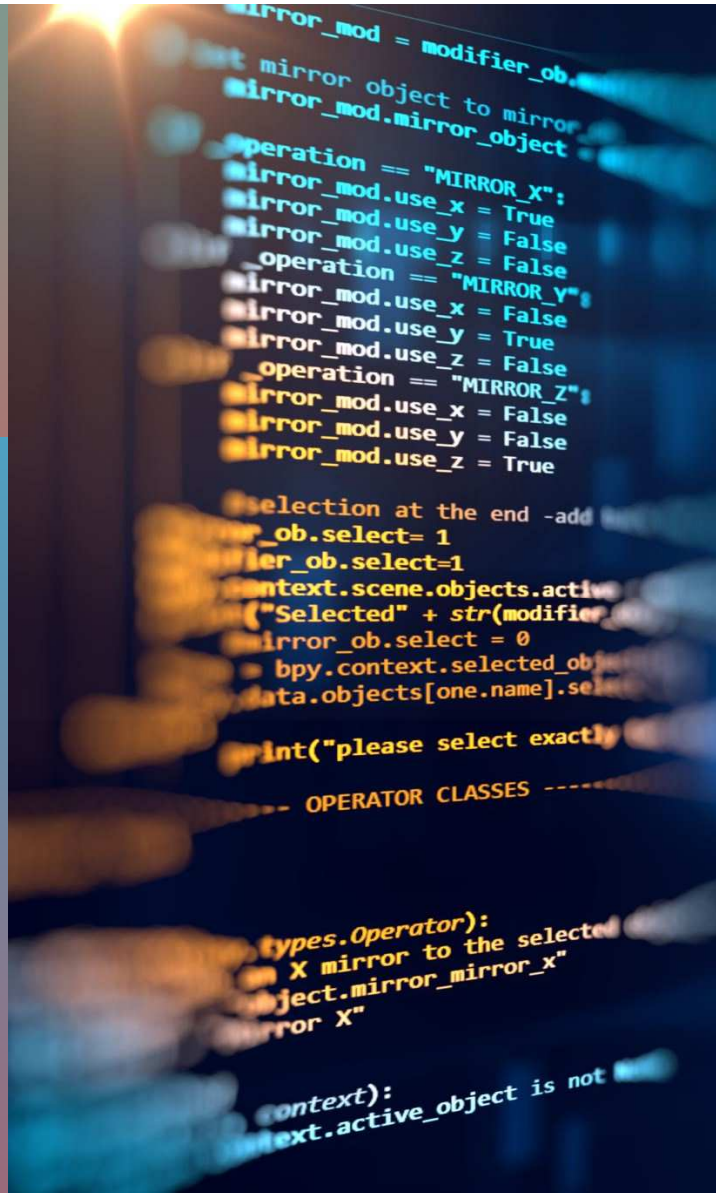
UFCD 0789

FUNDAMENTOS DE LINGUAGEM JAVA



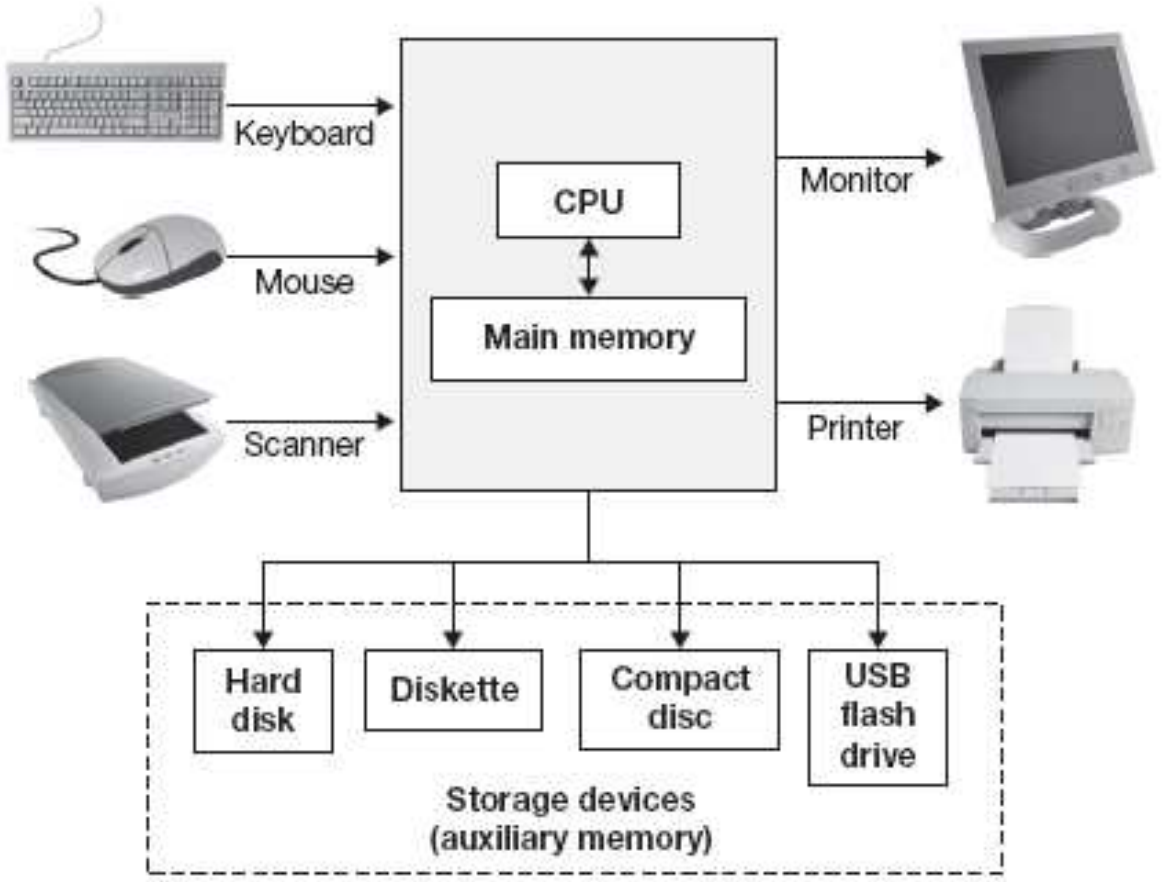
PRINCÍPIOS DA PROGRAMAÇÃO

PARTE I

PRINCÍPIOS DA PROGRAMAÇÃO

- Conceitos básicos de programação
- Estrutura lógica de uma aplicação
- Variáveis e tipos de dados
- Expressões e operações
- Regras de precedência, ordens de avaliação
- Estruturas de decisão
- Estruturas cíclicas
- Definir e invocar funções
- Modular código usando funções reutilizáveis

