

Condições e Decisões

Lista de exercícios

**Material didático 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:
 - Usar o Eclipse na construção de classes simples, definindo:
 - Constantes, variáveis de instância
 - Múltiplos construtores de instâncias da classe, operações modificadoras e de consulta, usando:
 - Chamadas a operações da classe dentro de outras operações
 - Operações aritméticas na implementação de métodos
 - Operações da biblioteca Math
 - Operações lógicas na implementação dos métodos
 - Variáveis locais/temporárias dentro dos métodos
 - **Instruções de controle: if-then-else ; switch**
 - **tipo String**
 - Dada uma especificação de uma classe em língua natural, modelar a classe, recorrendo aos elementos acima descritos

Dieta

(novas funcionalidades)

Recorde a 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

Novas funcionalidades

- **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, e **o valor máximo de calorias ingeridas ou eliminadas numa refeição ou exercício, respectivamente.**
- 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

int balance()

Devolve o saldo total de calorias

Dieta

- Interface (classe Diet):

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

int maxEatenCallories()

Devolve o valor máximo de calorias ingeridas numa só refeição

pre: eatTimes() > 0

int maxBurntCallories()

Devolve o valor máximo de calorias consumidas num só exercício

pre: burnTimes() > 0

Novas operações



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.maxEatenCalories()  
100    (int)  
d.maxBurntCalories()  
40     (int)
```


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.



Estação Meteorológica

Estação Meteorológica

- **Objectivo**
 - Manipular valores de temperaturas ao longo do tempo.
- **Descrição**
 - Numa estação meteorológica regista-se valores de temperaturas (valores reais).
- **Funcionalidades**
 - É sempre possível registar um dado valor de temperatura.
 - É necessário saber sempre o número de temperaturas registadas, poder consultar os valores médio, máximo e mínimo das temperaturas registadas.
 - Quando é criada, não existem temperaturas registadas.
- **Interacção com o utilizador**
 - Após criar uma estação meteorológica, pode invocar as operações.

Estação Meteorológica

- Interface (classe WeatherStation)

void sampleTemperature(**double** temp)

Registrar a amostra temp na estação

int numberTemperatures()

Consultar o número de temperaturas registadas até ao momento

double getMaximum()

Consultar a máxima temperatura observada até ao momento

pre: numberTemperatures() > 0

double getMinimum()

Consultar a mínima temperatura observada até ao momento

pre: numberTemperatures() > 0

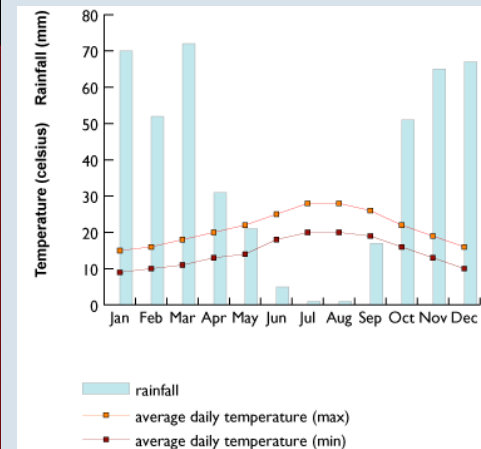
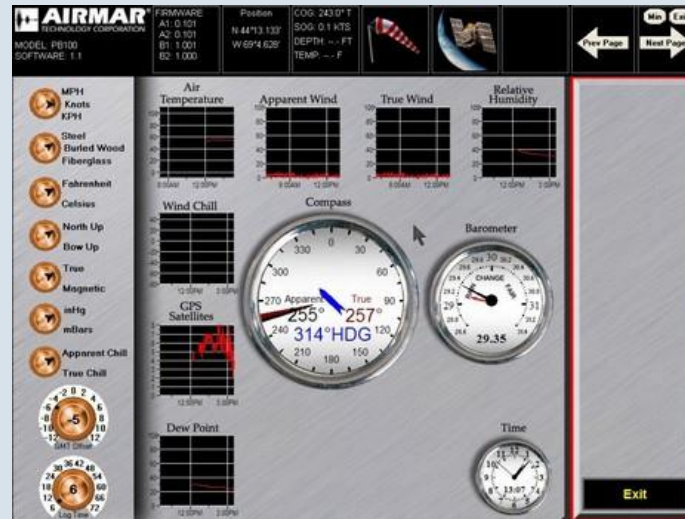
double getAverage()

Consultar a média das temperaturas observadas até ao momento

pre: numberTemperatures() > 0

Estação Meteorológica

- Defina em Java uma classe `WeatherStation` cujos objectos representam uma estação meteorológica.
- Programe a sua classe no BlueJ.
- Teste um (ou vários) objectos `WeatherStation`, e verifique se se comportam como se esperada.



Estação Meteorológica

