Programa Principal
Leitura e Escrita de dados

Mestrado Integrado em Engenharia Informática FCT UNL
http://ctp.di.fct.unl.pt/miei/ip/

Corpo Docente 2020/2021
António Ravara, Artur Miguel Dias, Bernardo Toninho,
Ema Vieira, Inês Fernandes, Margarida Mamede,
Miguel Monteiro, Rui Nóbrega

Nova interacção com o utilizador (menu de opções)

- Desenvolver um programa que:
  - Pede ao utilizador o saldo inicial de uma conta bancária.
  - Apresenta em forma de menu 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, apos 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, apos 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, "cliente");
        // Mostra o menu de opcoes
        // Le uma opcao
        // Executa uma opcao
        // Mostra o saldo da conta, apos 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, "cliente");
    ...
    in.close();
}

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

Re-utilizar o `getIntValue()`

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

private static SafeBankAccount createAccount(Scanner in, String name) {
    SafeBankAccount account;
    int initial = getIntValue(in, "Saldo da conta de "+ name + ": ");
    if (initial > 0) {
        account = new SafeBankAccount(initial);

    private static int getIntValue(Scanner in, String msg){
        System.out.print(msg);
        int value = in.nextInt();
        in.nextLine();
        return value;
    }
}
```

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, "cliente");
        printMenu();
        // Le uma opcao
        // Executa uma opcao
        // Mostra o saldo da conta, apos 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 = 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, "cliente");
    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 + " - 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 do método auxiliar: ***getIntValue*()**.

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

Ler a opção seleccionada pelo utilizador

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

private static int getIntValue(Scanner in, String msg){
    System.out.print(msg);
    int value = in.nextInt();
    in.nextLine();
    return value;
}
```

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, "cliente");
        printMenu();
        int option = getIntValue(in,"Opcao:");
        executeOption(in, account, option);
        // Mostra o saldo da conta, apos a execucao da opcao
        in.close();
    }
    private static void executeOption(Scanner in,
                                      SafeBankAccount account,int option) {
        ...
    }
}
```

Executar uma operação (sem variável)

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, "cliente");
        printMenu();
        executeOption(in, account, getIntValue(in,"Opcao:"));
        // Mostra o saldo da conta, apos a execucao da opcao
        in.close();
    }
    private static void executeOption(Scanner in,
                                      SafeBankAccount account,int option) {
        ...
    }
}
```

Executar uma operação (sem variável)

Como escolher uma das 6 opções?

```
import java.util.Scanner;
public class Main {
    private static final int DEPOSIT = 1;
    private static final int WITHDRAW = 2;
    private static final int BALANCE = 3;
    private static final int INTEREST = 4;
    private static final int CREDIT_INTEREST = 5;
    private static final int EXIT = 6;
    ...
    private static void executeOption(Scanner in,
                        SafeBankAccount account,int option) {
        Usamos if encadeado???
            if(option == DEPOSIT) faz_algo
            else testa_as_outras_hipoteses
    }
}
```

Executar uma operação

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

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

Instrução `switch`

A instrução de composição alternativa múltipla

```
switch (expression) {
    case literal-1:
       statements-1;
       break;
    case literal-2:
       statements-2;
       break;
       // (more cases) ...
    case literal-N:
       statements-N;
       break;
    default: // optional default case
       statements-(N+1);
       break;
} // end of switch statement
```

expression tem de ser do tipo **int**, **char**, ou **String**

Instrução `switch`

A instrução de composição alternativa generalizada

```
switch (expression) {
    case literal-1:
        statements-1;
        break;
    case literal-2:
        statements-2;
        break;
        // (more cases) ...
    case literal-N:
        statements-N;
        break;
    default: // optional default case
        statements-(N+1);
        break;
} // end of switch statement
```

Cada *literal* tem um valor do mesmo tipo que *expression*.

O código executado é o que segue o *literal* que corresponde ao valor guardado em *expression*.

Instrução `switch`

A instrução de composição alternativa generalizada

```
switch (expression) {
    case literal-1:
        statements-1;
        break;
    case literal-2:
        statements-2;
        break;
        // (more cases) ...
    case literal-N:
        statements-N;
        break;
    default: // optional default case
        statements-(N+1);
        break;
} // end of switch statement
```

A utilização de **`break`** é opcional e põe termo à execução da instrução **`switch`**.
Se for omitido, o caso seguinte ao escolhido também será executado (e assim sucessivamente até haver um caso com **`break`**).

Instrução `switch`

A instrução de composição alternativa generalizada

```
switch (expression) {
    case literal-1:
        statements-1;
        break;
    case literal-2:
        statements-2;
        break;
        // (more cases) ...
    case literal-N:
        statements-N;
        break;
    default: // optional default case
        statements-(N+1);
        break;
} // end of switch statement
```

A utilização do **`default`** também é opcional. Este ramo é executado se nenhum dos casos explícitos corresponder ao valor de ***`expression`***.

