

Programa Principal

Leitura e Escrita de dados

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

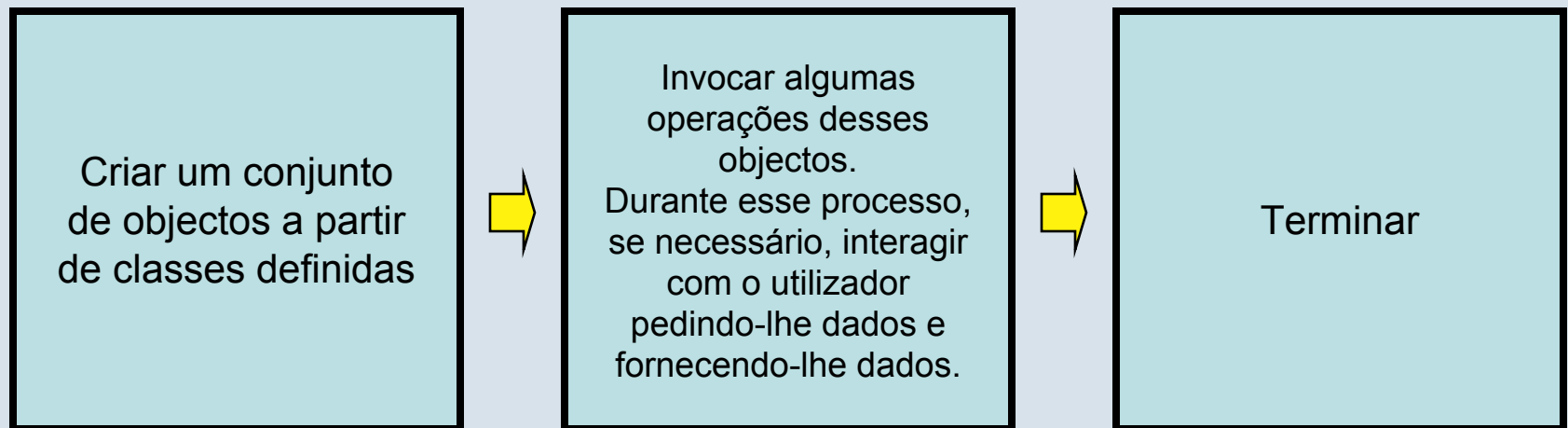
Luís Caires (Responsável), Armanda Rodrigues, António Ravara, Carla Ferreira, Fernanda Barbosa, Fernando Birra, Jácome Cunha, João Araújo, Miguel Goulão, Miguel Pessoa Monteiro, e Sofia Cavaco.

Mestrado Integrado em Engenharia Informática FCT UNL

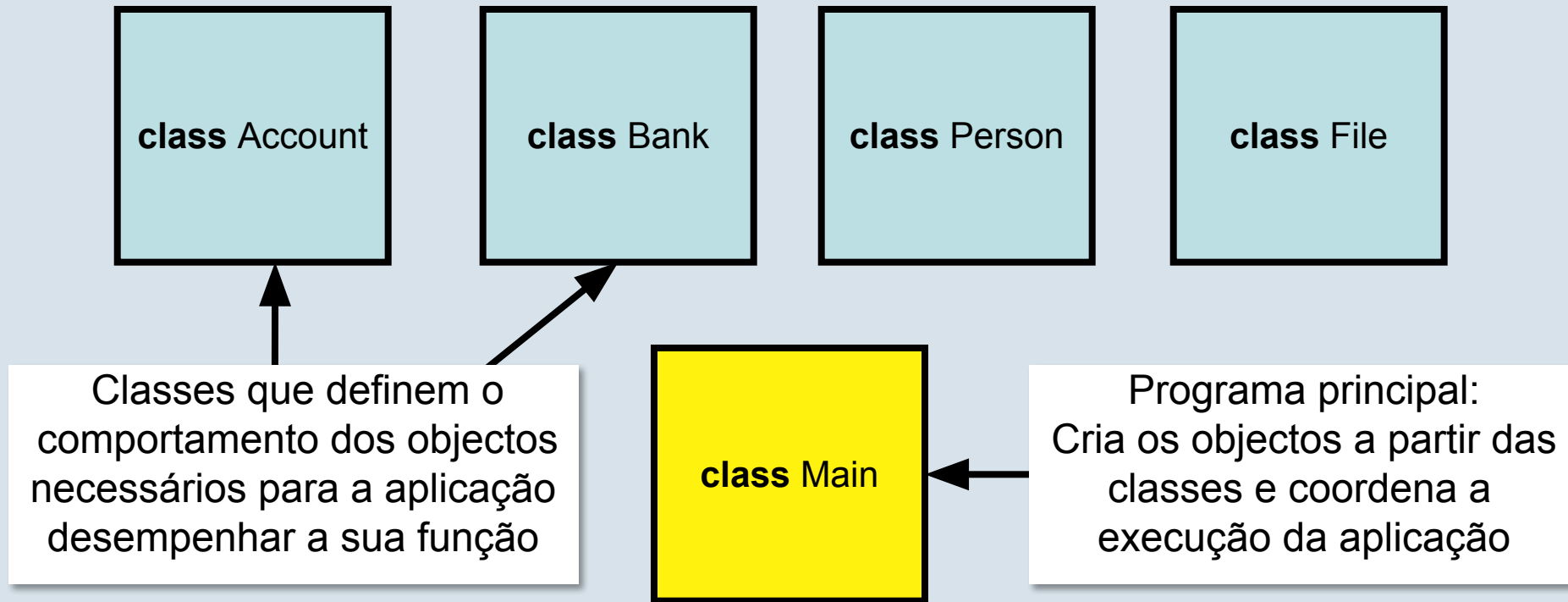
Programa Principal Simples

Programa Principal Simples

- Os programas mais simples têm sempre a seguinte estrutura de tarefas em sequência



Estrutura de uma Aplicação Simples



- Os componentes de qualquer aplicação são sempre **um conjunto de classes** e um **programa principal**, que coordena a execução do sistema de software

Programa Principal Simples

- Em Java, o programa principal de uma aplicação define-se numa classe especial (a que vamos chamar `Main`), definida à parte das outras classes
- Na classe `Main` define-se um método especial, *estático*, chamado `main`, com o programa principal

```
public class Main {
```

```
    public static void main(String[] args) {
```

```
    }
```

```
}
```

Modificador de acesso `static`

Num método indica que este é da classe e não dos objectos.

Programa principal

Quando a aplicação iniciar a sua execução, são as instruções aqui colocadas que irão ser executadas

Programa Principal Simples

- Exemplo (Lâmpadas)
 - O programa principal de uma aplicação simples que cria duas lâmpadas e realiza algumas operações

```
public class Main {  
    public static void main(String[] args) {  
        Lamp l1 = new Lamp();  
        Lamp l2 = new Lamp();  
        l1.on();  
        l2.on();  
        l1.off();  
    }  
}
```

Programa Principal Simples

- Exemplo (Conta)
 - O programa principal de uma aplicação simples que cria uma conta bancária segura e realiza algumas operações

```
public class Main {  
    public static void main(String[] args) {  
        SafeBankAccount b = new SafeBankAccount(1000);  
        b.deposit(20);  
        b.applyInterest();  
    }  
}
```

Programa Principal Simples

- Exemplo (Contas)
 - O programa principal de uma aplicação simples que cria duas contas e realiza uma transferência entre elas

```
public class Main {  
    public static void main(String[] args) {  
        SafeBankAccount b1 = new SafeBankAccount (1000) ;  
        SafeBankAccount b2 = new SafeBankAccount (2000) ;  
        int someAmount = 20 ;  
        b1.withdraw (someAmount) ;  
        b2.deposit (someAmount) ;  
    }  
}
```

Note que os programas principais de exemplo apresentados até aqui não produzem resultados visíveis para o utilizador.

Output de texto

- Exemplo (Output)

A operação `System.out.println` pode ser usada no programa principal para escrever mensagens ao utilizador

```
public class Main {  
    public static void main(String[] args) {  
        String name = "Luis";  
        int age = 20;  
        System.out.println(name);  
        System.out.println("The age is "+age+", not 42");  
        System.out.println("Pi="+Math.PI);  
    }  
}
```

Programa Principal Simples

- Exemplo (Contas e Output)
 - O programa principal de uma aplicação simples que cria duas contas e realiza uma transferência entre elas

```
public class Main {  
    public static void main(String[] args) {  
        SafeBankAccount b1 = new SafeBankAccount(1000);  
        SafeBankAccount b2 = new SafeBankAccount(2000);  
        int someAmount = 20;  
        b1.withdraw(someAmount);  
        b2.deposit(someAmount);  
        System.out.println( b1.getBalance() );  
        System.out.println( b2.getBalance() );  
    }  
}
```

Princípio arquitectural

- Em qualquer aplicação que se programe é muito importante separar claramente
 - As partes responsáveis pela interacção com o utilizador
 - As partes responsáveis pela funcionalidade dos vários objectos (definidas em várias classes)
- Assim, o uso de operações de output (ou input, como veremos mais tarde) é apenas permitido no interior da classe `Main` ou nas classes responsáveis pela interacção. Esta regra é **muito importante**.
- Por exemplo, o uso de `System.out.println` numa classe de aplicação, como `SafeBankAccount` ou `Calculator` será **sempre proibido** nas disciplinas de programação do MIEI.