```
WeatherStation alaska = new WeatherStation();  
alaska.numberTemperatures();  
0    (int)  
alaska.sampleTemperature(10.0);  
alaska.getMinimum()  
10.0  (double)  
alaska.sampleTemperature(12.0);  
alaska.sampleTemperature(-13.0);  
alaska.sampleTemperature(16.0);  
alaska.sampleTemperature(9.0);  
alaska.getMaximum()  
16.0  (double)  
alaska.getMinimum()  
-13.0  (double)  
alaska.getAverage()  
6.8    (double)  
alaska.numberTemperatures();  
5      (int)
```

Stand de Limonada

Recorde o Stand de Limonada

- **Objectivo**

- Gerir as vendas da limonada e o stock de ingredientes de um stand de limonada.

- **Descrição**

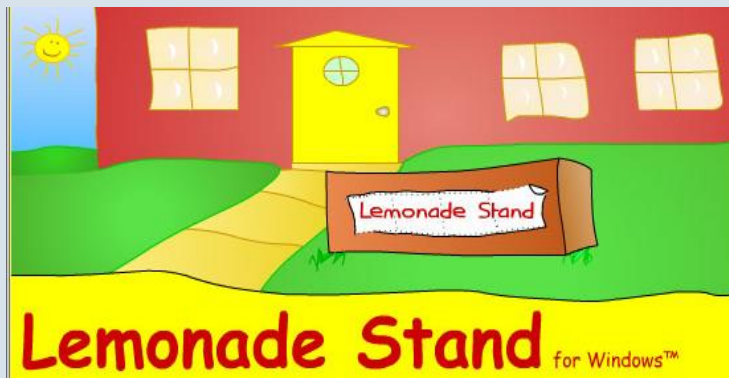
- Os únicos ingredientes utilizados são limões, açúcar e água.
- A receita de tanto sucesso é utilizar 5 limões, 100g de açúcar e 1l de água por cada jarro, que dá para 5 copos.
- Cada copo é vendido por 0.75€. No entanto, se uma pessoa comprar 5 ou mais copos de uma vez, o preço baixa para 0.65€.
- O espaço do stand é limitado, pelo que não é possível ter em stock mais do que 50 limões, 5kg de açúcar e 30l de água.

- **Funcionalidades**

- Uma pessoa pode sempre comprar uma dada quantidade de copos de limonada. Caso não exista em stock quantidade suficiente de ingredientes, é possível encomendar uma certa quantidade de um ingrediente para repor os stocks. Esta quantidade encomendada fica logo disponível. Note no entanto que nunca poderá exceder o espaço limitado do stand.

Recorde o Stand de Limonada

- É sempre possível consultar a quantidade em stock dos ingredientes: limões, açúcar e água. Assim como a quantidade usada, até ao momento, de cada um dos ingredientes.
 - É necessário poder consultar o número total de copos vendidos (tanto a preço normal como a preço reduzido), bem como o valor total das vendas (dinheiro em caixa).
 - Quando é criado um stand de limonada, deve-se encomendar os ingredientes de modo ao stock ficar cheio (50 limões, 5kg de açúcar e 30l de água).
- **Interacção com o utilizador**
 - Após criar um stand de limonada, pode invocar as operações.



Stand de Limonada

- Programe em Java uma classe `LemonadeStand`.
- Teste um (ou vários) objectos, e verifique se se comportam como se espera.



Tempo

Tempo

- **Objectivo**
 - Manipular quantidades de tempo.
- **Descrição**
 - O tempo indica uma quantidade de tempo com precisão ao nível do segundo.
- **Funcionalidades**
 - É possível somar e subtrair quantidades de tempo. Assim como somar e/ou subtrair a uma dada quantidade de tempo, um valor em segundos, minutos ou horas.
 - Deve-se poder consultar os segundos “ss”, minutos “mm” e horas “hh” da quantidade de tempo representada em “hh:mm:ss”. Assim como a sua especificação textual “hh horas mm minutos ss segundos”.
 - É possível converter a quantidade de tempo para um valor em segundos.
 - Quando é criada uma quantidade de tempo, pode-se indicar as “hh”, “mm” e “ss”. Caso não se indique nada a quantidade de tempo é 00:00:00.
- **Interacção com o utilizador**
 - Após criar uma quantidade de tempo, pode-se invocar as operações.

Tempo

- Interface (classe Time):

int getSeconds()

Consulta "ss" da quantidade de tempo

int getMinutes()

Consulta "mm" da quantidade de tempo

int getHours()

Consulta "hh" da quantidade de tempo

int seconds()

Calcula o valor em segundos associado à quantidade de tempo

void addSeconds(**int** s)

Adiciona s segundos à quantidade de tempo

pre: s > 0

void addMinutes(**int** m)

Adiciona m minutos à quantidade de tempo

pre: m > 0

Tempo

- Interface (classe Time):

void addHours(**int** h)

Adiciona h horas à quantidade de tempo

pre: s > 0

void subtractSeconds(**int** s)

Subtrai s segundos à quantidade de tempo

pre: s > 0 && s <= seconds()

void subtractMinutes(**int** m)

Subtrai m minutos à quantidade de tempo

pre: m > 0 && m*60 <= seconds()

void subtractHours(**int** h)

Subtrai h horas à quantidade de tempo

pre: h > 0 && h*60*60 <= seconds()

String asString()

Consulta descrição textual associada à quantidade de tempo

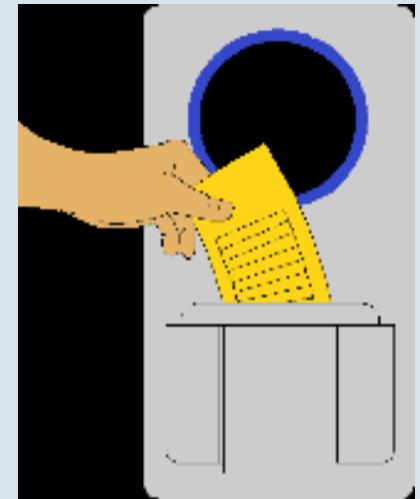
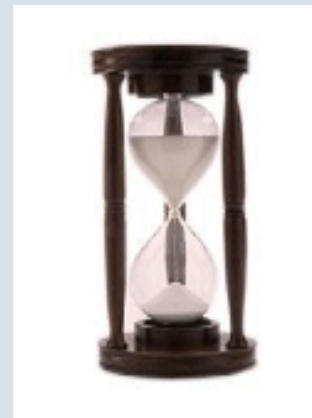
Tempo