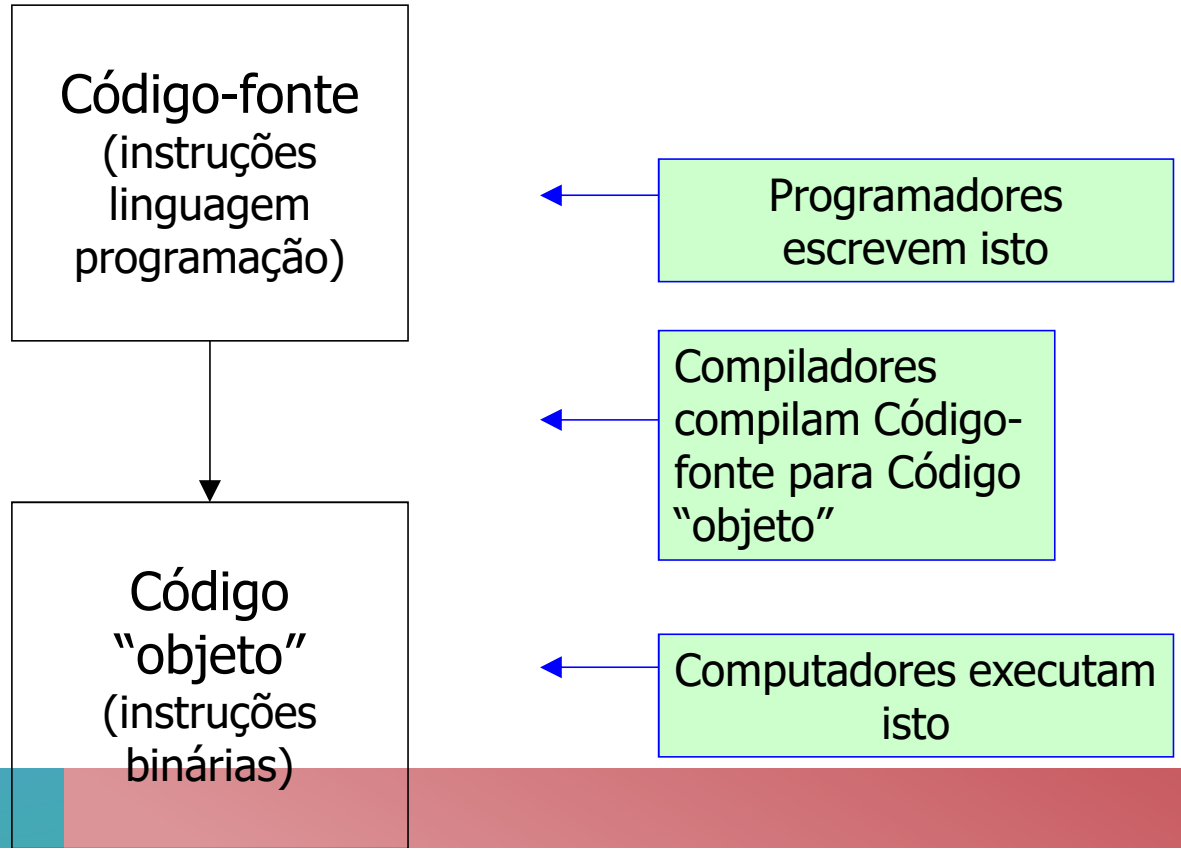
GLOSSÁRIO DE HARDWARE



LINGUAGEM DE PROGRAMAÇÃO

- É uma linguagem que utiliza palavra, gramática e sintaxe que um computador entende
- Inicialmente uma linguagem de programação pode ser difícil de entender
- Mas após ganhar alguma experiência com uma linguagem de programação, facilmente conseguirá passar de uma linguagem de programação para outra sem dificuldade
- No entanto eu recomendo que apenas passe para outra linguagem de programação quando já está confortável numa em específico

O PROCESSO DE COMPILAÇÃO PARA PROGRAMAS "NÃO JAVA"



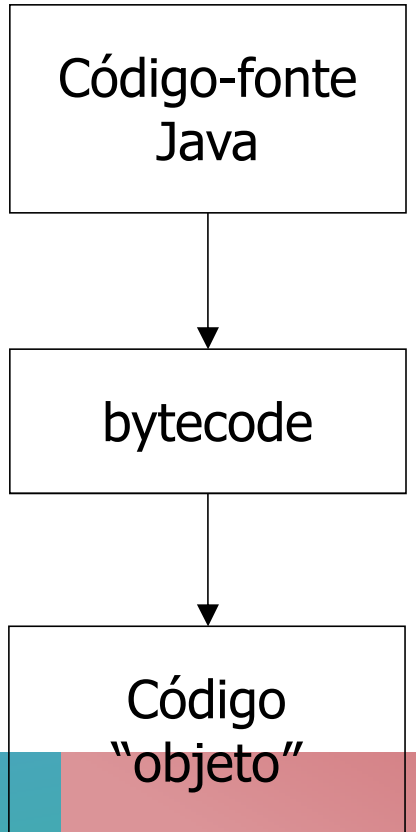
JAVA VIRTUAL MACHINE

- Como é que o bytecode pode ser executado em cada tipo de computador

- Quando um programa Java é executado, o bytecode é “traduzido” em código “objeto” pela Java Virtual Machine

- Também conhecida por JVM

PROCESSO DE COMPILAÇÃO PARA PROGRAMAS JAVA



Compiladores Java compilam código-fonte em bytecode

Quando um programa Java é executado, a JVM traduz o bytecode para código "objeto"

CLASSE JAVA OLÁ MUNDO

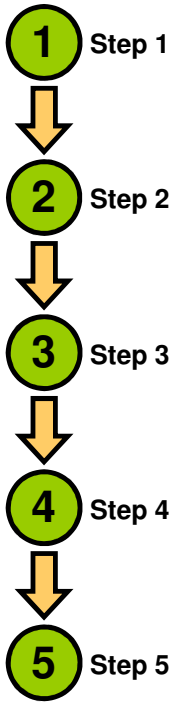
```
public class HelloWorld
{
  public static void main(String[] args)
  {
    System.out.println("Olá mundo ☺");
  }
} // end class HelloWorld
```