Recorde a Conta Bancária Segura

- **Objectivo**

- Simular uma conta bancária segura.

- **Descrição**

- Uma conta bancária é um “depósito” de dinheiro (valor inteiro em cêntimos). O saldo pode ser positivo (credor ou nulo) ou negativo (devedor), e é sempre um valor inteiro em cêntimos.

- **Funcionalidades**

- Numa conta pode-se depositar e levantar dinheiro. Deve ser sempre possível consultar o saldo da conta e verificar se a conta tem um saldo devedor.
- O levantamento é seguro, pois só pode ser efectuado caso o valor a levantar seja inferior ou igual ao valor do saldo. Qualquer tentativa de levantamento de um valor sem provisão na conta leva à aplicação de uma multa de 2 Euros.
- Usualmente, o banco credita um certo juro nas contas dos seus clientes, numa base periódica (por exemplo, uma vez por ano). O valor do juro é calculado por aplicação de uma taxa ao valor do saldo, ou seja, a taxa de juro é determinada com base no valor do saldo, através de um sistema de escalões (quanto maior o saldo, maior a taxa).

Recorde a Conta Bancária Segura

- Deve ser sempre possível calcular o valor de juro anual a aplicar ao saldo da conta. Existem três taxas possíveis:

Valor do Saldo	Taxa
$\leq 2000 \text{ €}$	1%
$]2000 \text{ €, } 10.000 \text{ €}]$	2%
$> 10.000 \text{ €}$	3%

- Deve ser sempre possível creditar os juros no saldo da conta.
- Se não indicarmos nada, a conta é criada com saldo zero. Em alternativa, podemos indicar um valor inicial para o saldo.
- **Interacção com o utilizador**
 - Após criar uma conta bancária segura, pode invocar as operações da conta.

Recorde a Conta Bancária Segura

- Interface de SafeBankAccount:

void deposit(**int** amount)

Depositar a importância amount na conta

Pre: amount > 0

void withdraw(**int** amount)

Levantar a importância amount na conta ou aplicar a multa (**FINE**)

Pre: amount > 0

int getBalance()

Consultar o saldo da conta

boolean redZone()

Indica se a conta está devedora

int computeInterest()

Calcular qual o valor do juro anual a aplicar

void applyInterest()

Creditar o juro anual ao saldo da conta

Nova interacção com o utilizador

- Desenvolver um programa que:
 - Pede ao utilizador o saldo inicial duma conta bancária.
 - Pede o valor a levantar na conta. Informa caso não seja possível efectuar o levantamento.
 - O levantamento só será efectuado com sucesso, se o saldo da conta for igual ou superior ao valor a levantar;
 - Note que, caso não seja efectuado o levantamento será sempre aplicada uma multa (recorde a definição de conta segura).
 - Lista o saldo corrente da conta.

Leitura de dados

- `System.in` é um objecto pré-definido utilizado para fazer leitura de dados a partir do *standard input* (o teclado, por omissão)
- No entanto, o `System.in` só lê um carácter de cada vez
- No Java, a classe `Scanner` foi criada para fazer a leitura a partir do teclado de uma forma simples
- O `Scanner` é inicializado estabelecendo uma ligação a uma *stream* de entrada de dados (neste caso, o `System.in`)
 - Indicamos a *stream* a que queremos aceder no construtor

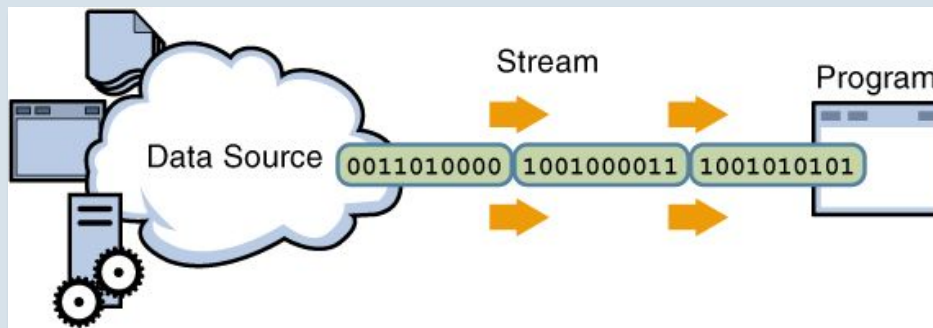
```
Scanner in = new Scanner(System.in);
```

- Quando já não necessitamos do `Scanner`, devemos libertar a *stream* usando o método `close()` do `Scanner`, que encerra o acesso desse `Scanner` à *stream* indicada no construtor

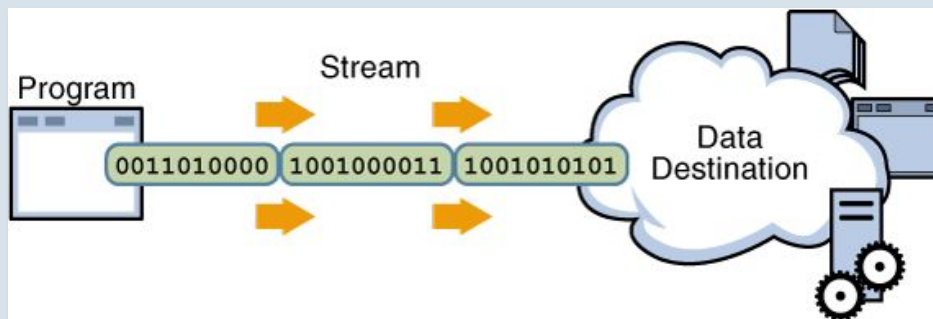
```
in.close();
```

System.*in* e System.*out*

- Os objectos System.*in* e System.*out* representam duas *streams* de entrada e saída de dados do programa, respectivamente
- Uma *stream* é um fluxo de dados de onde podemos ler ou onde podemos escrever informação.
 - System.*in* é a stream de entrada de dados usada por omissão no sistema



- System.*out* é a stream de saída de dados usada por omissão no sistema



Disciplina de uso do Scanner

- Como a classe `Scanner` não faz parte do *core* do Java, temos que a importar para a usar

```
import java.util.Scanner;
```

- Antes de ler ou escrever dados numa *stream*, temos de lhe aceder; depois devemos “libertar” o acesso

```
Scanner in = new Scanner(System.in); // Scanner acede a System.in
System.out.println("Quantidade: "); // Informacao pretendida
...
in.close(); // Ja nao necessitamos do Scanner, devemos liberta-lo
```

Operações da classe Scanner

- **int** `nextInt()` lê um inteiro (`int`)
- **float** `nextFloat()` lê um real (`float`)
- **double** `nextDouble()` lê um real (`double`)
- `String next()` lê uma `String` sem espaços
- `String nextLine()` lê uma linha até ao fim (`String`)
- **void** `close()` liberta os recursos do `Scanner`, desactivando-o

```
Scanner in = new Scanner(System.in); // Scanner acede a System.in
System.out.println("Quantidade: "); // Informacao pretendida
int value = in.nextInt(); //inteiro lido fica guardado em value
in.close(); // Ja nao necessitamos do Scanner, devemos liberta-lo
```

Estude a classe `Scanner` no pacote `java.util`

Construção da classe Main

```
import java.util.Scanner;
```

Temos que indicar onde está definida a classe

```
public class Main {
```

Scanner

```
    public static void main(String[] args) {
```

```
        //Cria uma conta e tenta fazer levantamento.
```

```
        Scanner in = new Scanner(System.in);
```

```
        SafeBankAccount account;
```

```
        int balance, value;
```

```
        System.out.print("Saldo inicial: ");
```

```
        balance = in.nextInt();
```

Temos que ler tudo o que está para além do inteiro (nomeadamente a mudança de linha)

```
        in.nextLine();
```

```
        //Criar conta bancaria com saldo inicial = balance
```

```
        if (balance >= 0) { //Verificar a pre-condicao de SafeBankAccount
```

```
            account = new SafeBankAccount(balance);
```

```
            //tentar levantar dinheiro
```

```
            System.out.print("Valor a levantar em centimos:");
```

```
            value = in.nextInt();
```

```
            in.nextLine();
```

```
            ...
```

```
        }
```