```
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)
```

Tempo

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

A close-up photograph of a clock face with a warm, orange-brown tint. The numbers 11, 5, and 6 are prominently displayed in a large, white, sans-serif font, representing the time 11:56. The clock hands are visible but blurred in the background.



Jogo da Adivinha


Jogo da Adivinha

- **Objectivo**
 - Adivinhar números.
- **Descrição**
 - Um jogo da adivinha consiste em adivinhar o número (valor inteiro) gerado num dado intervalo de valores.
- **Funcionalidades**
 - O jogador pode sempre tratar de adivinhar o número. Caso o número proposto pelo jogador seja maior ou menor que o número pretendido, o jogo indica, respectivamente essa informação: “Too high!” ou “Too low!”. Se o número proposto for igual ao pretendido, o jogo termina com a informação “You win!”. Caso o número não faça sentido, pois já foi excluído por jogadas anteriores, o jogo termina com a informação “Game Over”.
 - O jogador tem sempre a possibilidade de gerar um novo número para adivinhar.
 - Quando é criado, gera-se um número aleatório num dado intervalo de valores indicado.
- **Interacção com o utilizador**
 - Após criar um jogo, o jogador pode tratar de adivinhar o número.

Jogo da Adivinha

- Interface (classe `GuessWhat`)

tipo `String`



```
String tryit(int guess)
```

O jogador usa este método para apostar que o número secreto é `guess`.

O método `tryit` responde (devolve)

“You win!” se o número proposto é o certo

“Too high!” se o número proposto é maior que o secreto

“Too low!” se o número proposto é menor que o secreto

“Game Over” se o número proposto não faz sentido ... Pois já foi
excluído por jogadas anteriores