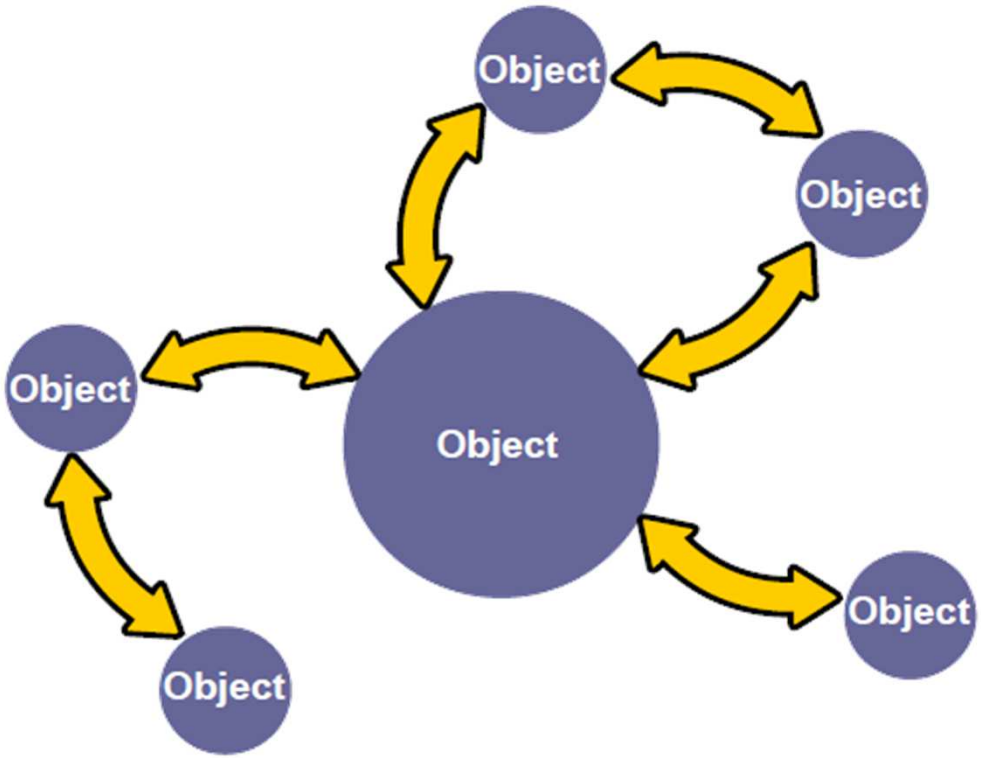
PRINCIPAIS CONCEITOS DO JAVA

- Orientado a objetos
- Distribuída
- Simples
- Multitarefa
- Segura
- Independente de plataforma

PROGRAMAÇÃO PROCEDIMENTAL

- Foca-se na sequência de processos



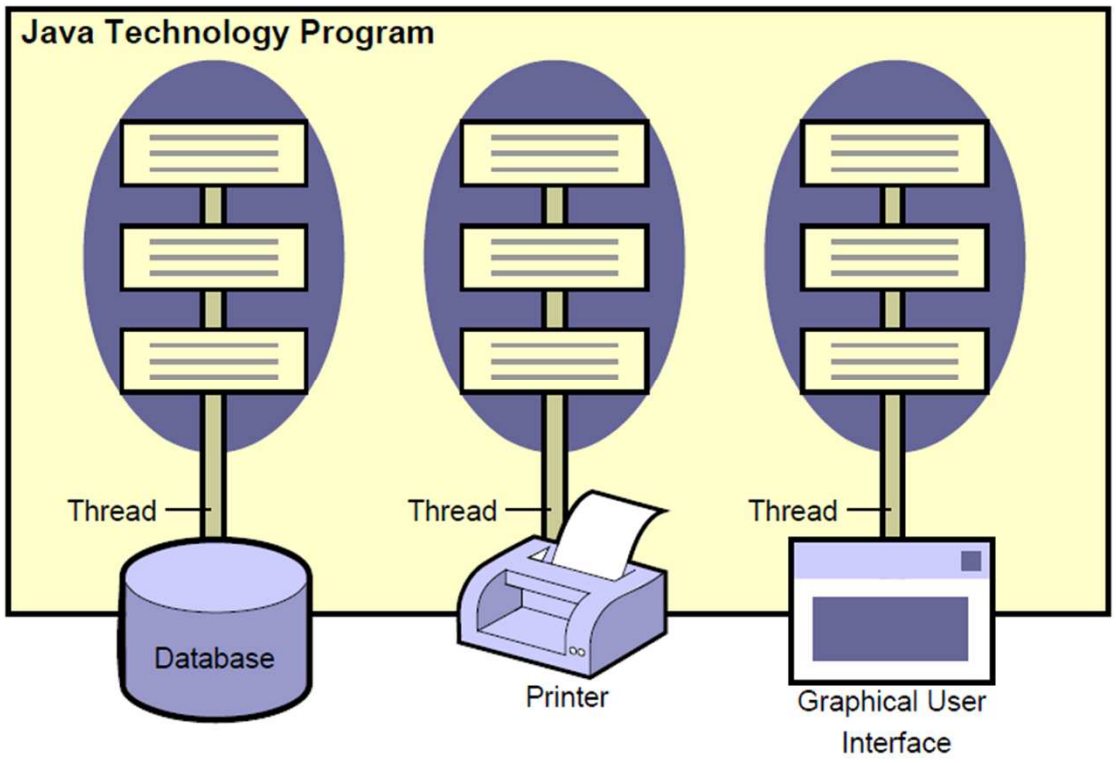


PROGRAMAÇÃO ORIENTADA A OBJETOS

DISTRIBUÍDA

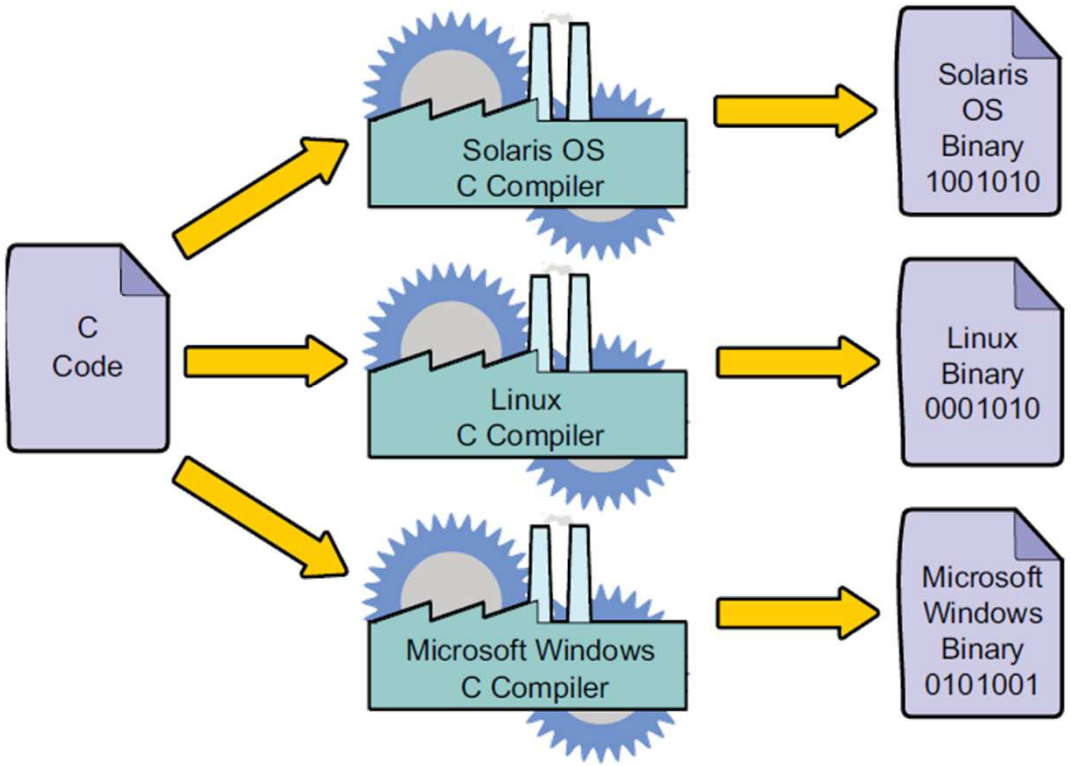
S I M P L E S

- Referências são usadas em vez de apontadores de memória
- O tipo de dados **boolean** pode ter o valor **verdadeiro** ou **falso**
- Gestão de memória automática