Devemos encerrar o Scanner in, libertando assim os seus recursos, quando terminamos as leituras de dados.

```
        in.close();
```

```
    }
```

```
}
```

Construção da classe Main

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        ...
        //Criar conta bancaria com saldo inicial = balance
        //Verificar a pré-condição de SafeBankAccount
        if (balance >= 0) {
            ...
            if (value > 0){ //Verificar a pre-condicao de withdraw
                if (account.getBalance() < value) //O saldo nao e suficiente.
                    System.out.println("Levantamento nao efectuado.");

                // Chama-se withdraw mesmo quando o saldo nao e suficiente
                // para a multa ser aplicada
                account.withdraw(value);
                System.out.println( "Saldo da conta=" + account.getBalance()
                                    + " centimos");
            }
        }
        in.close();
    }
}
```

Nova interacção com o utilizador (transferência bancária)

- Desenvolver um programa que:
 - Pede ao utilizador o saldo inicial de duas contas bancárias.
 - Pede um valor a transferir e tenta executar uma transferência bancária de uma conta para a outra.
 - A transferência só será efectuada com sucesso, se o saldo da conta de origem for igual ou superior ao valor a transferir;
 - Note que, caso não seja efectuada a transferência NÃO será aplicada uma multa.
 - Lista o saldo corrente da conta.

Primeira tentativa de construção da classe Main

Cria **duas** contas, com os saldos iniciais dados pelo utilizador...

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        SafeBankAccount accountOrigin = new SafeBankAccount(); //Comeca a 0
        System.out.println("Conta Origem");
        System.out.println("Saldo inicial em centimos:");
        int initialOrigin = in.nextInt(); // Le saldo inicial
        in.nextLine();
        if (initialOrigin > 0) // Se for positivo
            accountOrigin.deposit(initialOrigin); // Deposita o saldo inicial
        SafeBankAccount accountDestiny = new SafeBankAccount(); //Comeca a 0
        System.out.println("Conta Destino");
        System.out.println("Saldo inicial em centimos:");
        int initialDestiny = in.nextInt(); // Le saldo inicial
        in.nextLine();
        if (initialDestiny > 0) // Se for positivo
            accountDestiny.deposit(initialDestiny); // Deposita o saldo inicial
        System.out.println("Valor a transferir em centimos:");
        int value = in.nextInt();
        in.nextLine();
        if (accountOrigin.getBalance() >= value && value >= 0){
            accountOrigin.withdraw(value);
            accountDestiny.deposit(value);
            System.out.println("Transferencia efectuada com sucesso");
        } else
            System.out.println("Transferencia nao efectuada");
        System.out.print("Saldo conta origem: ");
        System.out.println(accountOrigin.getBalance() + " Centimos");
        System.out.print("Saldo conta destino: ");
        System.out.println(accountDestiny.getBalance() + " Centimos");
        in.close();
    }
}
```

... e depois lê o valor a transferir e transfere-o de accountOrigin para accountDestiny

Funciona, mas não é uma boa solução...

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        SafeBankAccount accountOrigin = new SafeBankAccount(); //Comeca a 0
        System.out.println("Conta Origem");
        System.out.println("Saldo inicial em centimos:");
        int initialOrigin = in.nextInt(); // Le saldo inicial
        in.nextLine();
        if (initialOrigin > 0) // Se for positivo
            accountOrigin.deposit(initialOrigin); // Deposita o saldo inicial
        SafeBankAccount accountDestiny = new SafeBankAccount(); //Comeca a 0
        System.out.println("Conta Destino");
        System.out.println("Saldo inicial em centimos:");
        int initialDestiny = in.nextInt(); // Le saldo inicial
        in.nextLine();
        if (initialDestiny > 0) // Se for positivo
            accountDestiny.deposit(initialDestiny); // Deposita o saldo inicial
        System.out.println("Valor a transferir em centimos:");
        int value = in.nextInt();
        in.nextLine();
        if (accountOrigin.getBalance() >= value && value >= 0){
            accountOrigin.withdraw(value);
            accountDestiny.deposit(value);
            System.out.println("Transferencia efectuada com sucesso");
        } else
            System.out.println("Transferencia nao efectuada");
        System.out.print("Saldo conta origem: ");
        System.out.println(accountOrigin.getBalance() + " Centimos");
        System.out.print("Saldo conta destino: ");
        System.out.println(accountDestiny.getBalance() + " Centimos");
        in.close();
    }
}
```

Cria **duas contas, com os saldos iniciais dados pelo utilizador...**

Este método está a ficar complicado. Já viu bem o tamanho de fonte que tivemos de usar para ele caber num único slide? :-(
O problema é que este método faz demasiadas tarefas...

... e depois lê o valor a transferir e transfere-o de accountOrigin para accountDestiny

Vamos usar uma lupa...

Cria **duas** contas, com os saldos iniciais dados pelo utilizador...

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        SafeBankAccount accountOrigin = new SafeBankAccount(); //Comeca a 0
        System.out.println("Conta Origem");
        System.out.println("Saldo inicial em centimos:");
        int initialOrigin = in.nextInt();    // Le saldo inicial
        in.nextLine();
        if (initialOrigin > 0)                // Se for positivo
            accountOrigin.deposit(initialOrigin); // Deposita o saldo inicial
        SafeBankAccount accountDestiny = new SafeBankAccount(); //Comeca a 0
        System.out.println("Conta Destino");
        System.out.println("Saldo inicial em centimos:");
        int initialDestiny = in.nextInt();    // Le saldo inicial
        in.nextLine();
        if (initialDestiny > 0)                // Se for positivo
            accountDestiny.deposit(initialDestiny); // Deposita o saldo inicial
        ...
    }
}
```

... e depois lê o valor a transferir e transfere-o de accountOrigin para accountDestiny

Temos bastante código repetido

O código de criação das duas contas é bastante semelhante.

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        SafeBankAccount accountOrigin = new SafeBankAccount(); //Comeca a 0
        System.out.println("Conta Origem");
        System.out.println("Saldo inicial em centimos:");
        int initialOrigin = in.nextInt();    // Le saldo inicial
        in.nextLine();
        if (initialOrigin > 0)                // Se for positivo
            accountOrigin.deposit(initialOrigin); // Deposita o saldo inicial
        SafeBankAccount accountDestiny = new SafeBankAccount(); //Comeca a 0
        System.out.println("Conta Destino");
        System.out.println("Saldo inicial em centimos:");
        int initialDestiny = in.nextInt();    // Le saldo inicial
        in.nextLine();
        if (initialDestiny > 0)                // Se for positivo
            accountDestiny.deposit(initialDestiny); // Deposita o saldo inicial
        ...
    }
```

Podemos e devemos evitar código desnecessariamente repetido.

Código repetido é mesmo necessário?

O código de criação das duas contas é bastante semelhante.

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);

        SafeBankAccount accountOrigin = new SafeBankAccount(); //Comeca a 0
        System.out.println("Conta Origem");
        System.out.println("Saldo inicial em centimos:");
        int initialOrigin = in.nextInt();    // Le saldo inicial
        in.nextLine();
        if (initialOrigin > 0)                // Se for positivo
            accountOrigin.deposit(initialOrigin); // Deposita o saldo inicial

        SafeBankAccount accountDestiny = new SafeBankAccount(); //Comeca a 0
        System.out.println("Conta Destino");
        System.out.println("Saldo inicial em centimos:");
        int initialDestiny = in.nextInt();    // Le saldo inicial
        in.nextLine();
        if (initialDestiny > 0)                // Se for positivo
            accountDestiny.deposit(initialDestiny); // Deposita o saldo inicial
        ...
    }
}
```

E se fundirmos estes dois excertos num único método auxiliar?
Alguns **detalhes** terão de ser tratados, claro.

Podemos criar um método auxiliar

Podemos declarar um método
createAccount (...) que construa a conta

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        SafeBankAccount origin = createAccount(in, "origem");
        SafeBankAccount destiny = createAccount(in, "destino");
        ...
        in.close();
    }

    private static SafeBankAccount createAccount(Scanner in, String name) {
        SafeBankAccount account = new SafeBankAccount(); //Começa a 0
        System.out.println("Conta " + name);
        System.out.println("Saldo inicial em centimos:");
        int initial = in.nextInt(); // Le saldo inicial
        in.nextLine();
        if (initial > 0)                // Se for positivo
            account.deposit(initial); // Deposita o saldo inicial
        return account;
    }
}
```

Este método auxiliar deve ser **privado**. Necessitamos de um parâmetro para o Scanner (*in*) e outro com o nome da conta (*name*).

Invocação do método auxiliar

Invocamos `createAccount(...)` para construir duas novas contas.

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        SafeBankAccount origin = createAccount(in, "origem");
        SafeBankAccount destiny = createAccount(in, "destino");
        ...
        in.close();
    }

    private static SafeBankAccount createAccount(Scanner in, String name) {
        SafeBankAccount account = new SafeBankAccount(); //Começa a 0
        System.out.println("Conta " + name);
        System.out.println("Saldo inicial em centimos:");
        int initial = in.nextInt(); // Le saldo inicial
        in.nextLine();
        if (initial > 0) // Se for positivo
            account.deposit(initial); // Deposita o saldo inicial
        return account;
    }
}
```

Em vez de um método “gigante” `main`, podemos definir métodos auxiliares privados, como `createAccount(...)`

Continuando a olhar o `main` à lupa...