```
void newGame ()
```

O jogador gera um novo número para adivinhar.

Jogo da Adivinha

```
GuessWhat game = new GuessWhat(0,9);
```

```
game.tryit(5)
```

```
"Too low!" (String)
```

```
game.tryit(8)
```

```
"Too high!" (String)
```

```
game.tryit(2)
```

```
"Game Over" (String)
```

```
game.newGame();
```

```
game.tryit(5)
```

```
"Too high!" (String)
```

```
game.tryit(3)
```

```
"Too low!" (String)
```

```
game.tryit(4)
```

```
"You win!" (String)
```

Intervalo de pesquisa:

segredo ≥ 0 && segredo ≤ 9

segredo ≥ 6 && segredo ≤ 9

segredo ≥ 6 && segredo ≤ 7

2 já foi excluído por jogadas anteriores: o jogo termina!

segredo ≥ 0 && segredo ≤ 9

segredo ≥ 0 && segredo ≤ 4

segredo ≥ 4 && segredo ≤ 4

Sucesso! Adivinhou o número

Adivinha o Número Secreto

- Defina em Java uma classe `GuessWhat` cujos objectos são jogos de adivinhar números.
- Programe a sua classe no Eclipse.
- Teste um (ou vários) objectos `GuessWhat`, e verifique se se comportam como se espera.



Como gerar um número aleatório?

- Para programar a classe `GuessWhat` vai necessitar de gerar um número aleatório
- O Java oferece, numa das suas bibliotecas, uma classe chamada `Random`, que permite criar um objecto especializado na geração de números aleatórios
- Para usar uma classe de biblioteca, temos de declarar que a queremos usar, usando a instrução de importação `import`
- A classe `Random` tem, entre outras operações, um construtor `Random` e uma operação `nextInt` que recebe um valor inteiro `n` e devolve um inteiro entre 0 e `n-1`

```
import java.util.Random;