Instrução `switch`

A instrução de composição alternativa generalizada

```
switch (expression) {
    case literal-1:
        statements-1;
        break;
    case literal-2:
        statements-2;
        break;
        // (more cases) ...
    case literal-N:
        statements-N;
        break;
    default: // optional default case
        statements-(N+1);
        break;
} // end of switch statement
```

O bloco de instruções de um caso pode ser vazio. Isto implica a execução do código do caso seguinte para ambos os casos.

Que expressão testar num **`switch`**?

- Constante (pouco útil)
- tipo **`int`**
  - Exemplo

```
public void goodExample1(int i) {
  ...
  switch (i) {
    ...
  } // end of switch statement
} // end of goodExample1
```

- tipo (de sequência de) caracteres

O tipo char

Tipo Char

- O tipo **`char`** permite representar caracteres (do alfabeto latino e de outros alfabetos), assim como outros símbolos.
- Tecnicamente, os valores de tipo char são inteiros de 16 bits sem sinal, com valores que variam de 0 até 65535.
- Um **`char`** representa uma letra, um dígito, um sinal de pontuação, uma tabulação (tab) um espaço, etc.
- Um literal do tipo char é escrito com o carácter entre plicas.
  Exemplos de literais: **'a' 'b' 'A' 'Z' ';' 'e' '#' '3' ' ' '.' '字'**
- Iremos usar apenas os caracteres da tabela ASCII, representados na memória por um valor inteiro entre 0 e 127 (i.e. com o byte mais significativo a valer zero).

Tabela ASCII

American Standard Code for Information Interchange

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Tipo `String`

- Em Java as sequências de caracteres são representadas como valores do tipo **`String`**.
- As Strings representam "bocados" de texto, que podem ser manipulados pelos programas através de operações especiais.
- Os valores de tipo **`String`** escrevem-se como sequências de caracteres arbitrários (menos as ″'s) entre **`" "`**.

  Por exemplo:

  - **`"As armas e os barões assinalados"`**
  - **`"%&Ghg1j9892bnsabdGHJbsu?*"`**
  - **`"this is a recording, please hang up!"`**
  - **`"class"`**
  - **`"早上好"`** **`"Καλημέρα"`** `(não vamos usar isto)`

Operações básicas sobre o tipo `String`

- Uma operação muito usada com Strings é a concatenação (justaposição) **+**.
  - `"hello"` **+** `" "` **+** `"world"` produz `"hello world"`
  - `"X"` **+** `"Y"` produz `"XY"`

- É possível concatenar também inteiros (**`int`**) com strings; nesse caso o inteiro é convertido na sua representação como String (por exemplo, o inteiro `12` é convertido na String `"12"`).
  - `"Altura="` **+** `height` produz `"Altura=10"`
  - `"Horas "` **+** `mi` **+** `" minutos"` produz `"Horas 0 minutos"`
  - `"Ping"` **+** `(2-1)` produz `"Ping1"`

- Para converter uma String numérica num inteiro, não existe mecanismo automático e teríamos de fazer como neste exemplo
  - int i = Integer.parseInt("123");

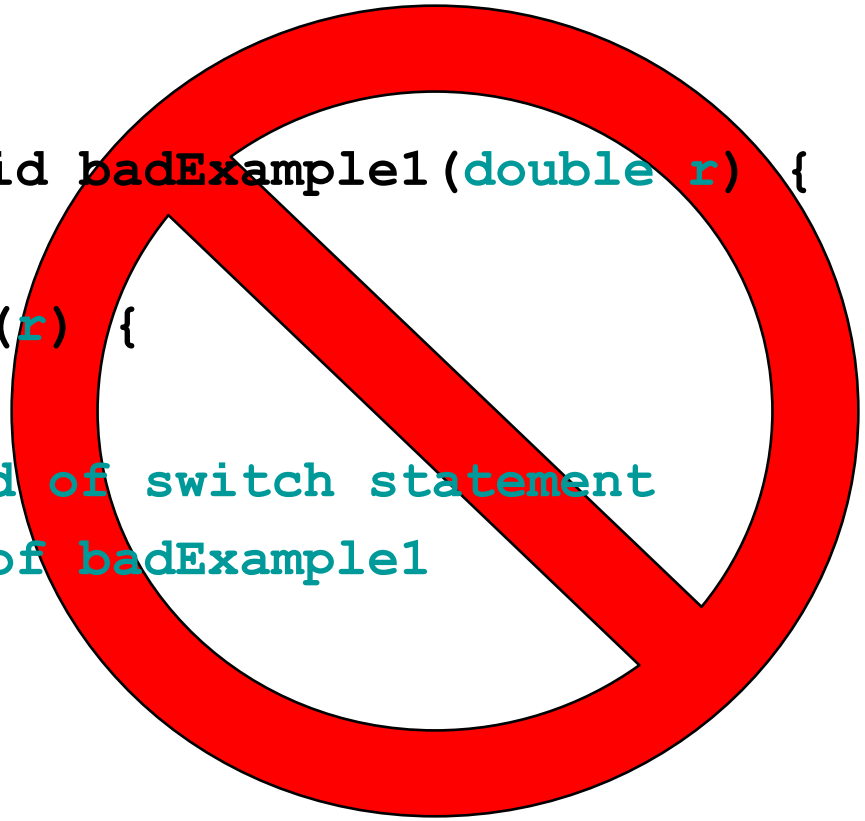
Expressão no switch: tem de ser `int`, `char`, ou `String`

```
// pre: month != null
public int getMonthNumber(String month) {
  int monthNumber = 0;
  switch (month) { // Only lowercase month names are recognized!
    case "january":
      monthNumber = 1;
      break;
    case "february":
      monthNumber = 2;
      break;
    //... Several months omitted here...
    case "november":
      monthNumber = 11;
      break;
    case "december":
      monthNumber = 12;
      break;
  } // end of switch statement
  return monthNumber;
} // end of getMonthNumber operation
```

