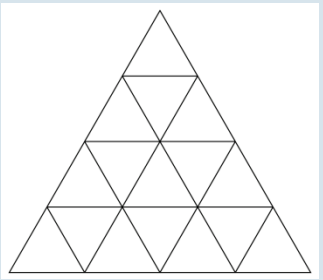
Triângulo

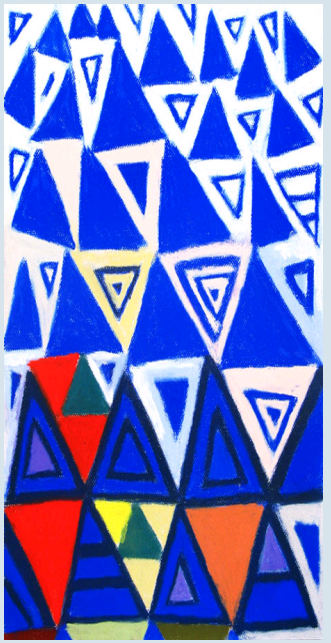
Triângulo

- Defina em Java uma classe Triang cujos objectos representam triângulos num plano.
- Programe a sua classe no Eclipse.
- 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

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

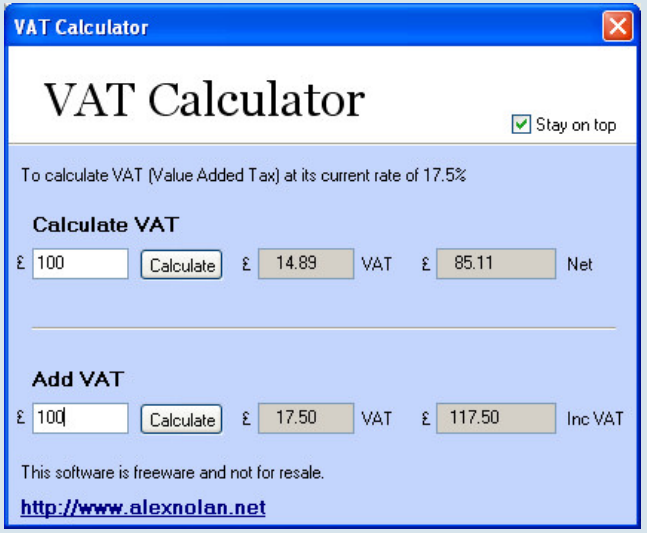
Triângulo

```
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();
// 5.999999999999999   (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 Eclipse.
- Teste um (ou vários) objectos VATCalculator, e verifique se se comportam como esperado.



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.getVATTotal();
//  484.65   (double)
 v.getNetTotal();
//  3853.0   (double)
 v.getGrossTotal();
//  4337.65   (double)
 v.addValueMed(405);
 v.getVATTotal();
//  533.25   (double)
 v.getNetTotal();
//  4258.0   (double)
 v.getGrossTotal();
//  4791.25   (double)
```