public class GuessWhat{
    private int intervalMin; // Limite mínimo do intervalo
    private int intervalMax; // Limite máximo do intervalo
    //...
    private int genSecret() {
        int dist = intervalMax - intervalMin + 1;
        Random r = new Random(); // Inicialização do gerador de aleatórios
        return r.nextInt(dist) + intervalMin; //Uso da operação nextInt
    }
    //...
}
```

Simulador do IRS

Simulador de IRS

- **Objectivo**

- Simular o cálculo do IRS.

- **Descrição**

- O IRS é um imposto a pagar, que corresponde à aplicação de uma taxa, menos uma parcela a abater. A taxa a aplicar depende do escalão de imposto consoante o montante da matéria colectável (aqui chamado rendimento bruto).
- A tabela do cálculo do IRS a pagar em 2009 foi a seguinte:

Taxas gerais 2009					
Escalões				Taxa	Parcela a abater
1º	até 4.755			10,50%	-
2º	De mais	4.755	até 7.192	13,00%	118,87
3º	De mais	7.192	até 17.836	23,50%	874,03
4º	De mais	17.836	até 41.021	34,00%	2.746,81
5º	De mais	41.021	até 59.450	36,50%	3.772,33
6º	De mais	59.450	até 64.110	40,00%	5.853,08
7º	superior a 64.110			42,00%	7.135,28

Simulador de IRS

- **Funcionalidades**

- Deve ser possível calcular:
 - escalão aplicável ao rendimento bruto (1-7)
 - taxa de IRS correspondente ao escalão aplicável
 - montante a pagar de imposto (cálculo de IRS): Note que para as parcelas a abater, deve ser usada somente a parte inteira (118, 874, 2746, 3772, 7135)
 - valor líquido restante após a cobrança do IRS
- Quando é criado, indica-se o rendimento bruto.

- **Interacção com o utilizador**

- Após criar um simulador de IRS, pode invocar as operações.

Defina a interface da classe `IRSSimulator`

Simulador do IRS

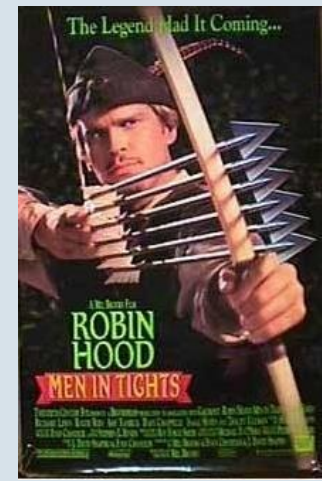
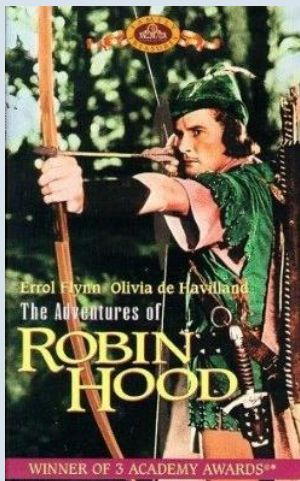
- Defina em Java uma classe IRSSimulator cujos objectos permitem o cálculo do IRS (Imposto sobre Pessoas Singulares) para os escalões existentes em Portugal (2009).
- Programe a sua classe no Eclipse.
- Teste um (ou vários) objectos IRSSimulator, e verifique se se comportam como esperado.



Robin Hood (aka, Archer)

Robin Hood (aka Archer)

- Defina em Java uma classe `Archer` cujos objectos são os arqueiros do lendário Robin Hood. Estes arqueiros assaltam os homens do xerife de Nottingham, mas têm de comprar as flechas para os atacar...
- Programe a sua classe no Eclipse.
- Teste um (ou vários) objectos `Archer`, e verifique se se comportam como se espera.



Robin Hood (aka, Archer)

- Cada arqueiro caracteriza-se pelo número de flechas que ainda tem disponível, a sua pontaria (um número real), e pelo dinheiro que lhe resta. O preço das flechas é constante e igual a 5 euros.
- Quando o arqueiro é criado:
 - Se não dissermos nada, o arqueiro começa com 0 flechas, 1.0f de pontaria e 20 euros);
 - Em alternativa, podemos indicar os valores iniciais para o número de flechas, a pontaria e o dinheiro.
- O arqueiro pode disparar flechas contra alvos que se encontram a uma determinada distância (leia-se, contra os malvados cobradores do Xerife de Nottingham):
 - se acertar, recebe um prémio em dinheiro (ou seja, fica com o dinheiro que cobrador do Xerife leva) e aumenta a sua pontaria;
 - se falhar, não aumenta nem o dinheiro nem a pontaria.

Robin Hood (aka, Archer)

- Operações reconhecidas:

```
public int getArrows()
```

```
public int getMoney()
```

```
public int howManyArrowsCanBuy()
```

```
public void buyArrows(int howMany)
```

```
public void fireArrow(int distance, int prizeMoney)
```

```
public int successfulShot(int distance)
```

```
public boolean canFireArrow()
```

```
public boolean canContinuePlaying()
```

Robin Hood (aka, Archer)

- Operações reconhecidas (interface de Archer):

public int getArrows()

Consultar o número de flechas que restam.

public float getAccuracy()

Consultar a pontaria do arqueiro.

public int getMoney()

Consultar dinheiro que lhe resta.

public int howManyArrowsCanBuy()

Consultar quantas flechas tem dinheiro para comprar. Lembre-se que as flechas custam 5 euros cada. Use uma constante, por favor.

Robin Hood (aka, Archer)

- Operações reconhecidas (interface de Archer):

```
public void buyArrows(int howMany)
```

Comprar flechas na quantidade indicada. Cada flecha custa 5 Euros. Se acha estranho que, já no Século XII, as flechas custassem 5 Euros, note que, para além de dinheiro, o Robin Hood também roubava Vinho da Madeira aos homens do Xerife de Nottingham, que o produzia a partir de maçãs dos seus pomares... 😊

Pre: howMany >=0