Pede o valor a transferir, efectua a transferência e dá feedback ao utilizador.

```
public class Main {  
    public static void main(String[] args) {  
        ...  
        System.out.println("Valor a transferir em centimos:");  
        int value = in.nextInt();  
        in.nextLine();  
        if (accountOrigin.getBalance() >= value && value >= 0) {  
            accountOrigin.withdraw(value);  
            accountDestiny.deposit(value);  
            System.out.println("Transferencia efectuada com sucesso");  
        } else {  
            System.out.println("Transferencia nao efectuada");  
        }  
        System.out.print("Saldo conta origem: ");  
        System.out.println(accountOrigin.getBalance()+ " Centimos");  
        System.out.print("Saldo conta destino: ");  
        System.out.println(accountDestiny.getBalance()+ " Centimos");  
        in.close();  
    }  
}
```

Continuamos a ver várias tarefas misturadas e algum código repetido

Várias oportunidades de melhoria

Pede o valor a transferir, efectua a transferência e dá feedback ao utilizador.

```
public class Main {  
    public static void main(String[] args) {  
        ...  
        System.out.println("Valor a transferir em centimos:");  
        int value = in.nextInt();  
        in.nextLine();  
        if (accountOrigin.getBalance() >= value && value >= 0) {  
            accountOrigin.withdraw(value);  
            accountDestiny.deposit(value);  
            System.out.println("Transferencia efectuada com sucesso");  
        } else {  
            System.out.println("Transferencia nao efectuada");  
        }  
        System.out.print("Saldo conta origem: ");  
        System.out.println(accountOrigin.getBalance()+ " Centimos");  
        System.out.print("Saldo conta destino: ");  
        System.out.println(accountDestiny.getBalance()+ " Centimos");  
        in.close();  
    }  
}
```

Continuamos a ver várias tarefas misturadas e algum código repetido

A leitura de um valor merece um método

Pede o valor a transferir, efectua a transferência e dá feedback ao utilizador.

```
public class Main {  
    public static void main(String[] args) {  
        ...  
        System.out.println("Valor a transferir em centimos:");  
        int value = in.nextInt();  
        in.nextLine();  
        if (accountOrigin.getBalance() >= value && value >= 0) {  
            accountOrigin.withdraw(value);  
            accountDestiny.deposit(value);  
            System.out.println("Transferencia efectuada com sucesso");  
        } else {  
            System.out.println("Transferencia nao efectuada");  
        }  
        System.out.print("Saldo conta origem: ");  
        System.out.println(accountOrigin.getBalance()+ " Centimos");  
        System.out.print("Saldo conta destino: ");  
        System.out.println(accountDestiny.getBalance()+ " Centimos");  
        in.close();  
    }  
}
```

Podemos ter um método privado para pedir o valor a transferir...

O método `getValue` lê um inteiro

Pede o valor a transferir, efectua a transferência e dá feedback ao utilizador.

```
public class Main {  
    public static void main(String[] args) {  
        ...  
        int value = getValue(in);  
        ...  
        in.close();  
    }  
  
    private static int getValue(Scanner in){  
        System.out.println("Valor a transferir em centimos:");  
        int value = in.nextInt();  
        in.nextLine();  
        return value;  
    }  
}
```

Podemos ter um método privado para pedir o valor a transferir...

O método `transferMoney` transfere o valor da origem para o destino

Pede o valor a transferir, efectua a transferência e dá feedback ao utilizador.

```
public class Main {
    public static void main(String[] args) {
        ...
        int value = getValue(in);
        transferMoney(accountOrigin, accountDestiny, value);
        ...
        in.close();
    }

    private static void transferMoney(SafeBankAccount accountOrigin,
                                      SafeBankAccount accountDestiny,
                                      int value) {
        if (accountOrigin.getBalance() >= value && value >= 0) {
            accountOrigin.withdraw(value);
            accountDestiny.deposit(value);
            System.out.println("Transferencia efectuada com sucesso");
        } else
            System.out.println("Transferencia nao efectuada");
    }
}
```

Podemos ter um método privado para transferir o valor `value`...

O método `printAccount` escreve o estado de uma conta na consola

Pede o valor a transferir, efectua a transferência e dá feedback ao utilizador.

```
public class Main {
    public static void main(String[] args) {
        ...
        int value = getValue(in);
        transferMoney(accountOrigin, accountDestiny, value);
        printAccount(accountOrigin, "origem");
        printAccount(accountDestiny, "destino");
        in.close();
    }

    private static void printAccount(SafeBankAccount accountOrigin,
                                     String name) {
        System.out.print("Saldo conta "+endpoint+": ");
        System.out.println(account.getBalance()+ " Centimos");
    }
}
```

Podemos ter um método privado para escrever os saldos finais.

A classe Main completa.

Cria duas contas bancárias, pede o valor a transferir, efectua a transferência e dá feedback ao utilizador. Cada tarefa no seu respectivo método auxiliar.

```
public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        SafeBankAccount origin = createAccount(in, "origem");
        SafeBankAccount destiny = createAccount(in, "destino");
        int value = getValue(in);
        transferMoney(accountOrigin, accountDestiny, value);
        printAccount(accountOrigin, "origem");
        printAccount(accountDestiny, "destino");
        in.close();
    }
    // Metodos auxiliares:
    private static SafeBankAccount createAccount(...) {...}
    private static int getValue(...) {...}
    private static void transferMoney(...) {...}
    private static void printAccount(...) {...}
}
```

Nova interacção com o utilizador (menú de opções)

- Desenvolver um programa que:
 - Pede ao utilizador o saldo inicial de uma conta bancária.
 - Apresenta em forma de menú todas as operações possíveis de realizar sobre a conta, da seguinte forma:

- 1- Depositar
 - 2- Levantar
 - 3- Consultar saldo
 - 4- Consultar juro anual
 - 5- Creditar juro anual
 - 6- Sair
 - Pede ao utilizador a opção do menu que deseja realizar. Em caso de levantamento e depósito deve pedir também o valor a levantar ou depositar, respectivamente. Nos casos de consulta deve escrever na consola a informação respectiva.
 - Escrever na consola o saldo da conta.

Construção da classe Main

Vamos aplicar o que aprendemos com o exemplo da transferência bancária. Em vez de criarmos um método muito complicado, vamos usar vários métodos auxiliares, para facilitar o nosso trabalho de desenvolvimento. O que é que este programa deve fazer?

Cria uma conta bancária, mostra as opções disponíveis, lê a opção escolhida pelo utilizador, executa a operação e mostra o saldo da conta após a operação. Começamos por criar o esqueleto da nossa aplicação. Cada tarefa terá o seu respectivo método auxiliar.

```
public class Main {  
    public static void main(String[] args) {  
        // Cria conta bancaria  
        // Mostra o menu de opcoes  
        // Le uma opcao  
        // Executa uma operacao  
        // Mostra o saldo da conta, após a execucao da operacao  
    }  
}
```

A aplicação a construir é interactiva

Cria uma conta bancária, mostra as opções disponíveis, lê a opção escolhida pelo utilizador, executa a operação e mostra o saldo da conta após a operação.

A nossa aplicação vai necessitar de fazer leituras e escritas de dados. Vamos necessitar de um Scanner, certo? Comecemos por adicionar o Scanner:

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        // Cria conta bancaria
        // Mostra o menu de opcoes
        // Le uma opcao
        // Executa uma opcao
        // Mostra o saldo da conta, após a execucao da opcao
        in.close();
    }
}
```

Criar uma conta bancária

Cria uma conta bancária, mostra as opções disponíveis, lê a opção escolhida pelo utilizador, executa a operação e mostra o saldo da conta após a operação. Para criar uma conta, podemos reutilizar o método já definido para o exemplo da transferência bancária: `createAccount()`.

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        SafeBankAccount account = createAccount(in, "conta");
        // Mostra o menu de opcoes
        // Le uma opcao
        // Executa uma opcao
        // Mostra o saldo da conta, após a execucao da opcao
        in.close();
    }
    private static SafeBankAccount createAccount(Scanner in,
                                                String name) {
        ...
    }
}
```

Criar uma conta bancária

```
public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    SafeBankAccount account = createAccount(in, "conta");
    ...
    in.close();
}

private static SafeBankAccount createAccount(Scanner in, String name) {
    SafeBankAccount account;
    System.out.print("Saldo da conta de " + name + ": ");
    int initial = in.nextInt();
    in.nextLine();
    if (initial >= 0) {
        account = new SafeBankAccount(initial);
    } else {
        account = new SafeBankAccount();
    }
    System.out.println("Conta " + name + " criada com saldo "
        + account.getBalance());
    return account;
}
```

Mostrar as opções disponíveis

Cria uma conta bancária, mostra as opções disponíveis, lê a opção escolhida pelo utilizador, executa a operação e mostra o saldo da conta após a operação. Para mostrar as opções, necessitamos de um novo método auxiliar: `printMenu()`.