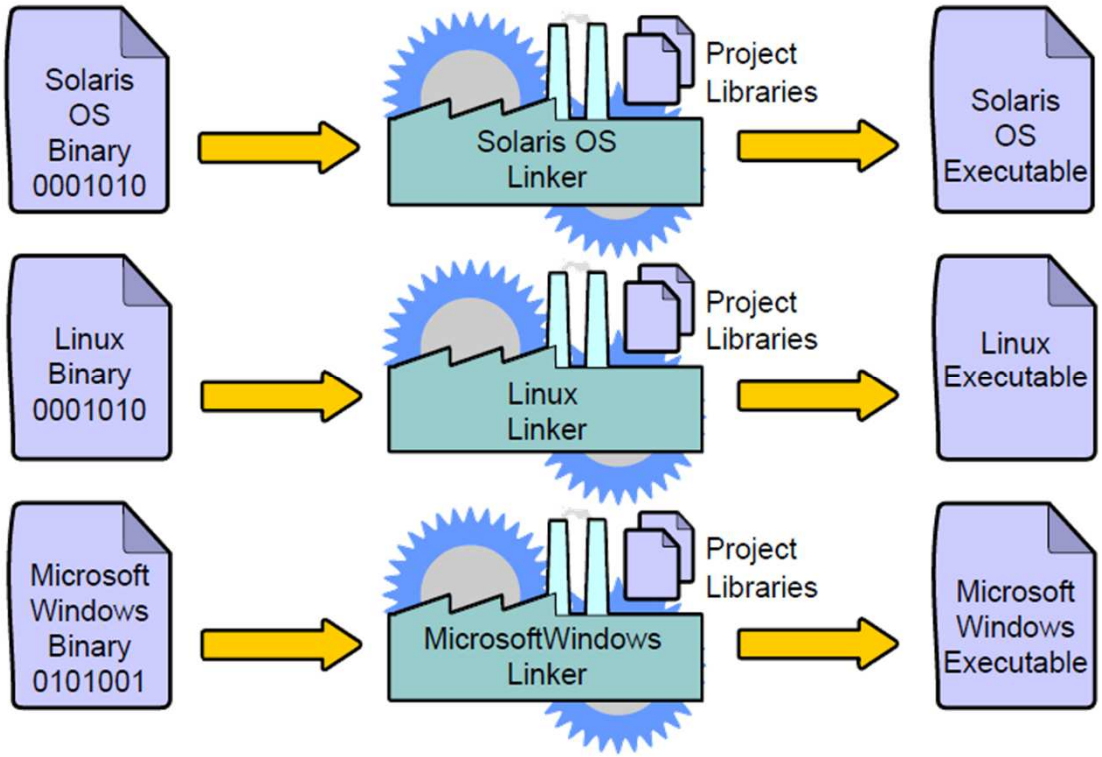


MULTITAREFA

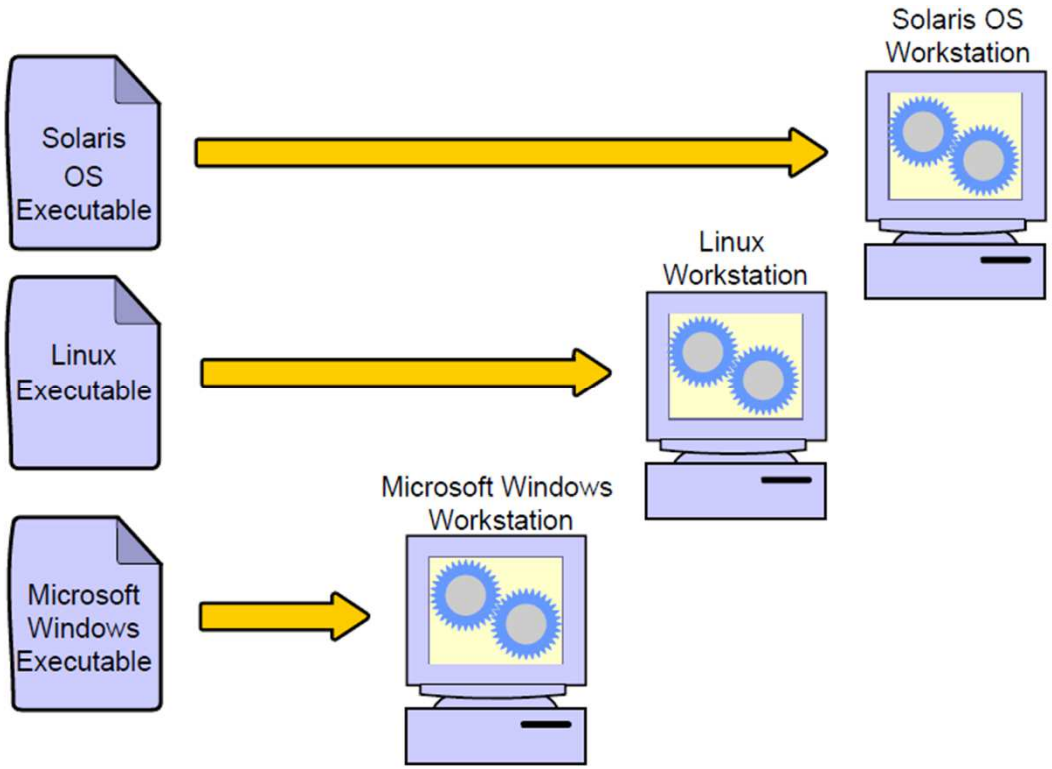
SEGURA



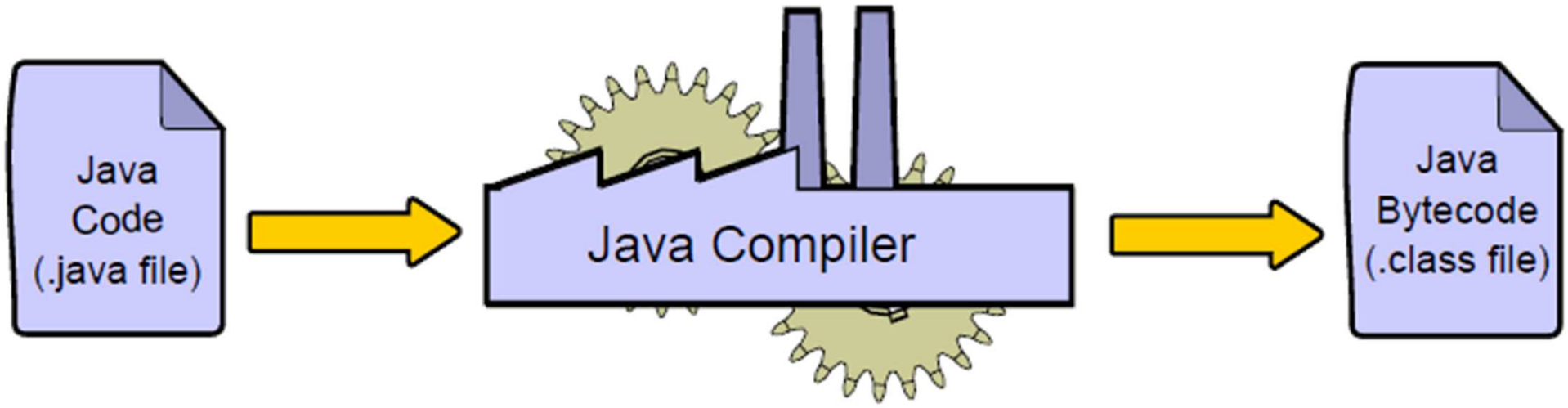
PROGRAMAS DEPENDENTES DE PLATAFORMA



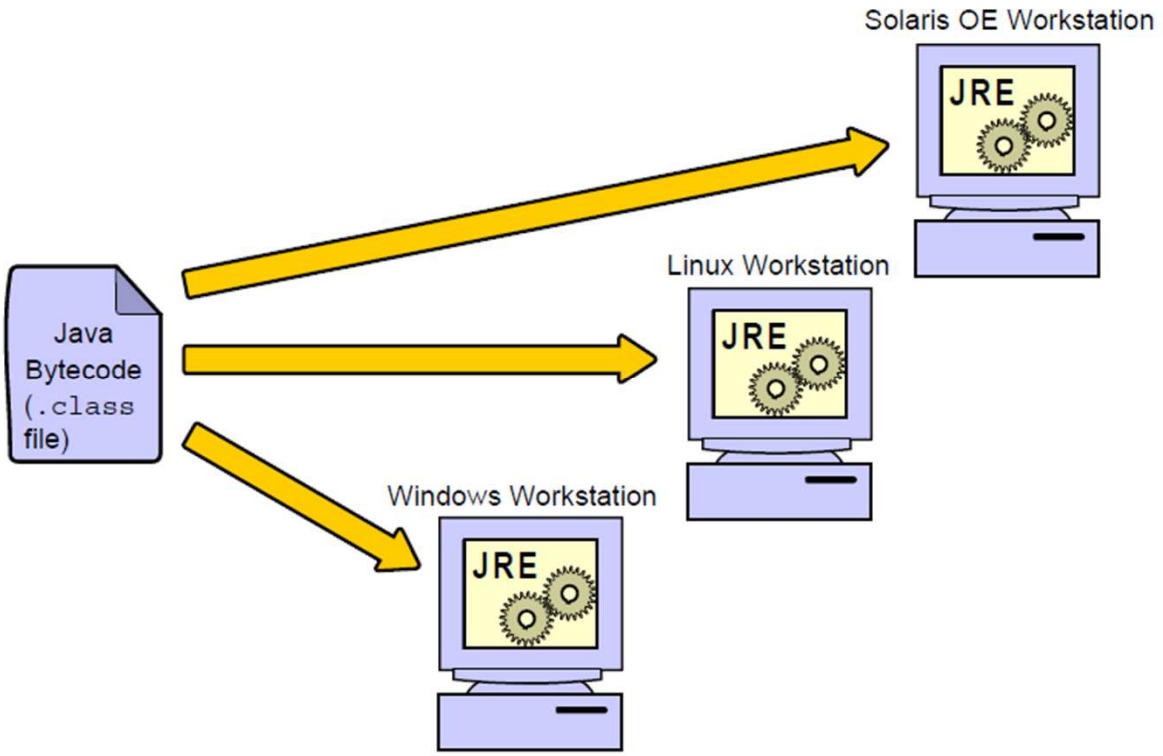
PROGRAMAS DEPENDENTES DE PLATAFORMA



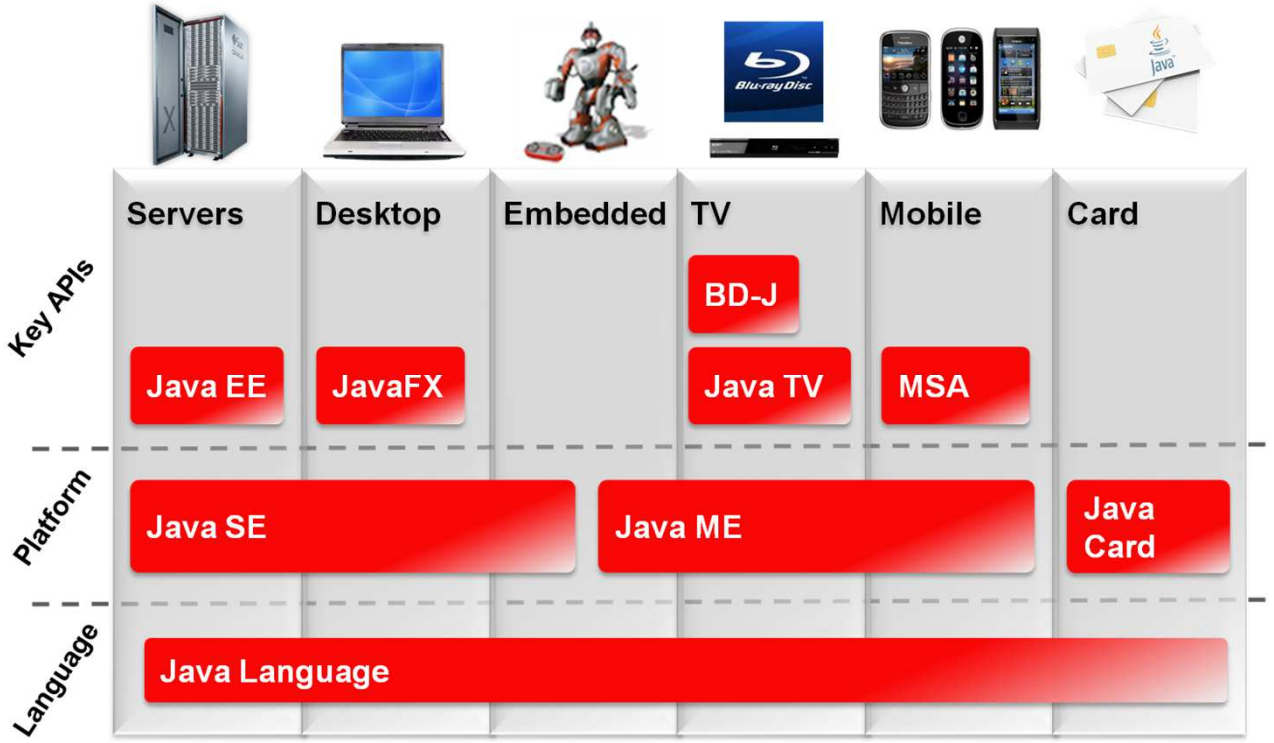