```
public void fireArrow(int distance, int prizeMoney)
```

Dispara a flecha a uma determinada distância do alvo. Cada tiro custa 1 flecha. Para saber se o tiro é ou não certo, use a operação `successfulShot()`. Se acertar:

- Adiciona o prémio, em dinheiro, à sua bolsa
- Aumenta a sua pontaria em metade da distância para o alvo

Quer acerte, quer falhe, gasta sempre uma flecha

Pre: distance >= 0 && prizeMoney >= 0

Robin Hood (aka, Archer)

- Operações reconhecidas (interface de Archer):

public int successfulShot(**int** distance)

Calcula o sucesso do tiro. Um tiro tem sucesso se a razão entre a pontaria do arqueiro e a distância for maior ou igual a $0.5f$. A função retorna `0` se o tiro falhou, `1` se acertou. Veja no próximo slide como programar esta operação.

Pre: `distance >= 0`

public boolean canFireArrow()

Indica se o arqueiro ainda tem flechas para disparar.

public boolean canContinuePlaying()

Indica se o arqueiro pode continuar na sua actividade de salteador. A função deve retornar `true` se o arqueiro ainda tiver flechas, ou dinheiro suficiente para as comprar. Caso contrário, retorna `false`.

Robin Hood (aka, Archer)

- **public int** successfulShot(**int** distance)

- O arqueiro só acerta se a razão entre o valor da sua pontaria e a distância até ao alvo for maior ou igual a 0.5f
- A função retorna:
 - 0 se o tiro falhou
 - 1 se acertou.

- Usar:

- operações aritméticas
- `Math.round(expr)`
- `Math.min(expr1, expr2)`
 - devolve o mínimo entre as duas expressões

Só acerto se a razão entre a minha pontaria e a distância for maior ou igual a 0.5f



Robin Hood (aka, Archer)

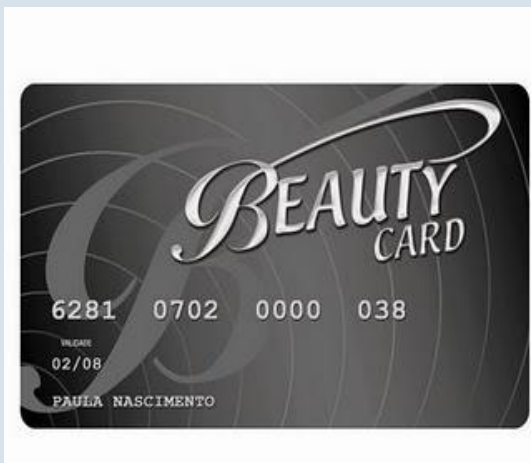
```
Archer robin = new Archer();  
robin.getArrows()  
0 (int)  
robin.getAccuracy()  
1.0 (float)  
robin.getMoney()  
20 (int)  
robin.buyArrows(4);  
robin.canContinuePlaying()  
true (boolean)  
robin.fireArrow(2000,3);  
robin.getArrows()  
3 (int)  
robin.getAccuracy()  
1.0 (float)  
robin.fireArrow(1000,3);  
robin.getAccuracy()  
1.0 (float)
```

```
robin.fireArrow(100,3);  
robin.getAccuracy()  
1.0 (float)  
robin.fireArrow(1,3);  
robin.getAccuracy()  
1.5 (float)  
robin.getArrows()  
0 (int)  
robin.getMoney()  
3 (int)  
robin.canContinuePlaying()  
false (boolean)  
robin.canFireArrow()  
false (boolean)  
robin.successfulShot(2)  
1 (int)  
robin.successfulShot(4)  
0 (int)
```

Beauty Card

Beauty Card

- Este exercício **não é para programar, mas para analisar**. Definir a interface.
- No entanto, se quiser, poderá programá-lo como exercício adicional.



Beauty Card

- Especifique uma classe para gerir os cartões de pontos de um salão de beleza
- Um cartão de pontos é um cartão no qual, para cada tratamento de beleza, a cliente recebe um determinado número de pontos
- Existem tratamentos de mãos, pés, cabelo e rosto
- Quando a cliente acumula 60 pontos em quaisquer tratamentos, pode trocar esses pontos por uma massagem num *Spa* associado ao salão de beleza
- Quando a cliente acumula pelo menos...
 - 20 pontos em tratamentos para mãos
 - 25 pontos em tratamentos para pés
 - 50 pontos em tratamentos para rosto
 - 30 pontos em tratamentos para cabelo
- ... pode ganhar uma semana de aulas com um personal trainer, num ginásio próximo do salão de beleza, em troca desses pontos

Beauty Card

- Cada BeautyCard é pessoal e intransmissível, caracterizando-se por:
 - nome de cliente, representado como String,
 - pontos acumulados em tratamentos para as mãos, pés, rosto e cabelos, representados como inteiros