Expressão no switch: tem de ser `int`, `char`, ou `String`

- A expressão `expression` em `switch (expression)` tem de ser dos tipos `int`, `char`, ou `String`, e constante
  - Exemplos de erros típicos

```
public void badExample1(double r) {
   ...
   switch (r) {
      ...
   } // end of switch statement
} // end of badExample1
```



O compilador dá erro, porque o tipo da expressão não é nem **`int`** nem **`char`**, nem **`String`**!

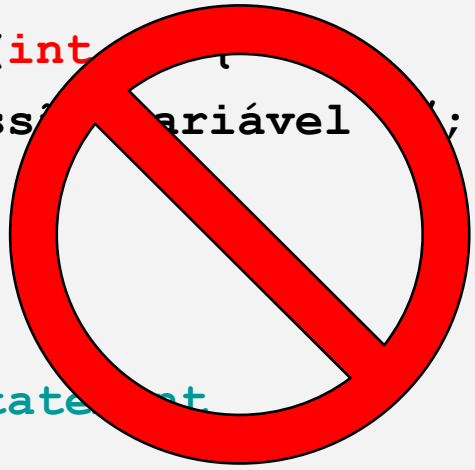
Os literais de cada `case` são constantes

- Exemplo correcto (o literal `3` é constante):

```
public void goodExample(int i) {
  ...
  switch (i) {
   case 3:...
    ...
  } // end of switch statement
} // end of goodExample
```

- Exemplo errado (`x` não é constante!):

```
public void badExample(int
  int x = /* uma expressi   ariável
  switch (i) {
     case x: ...
     ...
  } // end of switch state   nt
} // end of badExample
```



Os literais de cada **`case`** são constantes

- Exemplo correcto (o literal `'e'` é constante):

```
public void goodExample2(char c) {
   ...
   switch (c) {
     case 'e':...

   } // end of switch statement
} // end of goodExample2
```

Executar uma operação

```
public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    SafeBankAccount account = createAccount(in, "cliente");
    printMenu();
    executeOption(in, account, getIntVal
    // Mostra o saldo da conta, apos 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(Scanner in,
                                  SafeBankAccount account,int option) {
    switch (option) {
        case DEPOSIT: processDeposit(in, account); break;
        case WITHDRAW: processWithdraw(in, account); break;
        case BALANCE: processBalance(account); break;
        case INTEREST: processInterest(account); break;
        case CREDIT_INTEREST: processCreditInterest(account); break;
        case EXIT: break;
        default: showUnknownCommand(); break;
    }
}
```

Executar uma operação

```
private static void executeOption(Scanner in,
                                  SafeBankAccount account,int option) {
    switch (option) {
        case DEPOSIT: processDeposit(in, account); break;
        ...
        default: showUnknownCommand(); break;
    }
}
private static void processDeposit(Scanner in, SafeBankAccount b){
    int amount = getIntValue(in,
                        "Montante em centimos(valor nao negativo):");
    if (amount > 0){
        b.deposit(amount);
        System.out.println("Deposito efectuado com sucesso");
    } else
        System.out.println("Montante deve ser um valor positivo");
}
```

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

Executar uma operação

```
private static void processWithdraw(Scanner in,
                                    SafeBankAccount account){
    int amount = getIntValue(in,"Montante em centimos(valor positivo):");
    if (amount > 0 && amount <= account.getBalance()) {
        account.withdraw(amount);
        System.out.println("Quantia " + amount + " levantada.");
    } else
        System.out.println("Levantamento invalido!");
}
private static void processBalance(SafeBankAccount account) {
    System.out.println("Saldo actual = " + account.getBalance());
}
private static void processInterest(SafeBankAccount account) {
    System.out.println("Juro anual = "+account.computeInterest());
}
private static void processCreditInterest(SafeBankAccount account){
    account.applyInterest();
    System.out.println("Juro creditado, saldo actual = "
                       + account.getBalance());
}
```

Apresentar Saldo Final

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, "cliente");
        printMenu();
        executeOption(in, account, getIntValue(in,"Opcao:" );
        processBalance(account);
        in.close();
    }
    private static void processBalance(SafeBankAccount account) {