PROGRAMAS DEPENDENTES DE PLATAFORMA



PROGRAMAS INDEPENDTES DE PLATAFORMA



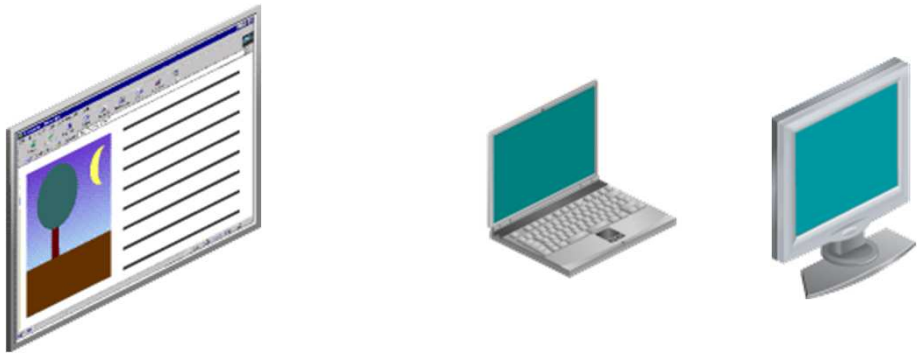
PROGRAMAS INDEPENDTES DE PLATAFORMA



AS DIFERENTES TECNOLOGIAS JAVA

JAVA SE

- É utilizada para desenvolver aplicações que são executadas em browsers e em desktops