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        SafeBankAccount account = createAccount(in, "conta");
        printMenu();
        // Le uma opcao
        // Executa uma opcao
        // Mostra o saldo da conta, após a execucao da opcao
        in.close();
    }
    private static void printMenu() {
        ...
    }
}
```

Mostrar as opções disponíveis

```
private static final int DEPOSIT = 1;
private static final int WITHDRAW = 2;
private static final int BALANCE = 3;
private static final int INTEREST_RATE = 4;
private static final int CREDIT_INTEREST = 5;
private static final int EXIT = 6;

public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    SafeBankAccount account = createAccount(in, "conta");
    printMenu();
    ...
    in.close();
}

private static void printMenu() {
    System.out.println(DEPOSIT + " - Depositar");
    System.out.println(WITHDRAW + " - Levantar");
    System.out.println(BALANCE + " - Consultar saldo");
    System.out.println(INTEREST_RATE + " - Consultar juro anual");
    System.out.println(CREDIT_INTEREST + " - Aplicar juro");
    System.out.println(EXIT + " - Sair");
}
```

Ler a opção seleccionada pelo utilizador

Cria uma conta bancária, mostra as opções disponíveis, lê a opção escolhida pelo utilizador, executa a operação e mostra o saldo da conta após a operação. Para ler a operação seleccionada pelo utilizador, necessitamos de um novo método auxiliar: *readOption()*.

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        SafeBankAccount account = createAccount(in, "conta");
        printMenu();
        int option = readOption(in);
        // Executa uma opcao
        // Mostra o saldo da conta, após a execucao da opcao
        in.close();
    }
    private static int readOption(Scanner in) {
        ...
    }
}
```

Ler a opção seleccionada pelo utilizador

```
public static void main(String[] args) {  
    Scanner in = new Scanner(System.in);  
    SafeBankAccount account = createAccount(in, "conta");  
    printMenu();  
    int option = readOption(in);  
    // Executa uma opcao  
    // Mostra o saldo da conta, após a execucao da opcao  
    in.close();  
}
```

```
private static int readOption(Scanner in) {  
    int option = in.nextInt();  
    in.nextLine();  
    return option;  
}
```

Executar uma operação

Cria uma conta bancária, mostra as opções disponíveis, lê a opção escolhida pelo utilizador, executa a operação e mostra o saldo da conta após a operação. Para executar a opção escolhida, necessitamos de um novo método auxiliar: *executeOption()*.

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        SafeBankAccount account = createAccount(in, "conta");
        printMenu();
        int option = readOption(in);
        executeOption(account, option, in);
        // Mostra o saldo da conta, após a execucao da opcao
        in.close();
    }
    private static int executeOption(Scanner in) {
        ...
    }
}
```

Executar uma operação

```
public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    SafeBankAccount account = createAccount(in, "conta");
    printMenu();
    int option = readOption(in);
    executeOption(account, option, in);
    // Mostra o saldo da conta, após a execucao da opcao
    in.close();
}
```

```
private static void executeOption(SafeBankAccount account,
                                   int option, Scanner in) {
    switch (option) {
        case DEPOSIT: processDeposit(account, in); break;
        case WITHDRAW: processWithdraw(account, in); break;
        case BALANCE: processBalance(account); break;
        case INTEREST_RATE: processInterestRate(account); break;
        case CREDIT_INTEREST: processCreditInterest(account); break;
        case EXIT: break;
        default: showUnknownCommand(); break;
    }
}
```

Executar uma operação

```
public static void main(String[] args) {  
    Scanner in = new Scanner(System.in);  
    SafeBankAccount account = createAccount(in, "conta");  
    printMenu();  
    int option = readOption(in);  
    executeOption(account, option, in);  
    // Mostra o saldo da conta, após a e  
    in.close();  
}
```

Em vez de criar um método *executeOption* gigante, com a implementação de cada uma das operações codificada no corpo do respectivo caso no switch, deve criar um método auxiliar por operação.

```
private static void executeOption(SafeBankAccount account,  
                                  int option, Scanner in) {  
    switch (option) {  
        case DEPOSIT: processDeposit(account, in); break;  
        case WITHDRAW: processWithdraw(account, in); break;  
        case BALANCE: processBalance(account); break;  
        case INTEREST_RATE: processInterestRate(account); break;  
        case CREDIT_INTEREST: processCreditInterest(account); break;  
        case EXIT: break;  
        default: showUnknownCommand(); break;  
    }  
}
```

Executar uma operação

```
private static void executeOption(SafeBankAccount account,
                                   int option, Scanner in) {
    switch (option) {
        case DEPOSIT: processDeposit(account, in); break;
        ...
        default: showUnknownCommand(); break;
    }
}
```

```
private static void processDeposit(SafeBankAccount b, Scanner in) {
    int amount;
    System.out.println("Montante em centimos(valor nao negativo):");
    amount = in.nextInt();
    in.nextLine();
    if (amount >= 0){
        b.deposit(amount);
        System.out.println("Deposito efectuado com sucesso");
    } else
        System.out.println("Montante deve ser um valor nao negativo");
}
```

Para os restantes métodos auxiliares de *executeOption*,
será algo semelhante a esta implementação.

Executar uma operação

```
private static void processWithdraw(SafeBankAccount account,
                                    Scanner in){
    int amount = input.nextInt();
    input.nextLine();
    if (amount > 0) {
        account.withdraw(amount);
        System.out.println("Quantia " + amount + " levantada.");
    } else
        System.out.println("So pode levantar valores positivos!");
}

private static void processBalance(SafeBankAccount account) {
    System.out.println("Saldo actual = " + account.getBalance());
}

private static void processInterestRate(SafeBankAccount account) {
    System.out.println("Taxa juro anual = "+account.computeInterest());
}

private static void processCreditInterest(SafeBankAccount account){
    account.applyInterest();
    System.out.println("Juro creditado, saldo actual = "
                      + account.getBalance());
}
```

Criar uma conta bancária

Cria uma conta bancária, mostra as opções disponíveis, lê a opção escolhida pelo utilizador, executa a operação e mostra o saldo da conta após a operação. Finalmente, necessitamos de um novo método auxiliar para mostrar o saldo da conta: *showBalance()*.

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        SafeBankAccount account = createAccount(in, "conta");
        printMenu();
        int option = readOption(in);
        executeOption(account, option, in);
        showBalance(account);
        in.close();
    }
    private static void showBalance(SafeBankAccount account,
                                    Scanner in) {
        ...
    }
}
```

Mostrar o saldo final

```
public static void main(String[] args) {  
    Scanner in = new Scanner(System.in);  
    SafeBankAccount account = createAccount(in, "conta");  
    printMenu();  
    int option = readOption(in);  
    executeOption(account, option, in);  
    showBalance(account);  
    in.close();  
}  
  
private static void showBalance(SafeBankAccount account){  
    System.out.println("Saldo final: " + account.getBalance());  
}
```