        ...
    }
}
```

Mostrar o saldo final

```
public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    SafeBankAccount account = createAccount(in, "cliente");
    printMenu();
    executeOption(in, account, getIntValue(in,"Opcao:" ));
    processBalance(account);
    in.close();
}

private static void processBalance(SafeBankAccount account){
    System.out.println("Saldo atual: " + account.getBalance());
}
```

Mostrar o saldo final

```
public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    SafeBankAccount account = createAccount(in, "cliente");
    printMenu();
    int option = getIntValue(in,"Opcao:" );
    executeOption(in, account, option);
    processBalance(account);
    in.close();
}
private static SafeBankAccount createAccount(...){...}
private static void printMenu(){...}
private static int getIntValue(...){...}
private static void executeOption(...){...}
private static void processDeposit(...){...}
private static void processWithdraw(...){...}
private static void processInterest(...){...}
private static void processCreditInterest(...){...}

private static void processBalance(SafeBankAccount account){...}
```

Pré-condições

```
import java.util.Scanner;
public class Main {

    public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      SafeBankAccount account =
                              createAccount(in, "cliente");
      printMenu();
      executeOption(in, account,
                    getIntValue(in,"Opcao:" ));
      processBalance(account);
      in.close();
    }

}
```

Todos estes métodos têm pré-condições:

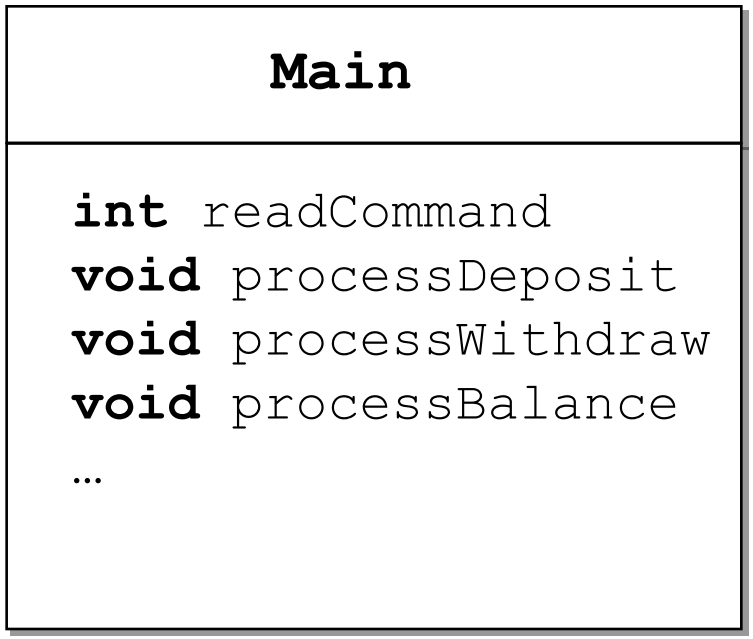
- Todos os parâmetros de Tipos de Classe (tipos não primitivos do Java), têm de ser não `null`
- No `executeOption()`, o 3º parâmetro tem de estar dentro dos valores válidos (1-6)

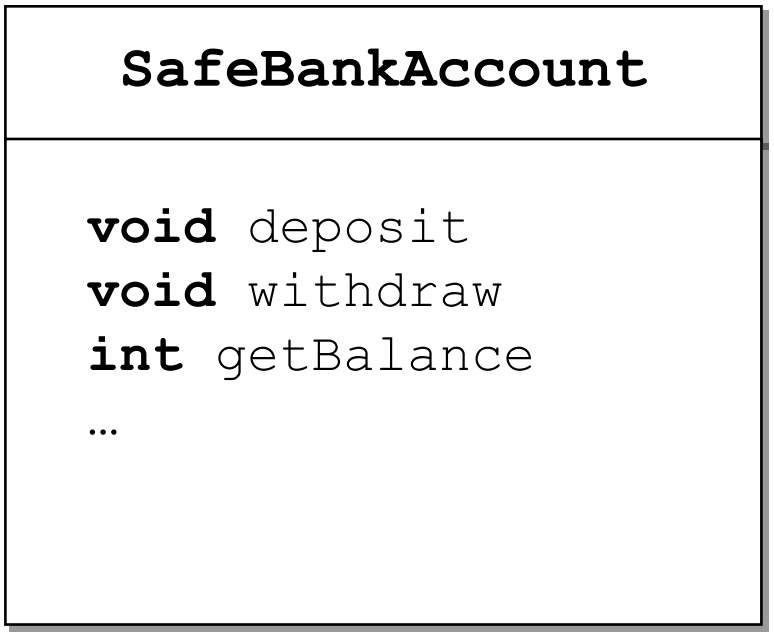
Pré-condições

```
...