JAVA EE

- Usado para criar grandes aplicações distribuídas baseadas em tecnologias server-side e client-side

JAVA ME

- Usado para criar aplicações optimizadas focadas em aparelhos de consumos com poucos recursos

JAVA CARD

- Tipicamente usada nas seguintes áreas (e mais algumas…):
  - Identificação
  - Segurança
  - Transações
  - Cartões SIM

CONFIGURAR AMBIENTE DESENVOLVIMENTO JAVA

1. Descarregar e instalar o Java Development Kit (JDK)

2. Definir o PATH para a pasta de instalação do JDK

3. Compilar e executar uma aplicação Java utilizando a linha de comando

AMBIENTE INTEGRADO DE DESENVOLVIMENTO

- Um ambiente integrado de desenvolvimento (IDE) é uma ferramenta que ajuda o programador a desenvolver aplicações

- Existem vários IDEs disponíveis:
  - Eclipse IDE
  - Netbeans IDE
  - JDeveloper
  - IntelliJ IDEA

- Funcionalidades incluídas:
  - Total integração
  - Deployment fácil
  - Editor inteligente
  - Desenvolvimento de projetos fácil

PRIMEIRO PROGRAMA JAVA

```
public class Hello
{
  public static void main(String[] args)
  {
    System.out.println(“Olá, mundo!");
  }
} // end class Hello
```

COMENTÁRIOS

- Inclua comentários nos seus programas para os tornar mais percetíveis
- Sintaxe de comentário em bloco: /* **...** */
- Sintaxe de comentário de uma linha: //**...**
- O texto comentado é ignorado pelo compilador

NOME DE CLASSES E NOME DE FICHEIRO CÓDIGO-FONTE

- Todos os programas Java devem estar contidos numa classe
- O nome do ficheiro do programa Java deve ter o mesmo nome da classe Java
- A convenção define que os nomes das classes começam com uma letra maiúscula
- Visto que o Java é case-sensitive, o nome do ficheiro também deve começar com uma letra maiúscula
- Isso significa que o compilador Java distingue entre maiúsculas e minúsculas

MÉTODO MAIN

- Memorizar (e utilizar sempre) **public class** antes da nome da classe
- Dentro da sua classe, deve-se incluir um ou mais métodos
- Um método é um grupo de instruções para realizar uma tarefa
- Memorizar (e utilizar sempre) esta assinatura do métdo main:
  - **public static void main(String[] args)**
- Quando um programa se inicia, o computador procura pelo método **main** para iniciar as instruções do programa

CHAVETAS

- Usar **{ }** para agrupar coisas

- Por exemplo, no programa “Olá Mundo”, as chavetas de topo e do fundo agrupam os conteúdos da classe, e as chavetas interiores agrupam conteúdos do método main

IMPRIMIR DADOS NA CONSOLA

- Para gerar um output utilizar **System.out.println()**
  - Por exemplo, para imprimir uma mensagem de olá escrevemos: **System.out.println(“Olá mundo”);**
  - Nota:
    - Colocar o que se quer imprimir dentro dos parêntesis
    - Envolver strings com aspas
    - Colocar ; no fim da instrução System.out.println
- O que significa **ln** no println?

COMPILAÇÃO E EXECUÇÃO

- Para criar um programa Java que pode ser executado num computador, o programa tem de passar por um compilador

- No processo de compilação, o compilador gera um ficheiro **bytecode** que vai ser executado pela JVM

- Nome do ficheiro código-fonte Java = <nome-classe> + .java

- Nome do ficheiro bytecode = <nome-classe> + .class

IDENTIFICADORES

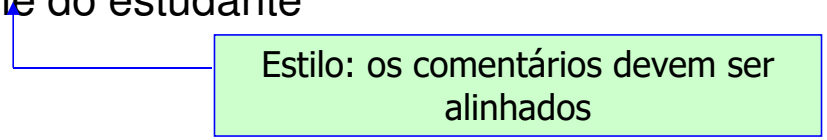
- Identificador = termo técnico para um nome numa linguagem de programação
- Exemplos de identificadores:
  - Nome de classe: Hello
  - Nome de método: main
  - Nome de variável: height
- Convenção de **regras** de nomes:
  - Apenas pode conter letras, dígitos, o sinal dólar ($), e/ou underscore(_)
  - O primeiro caracter não pode ser um dígito
  - Se não forem cumpridas estas regras, o programa não compila

IDENTIFICADORES

- Convenção de identificadores:
  - Estas regras podem ser quebras, mas serão mais difíceis de entender
  - Utilizar letras e dígitos apenas, não $ e _
  - Toda as letras devem ser minúsculas exceto a primeira letra no início das palavras (excetuando a primeira) ex:
    - firstName, x, daysInMonth
  - Nos casos das classes, a primeira letra em cada palavra deve ser maiúscula
  - Os nomes devem ser descritivos

VARIÁVEIS

- Uma variável consegue guardar apenas uma tipo de dado
- Como é que um computador sabe que tipo de dado uma particular variável pode guardar?
  - Antes de uma variável ser usada, o tipo de dado deve ser declarado
- Sintaxe da instrução de declaração: <tipo> <lista de variáveis separadas por vírgulas>;
- Exemplos de declaração:
  - String firstName; //primeiro nome do estudante
  - String lastName; //último nome do estudante
  - int studentId;
  - int row, col;



DECLARAÇÃO DE ATRIBUIÇÃO

- O Java usa o sinal de igualdade (=) para declarações de atribuição
- No fragmento de código posterior, a primeira declaração de atribuição atribui o valor de 50000 na variável salary

```
int salary;
String bonusMessage;
salary = 50000;
bonusMessage = "Bonus = $" + (.02 * salary);
```

Virgulas não são permitidas em números.

Concatenação de strings

DECLARAÇÃO DE ATRIBUIÇÃO

```
int salary;
String bonusMessage;
salary = 50000;
bonusMessage = "Bonus = $" + (.02 * salary);
```

Virgulas não são permitidas em números.