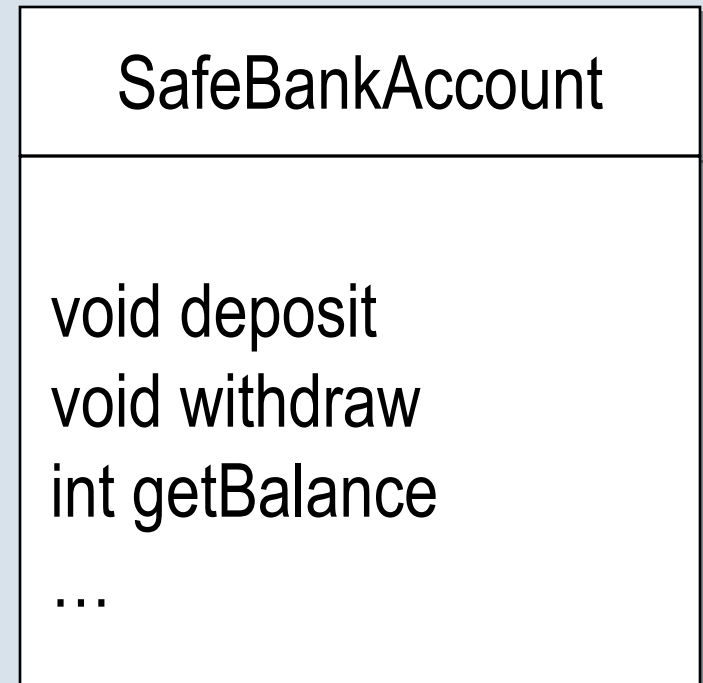
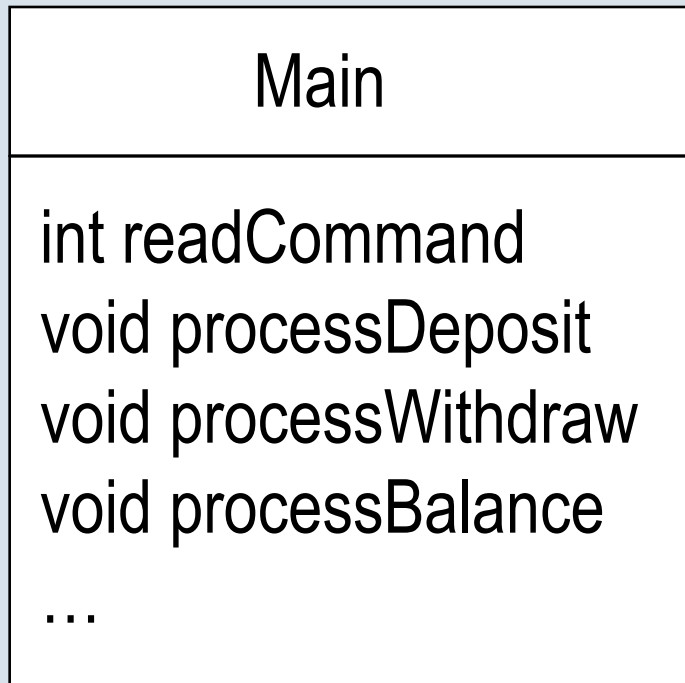
Mostrar o saldo final

```
public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    SafeBankAccount account = createAccount(in, "conta");
    printMenu();
    int option = readOption(in);
    executeOption(account, option, in);
    showBalance(account);
    in.close();
}

private static SafeBankAccount createAccount(...) {...}
private static void printMenu() {...}
private static int readOption(...) {...}
private static void executeOption(...) {...}
private static void processDeposit(...) {...}
private static void processWithdraw(...) {...}
private static void processBalance(...) {...}
private static void processInterestRate(...) {...}
private static void processCreditInterest(...) {...}
private static void showBalance(SafeBankAccount account) {...}
```

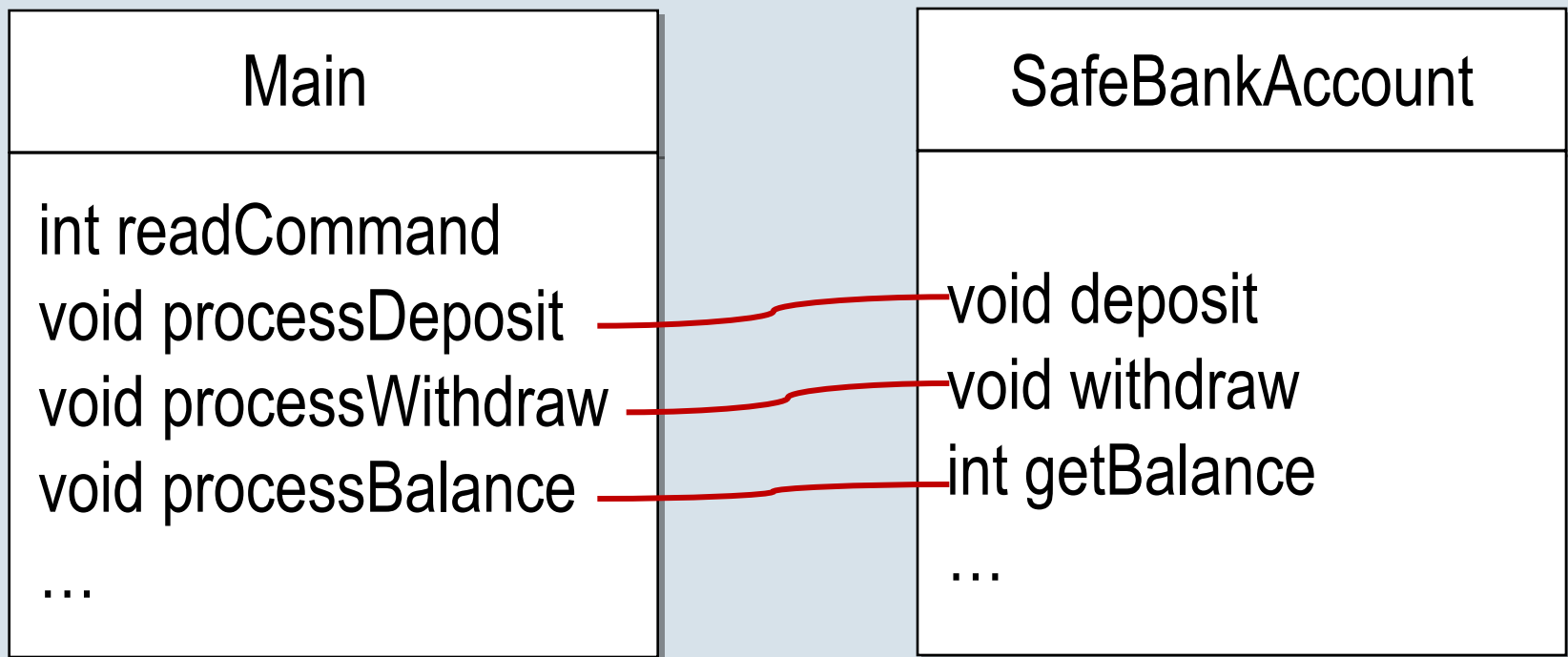
A classe `Main` é responsável pelo I/O

- Por ser responsável pelo I/O, a classe `Main` estabelece uma “ponte” entre o *utilizador final* (que usa o programa e interage com ele por meio do teclado, consola, etc.) e as restantes classes do programa, ditas específicas do domínio (e.g., `Semaphore`, `SafeBankAccount`, `Rect`, etc.)



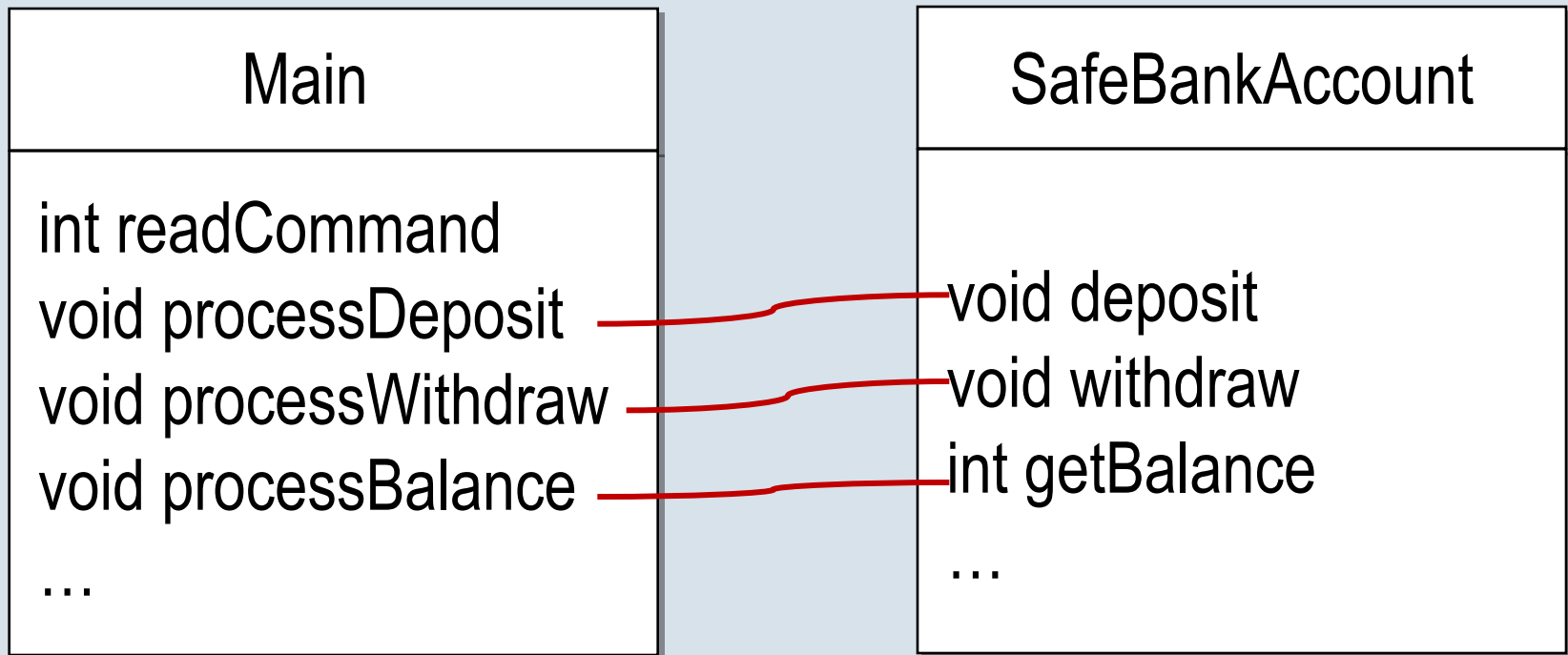
Métodos da classe `Main` usam métodos das classes lógicas

- Porque estabelece a ponte entre `SafeBankAccount` e o utilizador final, é de esperar que haja alguma correspondência entre alguns dos métodos de `Main` e métodos de `SafeBankAccount`.