//pre: in != null && name != null
private static SafeBankAccount createAccount(Scanner in, String name) {...}
// pre: in != null && msg != null
private static int getIntValue(Scanner in, String msg) {...}
// pre: in != null && account != null
private static void processDeposit(Scanner in, SafeBankAccount account) {...}
// pre: in != null && account != null
private static void processWithdraw(Scanner in, SafeBankAccount account){...}
// pre: account != null
private static void processInterest(SafeBankAccount account) {...}
// pre: account != null
private static void processCreditInterest(SafeBankAccount account) {...}
// pre: account != null
private static void processBalance(SafeBankAccount account) {...}
// pre: in != null && account != null && option >= 1 && option <= 6
private static void executeOption(Scanner in,
                                  SafeBankAccount account, int option) {...}


...
```

A classe `Main` é responsável pelo I/O

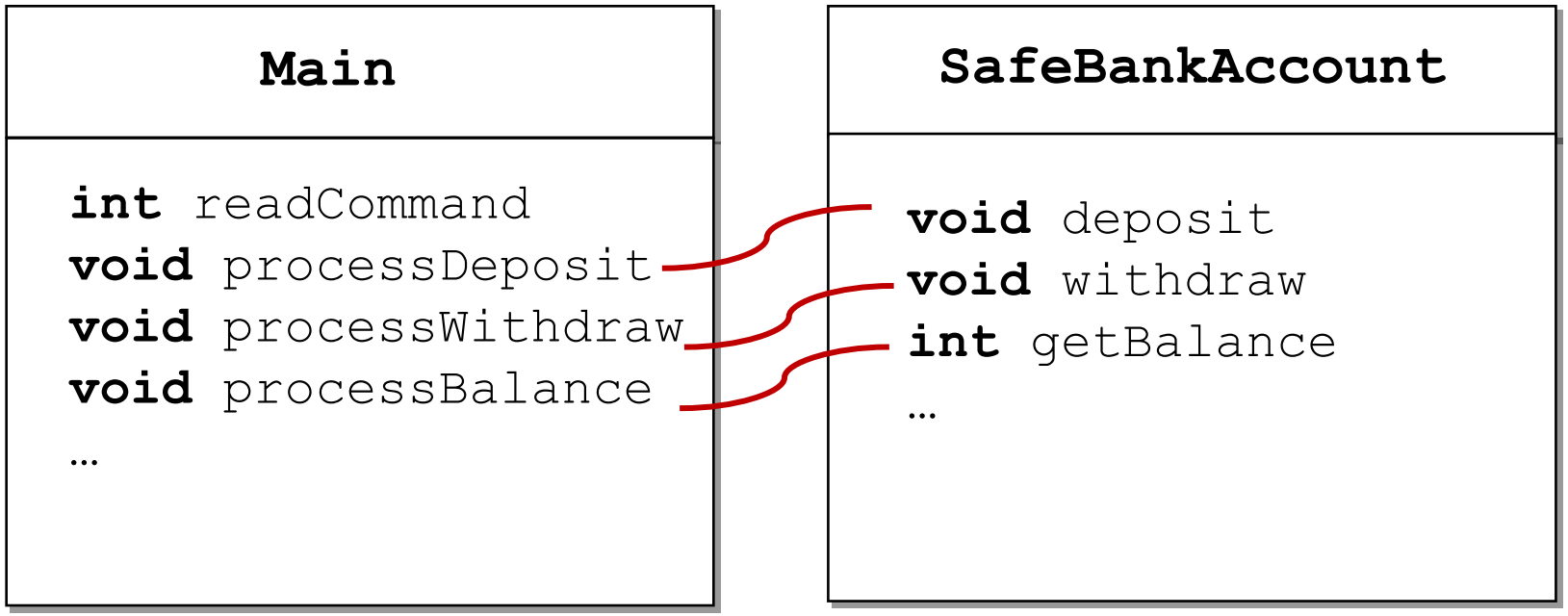
- Por ser responsável pelo I/O, a classe `Main` estabelece uma "ponte" entre o *utilizador final* (que usa o programa e interage com ele por meio do teclado, consola, etc.) e as restantes classes do programa, ditas específicas do domínio (e.g. `SafeBankAccount`, `Rect`, etc.)





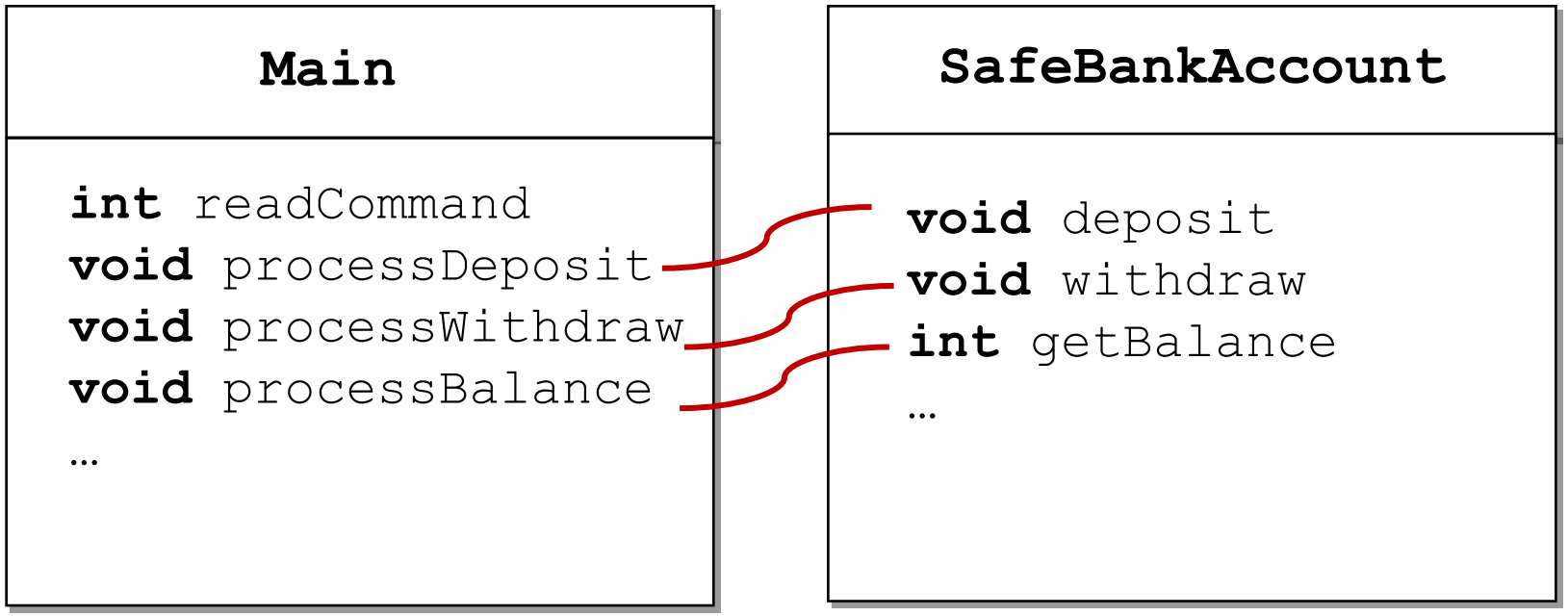