Concatenação de strings

- Reparar no operador + na segunda declaração de atribuição. Se o + aparece entre uma string e outro tipo de dado qualquer, então o operador + efetua uma concatenação de strings

ESQUELETO DE PROGRAMA

- Praticamente todo os fragmentos de código dos slides seguintes poderão ser facilmente convertíveis para programas completos, substituindo <corpo-método> no esqueleto de programa em baixo:

```
public class Test
{
  public static void main(String[] args)
  {
    <corpo-método>
  }
} // end class Test
```

DECLARAÇÃO DE INICIALIZAÇÃO

- Quando se atribui um valor a uma variável como parte da sua declaração

- Sintaxe: **<tipo> <variável> = <valor>;**

- Exemplos:
  - int totalscore = 0;
  - int maxScore = 300;

DECLARAÇÃO DE INICIALIZAÇÃO

- Existe uma forma alternativa de fazer o mesmo que no slide anterior:
  - int totalScore;
  - int maxScore;
  - totalScore = 0;
  - maxScore = 300;

- Ambas as formas podem ser usadas, sem qualquer tipo de consequência

TIPOS DE DADOS NUMÉRICOS – INT, LONG

- Variáveis que guardam valores numéricos inteiros, devem ser normalmente declarados com o tipo – int, long
- Intervalo de valores que podem ser guardados numa variável int: -2 mil milhões até 2 mil milhões
- Intervalo de valores que podem ser guardados numa variável long: -9×10^{18} até 9×10^{18}
- Exemplos de declarações de integers:
  - int studentId;
  - long satelliteDistanceTraveled;
- Recomendação: Usar o tipo de dado mais pequeno para variáveis que não precisam de tipos de dados maiores

TIPOS DE DADOS NUMÉRICOS – FLOAT, DOUBLE

- Variáveis que guardam valores numéricos decimais, devem ser normalmente declarados com o tipo – float, double
- Exemplo:
  - float gpa;
  - double bankAccountBalance;

- O tipo de dado double guardam números decimais de 64 bits, enquanto o tipo de dado float guardam números decimais de 32 bits.
- Significa que as variáveis double são melhores para guardar números com mais casas decimais

TIPOS DE DADOS NUMÉRICOS – FLOAT, DOUBLE

- Recomendações:
  - Deve normalmente declarar as variáveis de vírgula flutuante com o tipo double
  - Principalmente não usar variáveis do tipo float quando estas variáveis forem guardar cálculos monetários ou medidas científicas, pois requerem um nível de precisão muito aprimorado

- Intervalo de valores que podem ser guardados numa variável float: $-3,4 \times 10^{38}$ até $3,4 \times 10^{38}$

- Intervalo de valores que podem ser guardados numa variável double: $-3,4 \times 10^{308}$ até $3,4 \times 10^{308}$

TIPOS DE DADOS BOOLEANOS

- Por vezes é necessário monitorizar o estado de uma condição

- Para implementar o estado de uma condição usamos as variáveis booleanas

- Por exemplo, se estivesse a escrever um programa que simula a operação de uma porta de garagem, tem de monitorizar o estado da direção da porta da garagem – Está a subir ou a descer?
  Se a porta da garagem estiver a subir, ao carregar num botão ela vai descer. No caso de estar a descer, carregando num botão ela deverá subir.

TIPOS DE DADOS BOOLEANOS

up (subir)	true
down (descer)	false

- Uma variável booleana é uma vriável que:
  - É declarada com o tipo boolean
  - Só pode ter dois valores – true ou false

TIPOS DE DADOS BOOLEANOS

- Este fragmento de código utiliza uma variável chamada upDirection inicializada a true, e que intercala o seu valor dentro de um ciclo

```
boolean upDirection = true;
do
{
   ...
   upDirection = !upDirection;
   ...
} while (<user presses the garage door opener button>);
```

se `upDirection` tem o valor `true`, esta instrução altera-o pra `false`, e vice versa.

TIPOS DE DADOS BOOLEANOS- EXEMPLO

```
import java.util.Scanner;

public class GarageDoor
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String entry;                // user's entry - enter key or q
    boolean upDirection = true; // Is the current direction up?
    boolean inMotion = false;   // Is garage door currently moving?

    System.out.println("GARAGE DOOR OPENER SIMULATOR\n");

    do
    {
      System.out.print("Press Enter, or enter 'q' to quit: ");
      entry = stdIn.nextLine();

      if (entry.equals(""))    // pressing Enter generates ""
      {
        inMotion = !inMotion;  // button toggles motion state
```

TIPOS DE DADOS BOOLEANOS- EXEMPLO

```
        if (inMotion)
          {
            if (upDirection)
            {
              System.out.println("moving up");
            }
            else
            {
              System.out.println("moving down");
            }
          }
          else
          {
            System.out.println("stopped");
            upDirection = !upDirection;  // direction reverses at stop
          }
        } // end if entry = ""
      } while (entry.equals(""));
    } // end main
  } // end GarageDoor class
```

ATRIBUIÇÕES ENTRE DIFERENTES TIPOS

- Atribuir um valor inteiro a uma variável com vírgula flutuante é possível:
  - double bankAccountBalance = 1000;

- No sentido contrário isso já não é possível. É como tentar meter um objeto grande numa caixa mais pequena
  - int temperature = 26,7;

- Mesmo este exemplo origina um erro:
  - int count = 0,0;

CONSTANTES

- Uma constante é um valor fixo, que não pode ser alterado
- O tipo por omissão para uma constante numérica inteira é o int
- O tipo por omissão para uma constante numérica de vírgula flutuante é o double
- As constantes dividem-se em duas categorias:
  - Hard-coded: Constantes com um valor definido explicitamente

```
propagationDelay = cableLength / 299792458.0;
```

  - Named: Constantes que têm um nome associado

```
final double SPEED_OF_LIGHT = 299792458.0
```

NAMED CONSTANTS

- A palavra reservada final é um modificador, ou seja, modifica o valor de SPEED_OF_LIGHT para ser fixo

- Todas as named constantes usam a palavra reservada final

- O modificador final indica ao compilador para gerar um erro de compilação se o programa tentar mudar o nome marcada com a palavra reservada final

- A convenção sugere que as named constants devem ter toda as suas letras capitalizadas e utilizar um underscore (_) para separar as palavras em constantes que utilizam várias palavras no seu nome

OPERADORES ARITMÉTICOS

Operador	Operação
+	Soma
-	Subtração
*	Multiplicação
/	Divisão
%	Resto de divisão

OPERADORES ARITMÉTICOS

- A divisão em Java ocorre de forma diferenciada consoante os números/operadores que estejam a ser divididos sejam números inteiros ou números com casas decimais

1º operador	2º operador	Resultado
Número inteiro	Número inteiro	Número inteiro
Número inteiro	Número casas decimais	Números casas decimais
Número casas decimais	Número inteiro	Número casas decimais
Número casas decimais	Número casas decimais	Número casas decimais

DIVISÃO COM CASAS DECIMAIS

- Ou seja, o Java considera que os números com casas decimais são mais “complexos” do que os números inteiros…

- Isto deve ao facto de possuírem a componente fracionária

- Como visto na tabela anterior, sempre que uma divisão possui ambos os tipos de dados numéricos, o Java “promove” temporariamente o tipo de dado menos complexo para mais complexo de forma a poder efetuar a divisão