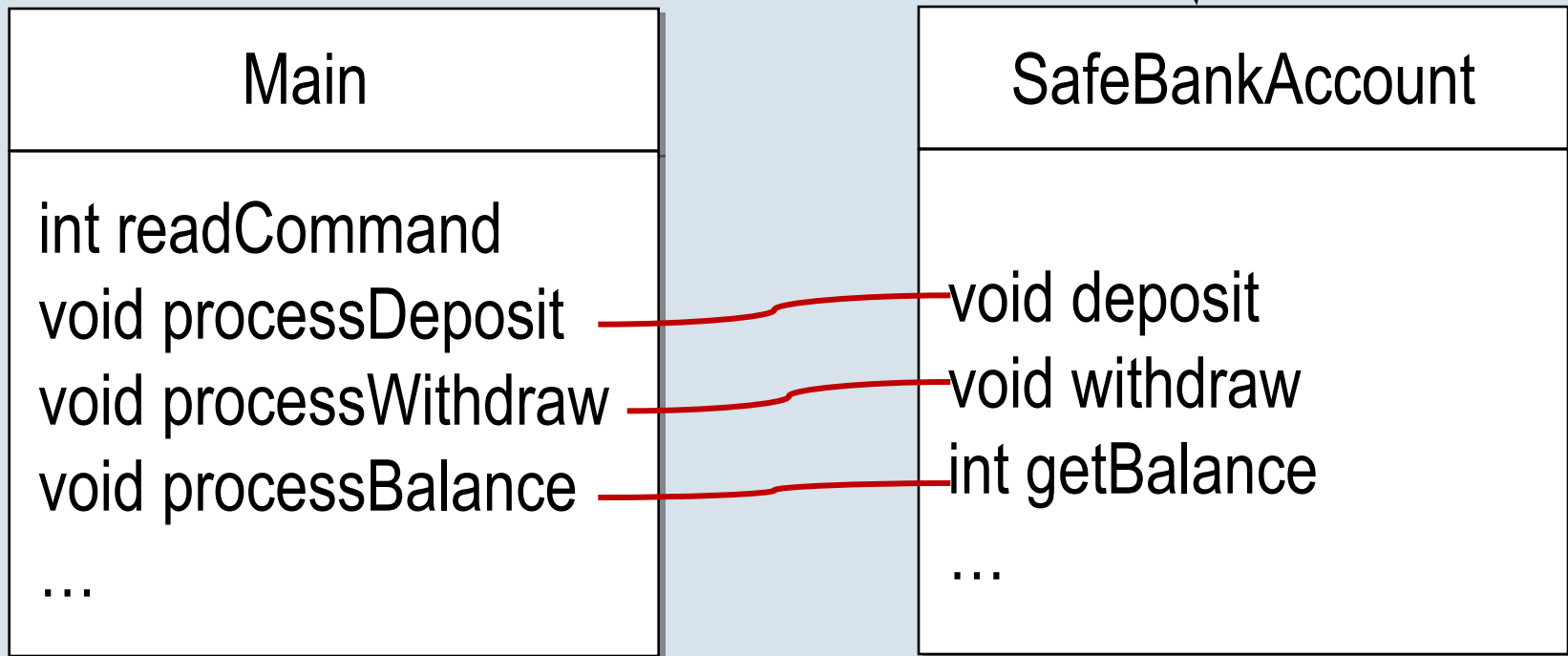
Métodos da classe `Main` usam métodos das classes lógicas

- Assim, encontramos em cada classe – `Main` e `SafeBankAccount` – um método relacionado com depósitos, um método relacionado com levantamentos, etc. Porém, o propósito dos métodos nas duas classes é bastante diferente.



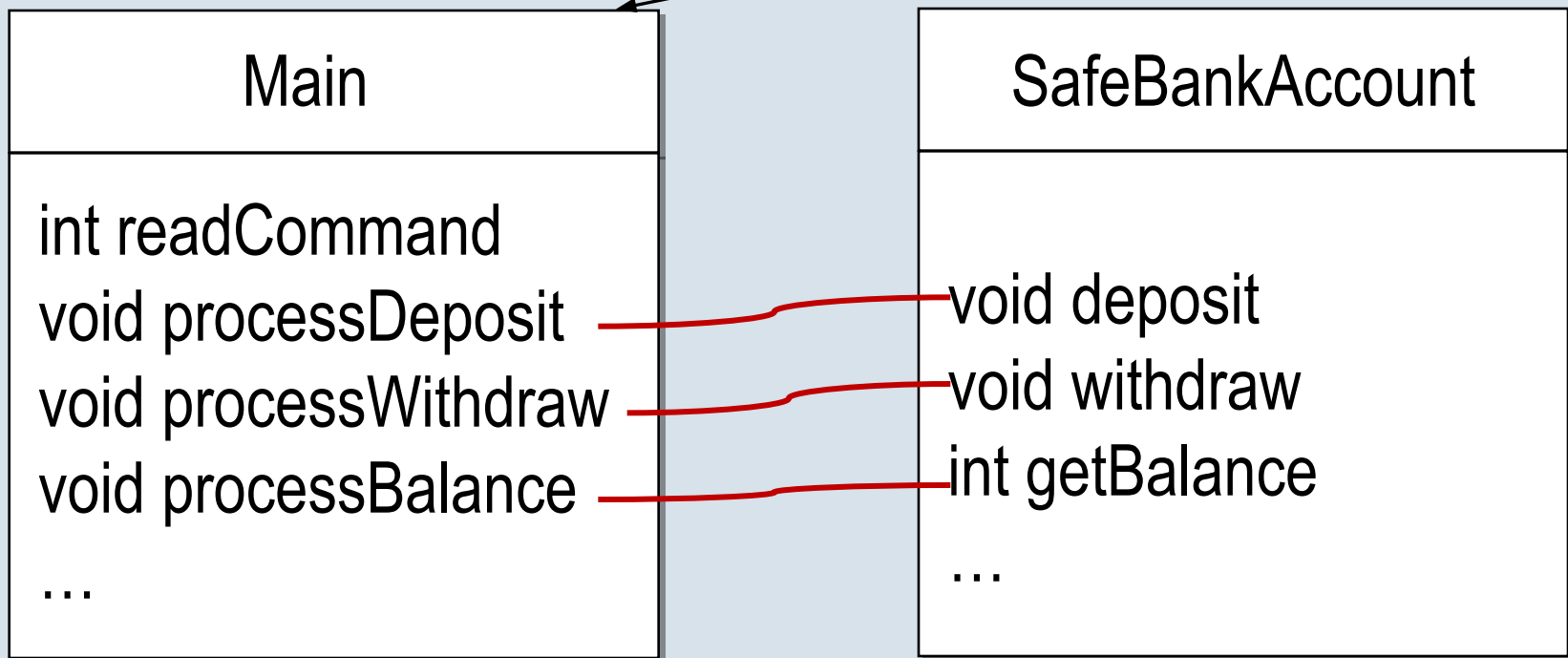
As classes lógicas são independentes da classe `Main`

A classe `SafeBankAccount` não sofre qualquer alteração: os seus métodos continuam dedicados a gerir o estado interno de uma conta bancária.



Métodos da classe `Main` realizam comandos

Os métodos – `private` e `static` – da classe `Main` destinam-se a realizar os comandos do interpretador: ler novo comando, determinar a tarefa a desempenhar, ler argumentos adicionais (e.g., saldo inicial, montante a depositar) e mostrar resultados na consola (e.g., saldo presente na conta, confirmação de que a operação teve sucesso, etc).