Métodos da classe `Main` usam métodos das classes lógicas

- 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`.



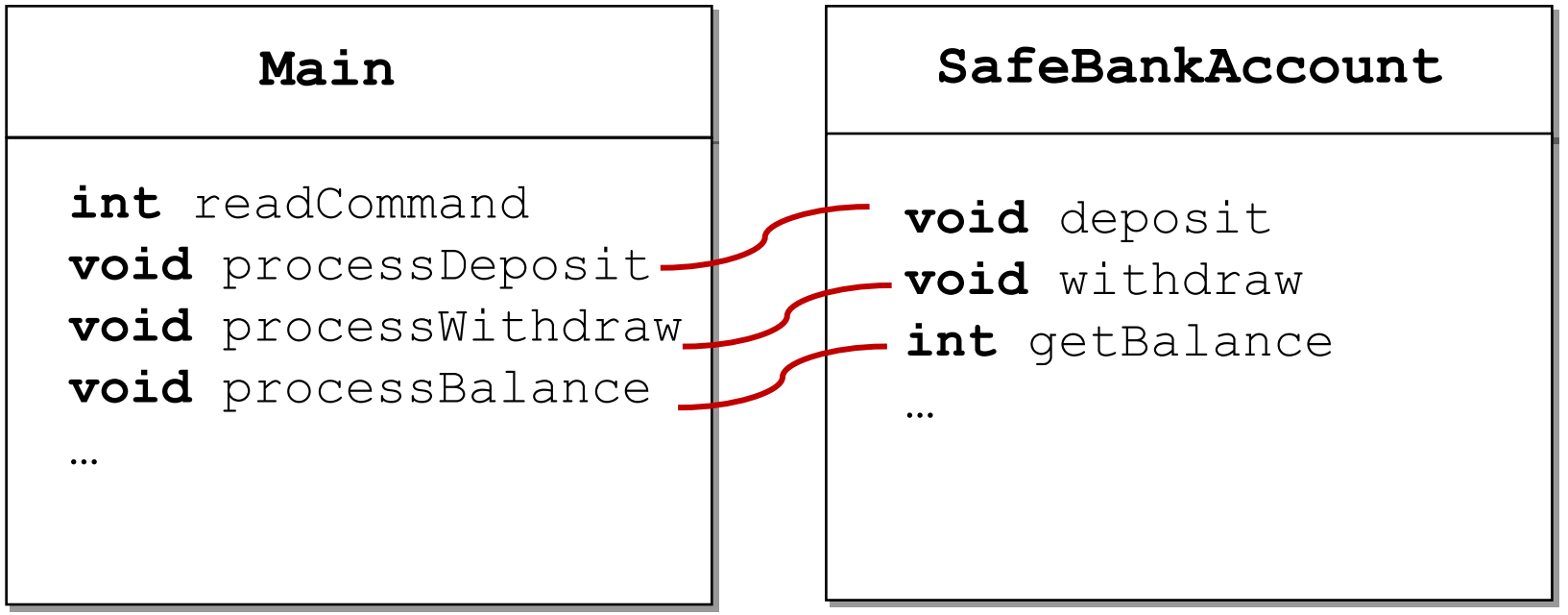
Métodos da classe `Main` usam métodos das classes lógicas

- 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.



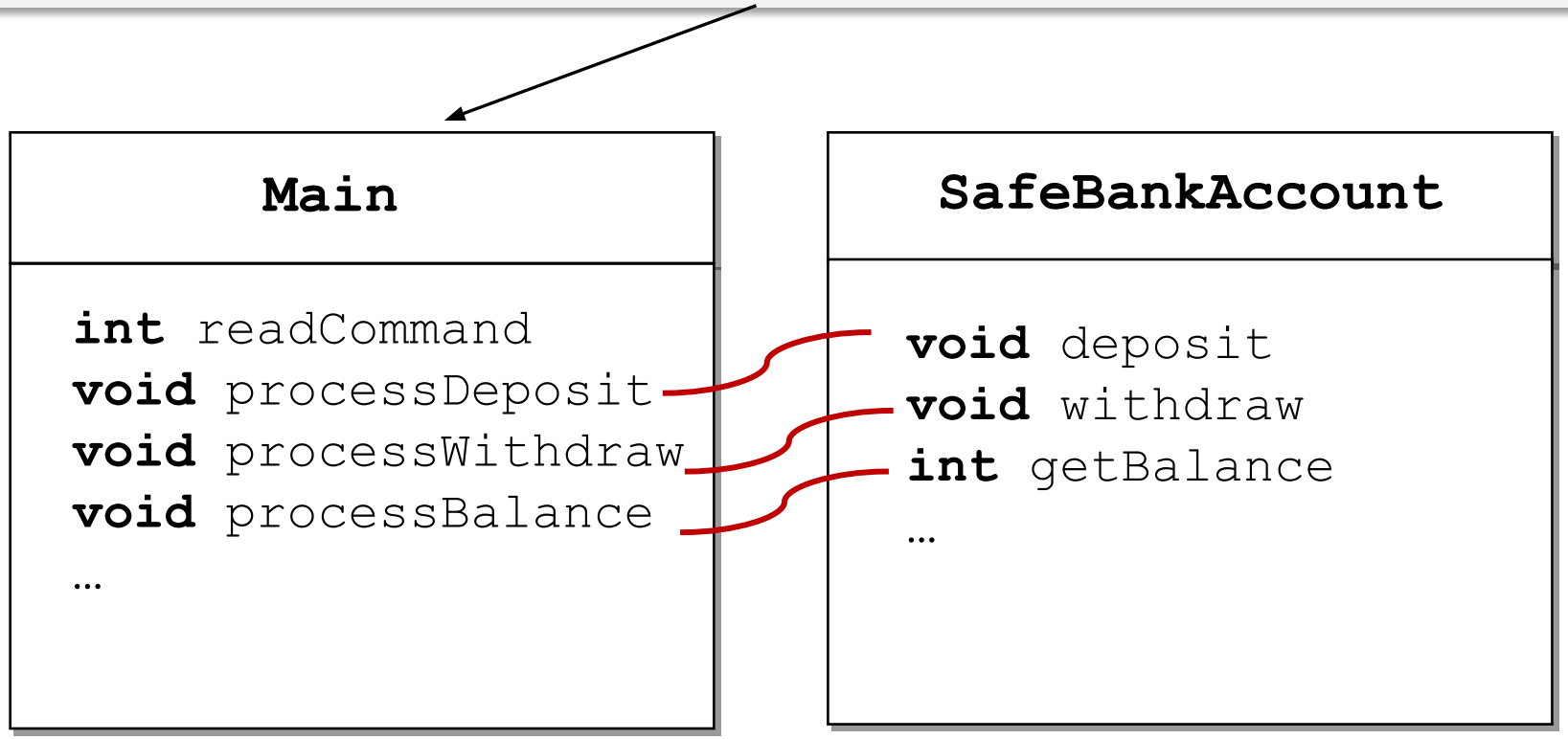
As classes lógicas são **independentes** da classe `Main`

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)

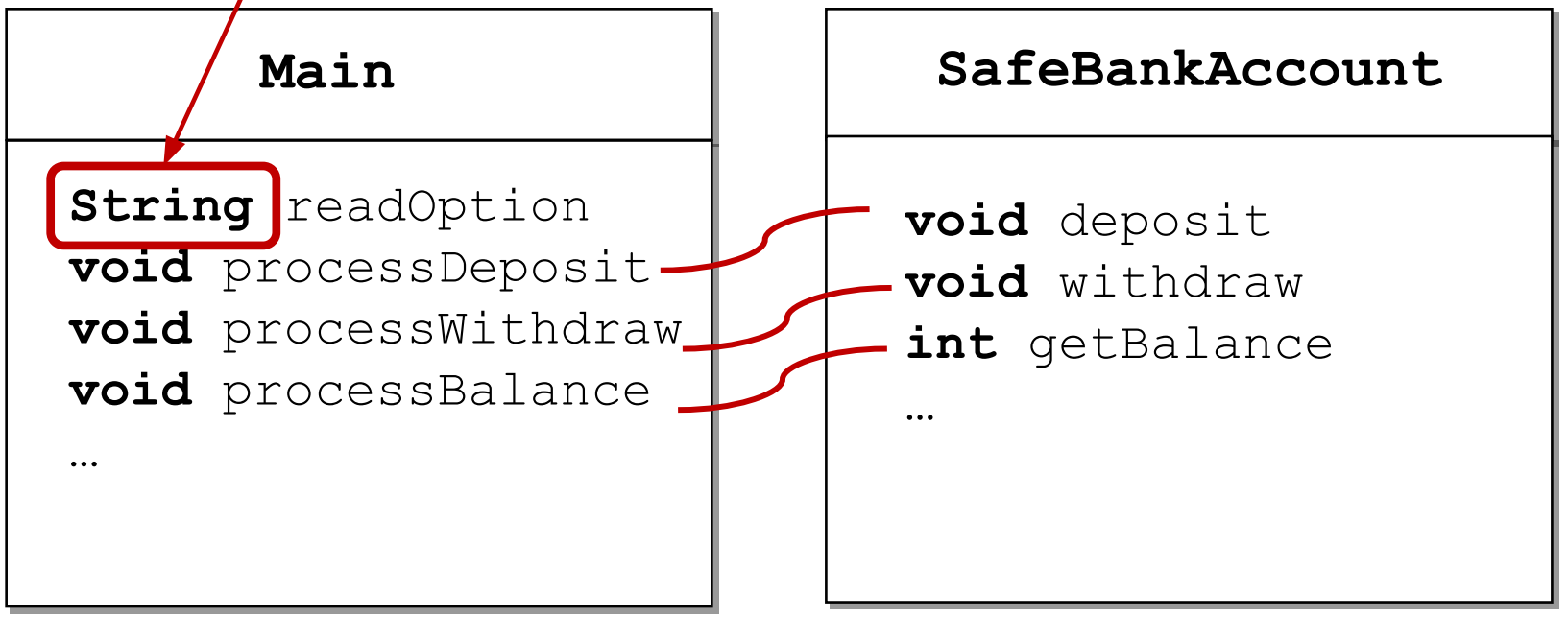
- Desenvolver um programa que:
  - Pede ao utilizador o saldo inicial de uma conta bancária.
  - Fica à espera dum comando do utilizador. Os comandos permitidos são:

```
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 = "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 + " - Consultar juro anual");
    System.out.println(CREDIT_INTEREST + " - Aplicar juro");
    System.out.println(EXIT + " - Sair");
}
```

Nesta versão, as constantes são do tipo String.

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 = "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 + " - 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`.

`getIntValue()` sem mensagem

```
private static int getIntValue(Scanner input) {
   int value = input.nextInt();
   input.nextLine();
   return value;
}
```

Main.java

```
private static void executeOption(Scanner input,
                                  SafeBankAccount account,
                                  String option){
    switch (option){
        case DEPOSIT:
            processDeposit(getIntValue(input), account);break;
        case WITHDRAW:
            processWithdraw(getIntValue(input), account);break;
        case BALANCE:
            in.nextLine();
            processBalance(account); break;
        case INTEREST:
            in.nextLine();
            processInterest(account); break;
        case CREDIT_INTEREST:
            in.nextLine();
            processInterest(account);
            processCreditInterest(account); break;
        case EXIT: in.nextLine(); break;
        default: in.nextLine();
                 showUnknownCommand();    }
}
```

Nesta versão, em todas as opções que não necessitam de input, temos de "libertar" o resto da linha.

Main.java

```
private static void processDeposit(int amount,
                                   SafeBankAccount account){
    if (amount > 0) {
        account.deposit(amount);
        System.out.println("Quantia " + amount + " depositada");
    } else
        System.out.println("Nao pode depositar valores nao positivos");
}

private static void processWithdraw(int amount,
                                    SafeBankAccount account){
    if (amount > 0 && amount <= account.getBalance()) {
        account.withdraw(amount);
        System.out.println("Quantia " + amount + " levantada.");
    } else
        System.out.println("Levantamento invalido");
}
```

Main.java

```
private static int getIntValue(Scanner input) {
    int value = input.nextInt();
    input.nextLine();
    return value;
}

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

private static void processInterest(SafeBankAccount account) {
    System.out.println("Juro anual = "+account.computeInterest());
}

private static void processCreditInterest(SafeBankAccount account){
    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 a partir de outra classe. É responsabilidade 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 excepção 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 input e output é apenas permitido no interior da classe `Main` ou nas classes responsáveis pela interacção.**
- Esta regra é **mesmo** muito importante.