Nova interação com o utilizador (interpretador de comandos)

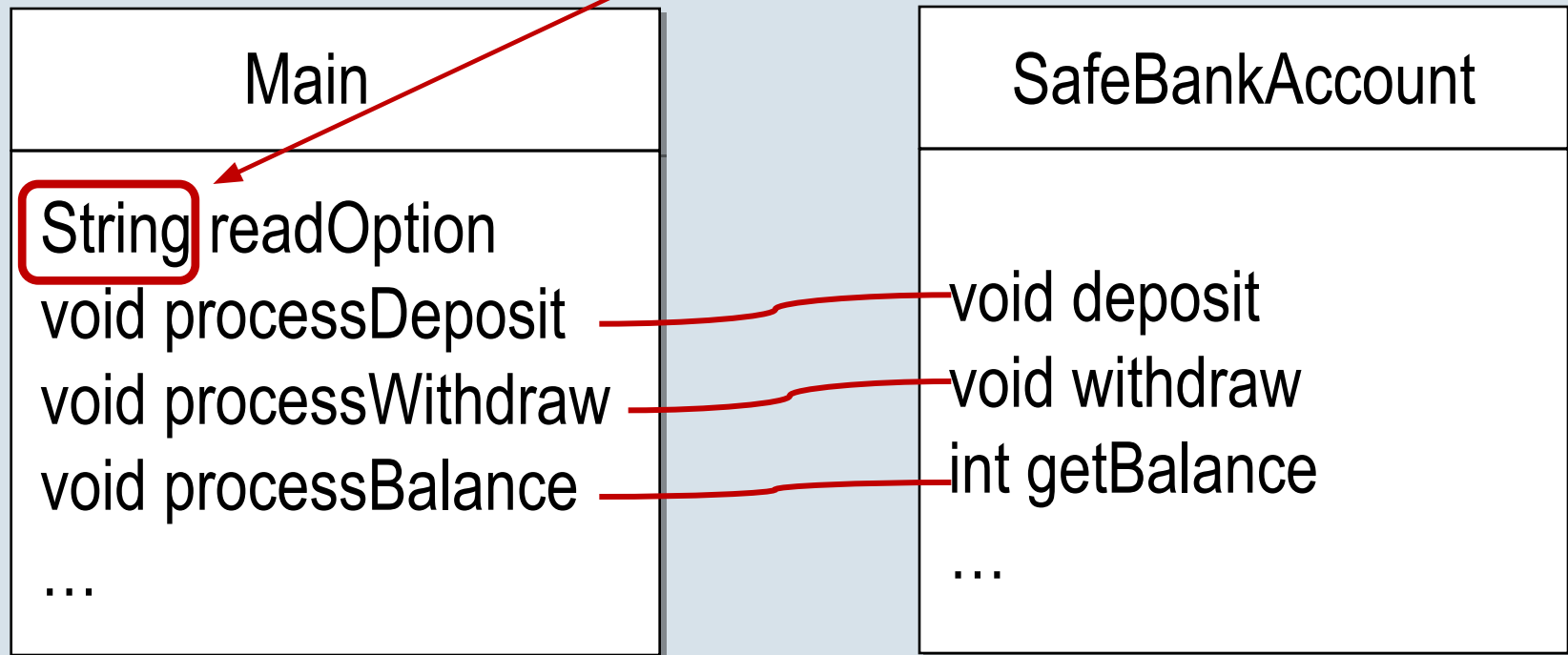
- Desenvolver um programa que:
 - Pede ao utilizador o saldo inicial de uma conta bancária.
 - Fica à espera dum comando do utilizador. Os comandos permitidos são:

```
L <montante_levantar> → levantar  
D <montante_depositar > → depositar  
CS → consultar saldo  
CJA → consultar juro anual  
AJA → aplicar juro anual  
S → sair
```

- Processa um comando. Para todos os casos deve escrever na consola se a acção foi bem sucedida ou não, e a informação pedida, caso exista.
- Escrever na consola o saldo da conta.

Alteração apenas na classe `Main`

Nesta versão, a única modificação é que o comando é representado por meio de uma `String` e não de um `int`. Esta é a prática mais habitual. Em relação à versão anterior de `Main`, as adaptações devem-se a esse aperfeiçoamento.



Main.java

```
private static final String DEPOSIT = "D";  
private static final String WITHDRAW = "L";  
private static final String BALANCE = "CS";  
private static final String INTEREST_RATE = "CJA";  
private static final String CREDIT_INTEREST = "AJA";  
private static final String EXIT = "S";
```

Nesta versão, as constantes são do tipo String.

```
private static void printMenu() {  
    System.out.println(WITHDRAW + "<Montante a levantar> - Levantar");  
    System.out.println(DEPOSIT + "<Montante a depositar> - Depositar");  
    System.out.println(BALANCE + " - Consultar saldo");  
    System.out.println(INTEREST_RATE + " - Consultar juro anual");  
    System.out.println(CREDIT_INTEREST + " - Aplicar juro");  
    System.out.println(EXIT + " - Sair");  
}
```

A leitura dos comandos será ligeiramente diferente. Os comandos de levantamento têm o argumento lido na mesma linha, agora. Além disso, o comando passou a ser uma String, em vez de um inteiro

Main.java

```
private static final String DEPOSIT = "D";
private static final String WITHDRAW = "L";
private static final String BALANCE = "CS";
private static final String INTEREST_RATE = "CJA";
private static final String CREDIT_INTEREST = "AJA";
private static final String EXIT = "S";

private static void printMenu() {
    System.out.println(WITHDRAW + "<Montante a levantar> - Levantar");
    System.out.println(DEPOSIT + "<Montante a depositar> - Depositar");
    System.out.println(BALANCE + " - Consultar saldo");
    System.out.println(INTEREST_RATE + " - Consultar juro anual");
    System.out.println(CREDIT_INTEREST + " - Aplicar juro");
    System.out.println(EXIT + " - Sair");
}

private static String readOption(Scanner input) {
    return input.next().toUpperCase();
}
```

Ao ler o comando, transformamos a String em maiúsculas, para facilitar a comparação de Strings. Estude o método `toUpperCase` da classe `String`.

Main.java

```
private static void executeOption(String option,
                                   SafeBankAccount account,
                                   Scanner input){

    switch (option){
        case DEPOSIT:
            processDeposit(account, input); break;
        case WITHDRAW:
            processWithdraw(account, input); break;
        case BALANCE:
            processBalance(account, input); break;
        case INTEREST_RATE:
            processInterestRate(account, input); break;
        case CREDIT_INTEREST:
            processCreditInterest(account, input); break;
        case EXIT:
            /* nao faz nada */ break;
        default: showUnknownCommand; break;
    }
}
```

Nesta versão, em todas as operações, temos de passar o Scanner, porque a leitura do comando não “deita fora” o resto da linha. Isto acontece porque alguns comandos têm argumentos, outros não.

Main.java

```
private static void processDeposit(SafeBankAccount account,
                                   Scanner in){
    int amount = in.nextInt();
    in.nextLine();
    if (amount > 0) {
        account.deposit(amount);
        System.out.println("Quantia " + amount + " depositada");
    } else
        System.out.println("Nao pode depositar valores negativos!");
}

private static void processWithdraw(SafeBankAccount account,
                                   Scanner in){
    int amount = input.nextInt();
    input.nextLine();
    if (amount > 0) {
        account.withdraw(amount);
        System.out.println("Quantia " + amount + " levantada.");
    } else
        System.out.println("So pode levantar valores positivos!");
}
```

Main.java

```
private static void processBalance(SafeBankAccount account,
                                   Scanner in) {
    in.nextLine();
    System.out.println("Saldo actual = " + account.getBalance());
}

private static void processInterestRate(SafeBankAccount account,
                                         Scanner in) {
    in.nextLine();
    System.out.println("Taxa juro anual = "+account.computeInterest());
}

private static void processCreditInterest(SafeBankAccount account,
                                           Scanner in){
    in.nextLine();
    account.applyInterest();
    System.out.println("Juro creditado, saldo actual = "
                      + account.getBalance());
}
```

Porque tudo na `Main` deve ser privado?

- Só devemos dar visibilidade pública aos membros de uma classe que se destinam ao seu uso externo
 - Essas partes públicas constituem a *interface* da classe
 - Usualmente, são operações – é por isso que as variáveis de instância costumam ser privadas
- Porém, os membros da classe `Main` nunca são chamados de outra classe. É função da `Main` chamar as outras classes, nunca o contrário.
Esta situação faz com que não faça sentido dar visibilidade pública aos membros de `Main`.
- A exceção é o método `main()`, porque é obrigatório que seja público, pelas regras do Java.

Princípio arquitectural

- Em qualquer aplicação que se programe é muito importante separar claramente
 - As partes responsáveis pela interacção com o utilizador
 - As partes responsáveis pela funcionalidade dos vários objectos (definidas em várias classes)
- Assim, **o uso de operações de output e input é apenas permitido no interior da classe Main ou nas classes responsáveis pela interacção.**
- Esta regra é mesmo muito importante.