OPERADORES DE INCREMENTO E DECREMENTO

- Utilize o operador de incremento (++) para somar a uma variável mais 1
- Utilize o operador de decremento (--) para subtrair a uma variável menos 1

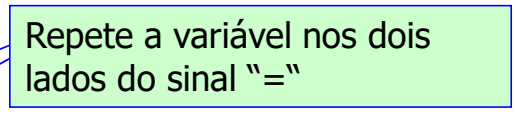
```
x++;    ≡    x = x + 1;
x--;    ≡    x = x - 1;
```

- A convenção diz que se deve utilizar estes operadores (incremento/decremento) em vez da forma “tradicional” para efetuar a soma/subtração de 1 a uma variável

OPERADORES DE ATRIBUIÇÃO COMPOSTOS

- Os operadores de atribuição compostos são:
  - +=, -=, *=, /=, %=

- A variável é atribuída com uma versão atualizada do valor original da variável

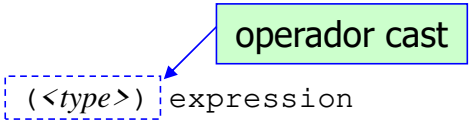



```
x += 3;        ≡    x = x + 3;
x -= 4;        ≡    x = x - 4;
```

- A convenção sugere que os operadores de atribuição compostos devem ser usados nestes casos

CASTING ENTRE TIPOS DE DADOS

- Por vezes num programa é necessário converter um valor para outro tipo de dado. A isso chama-se casting



- Imagine que tem uma variável denominada “interest” que guarda juros bancários como um double. Gostaria de extrair a parte dos dólares dos juros bancários e guardá-los numa variável do tipo int, com o nome interestInDollars:

```
interestInDollars = (int) interest;
```

CASTING ENTRE TIPOS DE DADOS

- Se alguma vez for necessário efetuar o cast a mais do que um simples valor ou variável, garanta que utiliza os parêntesis a englobar todos os valores alvos do cast

```
double interestRate;
double balance;
int interestInDollars;
...
interestInDollars = (int) (balance * interestRate);
```

Parêntesis não necessários aqui

TIPOS DE DADOS NÃO NUMÉRICOS - CHAR

- O tipo de dado char corresponde apenas um caracter

- É envolvida por plicas: ‘B’, ‘1’, ‘:’

- Exemplo:

```
char first, middle, last;
first = 'J';
middle = 'S';
last = 'D';
System.out.println("Hello, " + first + middle + last + '!');
```

CARACTERES DE “ESCAPE”

- Caracteres de “escape” são constantes do tipo char para imprimir caracteres com o enter ou um tab
- Um caracter de “escape” é sempre precedido de um “backslash” (\)
- O mais comuns:
  - \n – nova linha. Vai para a primeira coluna da próxima linha
  - \t – move o cursor para a próxima “paragem” do tabulador
  - \\ - imprime um “backslash”
  - \” – imprime aspas
  - \’ – imprime uma plica

VARIÁVEIS PRIMITIVAS VS. VARIÁVEIS DE REFERÊNCIA

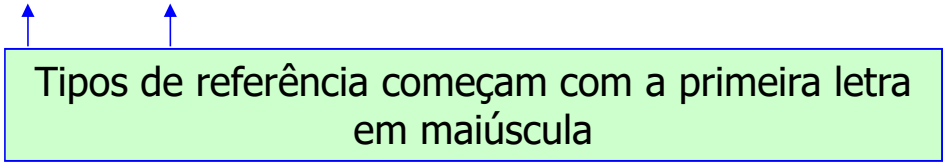
- Existe duas categorias básicas de variáveis em Java – variáveis primitivas e variáveis de referência

- As variáveis primitivas apenas podem guardar um valor de cada vez. As variáveis primitivas são declaradas com tipos de dados primitivos/básicos:
  - int, long (valores numéricos inteiros)
  - float, double (valores numéricos com parte decimal)
  - char (caracteres)

VARIÁVEIS PRIMITIVAS VS. VARIÁVEIS DE REFERÊNCIA

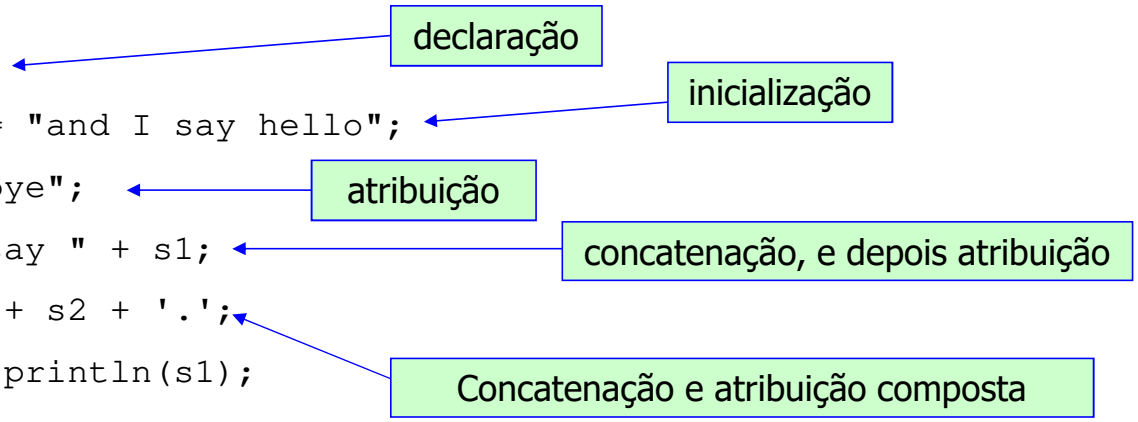
- As variáveis de referência são mais complexas. Elas podem guardar um grupo relacionado de dados

- Variáveis de referência são declaradas com um tipo de dado de referência/complexo
  - String, Calendar, classes definidas pelo programador



O BÁSICO SOBRE STRINGS

```
String s1;
String s2 = "and I say hello";
s1 = "goodbye";
s1 = "You say " + s1;
s1 += ", " + s2 + '.';
System.out.println(s1);
```



MÉTODOS DE STRINGS

- Método charAt:
  - Devolve o caracter de uma string numa dada posição
  - As posições de caracteres de uma string têm todos um índice associado
  - O primeiro índice de um caracter de uma string é o zero

```
String animal = "cow";
System.out.println("Last character: " + animal.charAt(2));
```

Para usar o método incluir: a referência da variável, o operador . (ponto), nome do método, parêntesis e argumentos

MÉTODOS DE STRINGS

- Método length:
  - Devolve o número de caracteres numa string

```
String s = "hi";
System.out.println(s.length()); //2
```

MÉTODOS DE STRINGS

- Para comparar strings para igualdade, deve-se utilizar o método equals.
- Para o mesmo efeito, mas ignorando a capitalização de um string, deve-se utilizar o método equalsIgnoreCase

```
public class Test
{
  public static void main(String[] args)
  {
    String animal1 = "Horse";
    String animal2 = "Fly";
    String newCreature;
    newCreature = animal1 + animal2;

    System.out.println(newCreature.equals("HorseFly"));
    System.out.println(newCreature.equals("horsefly"));
    System.out.println(newCreature.equalsIgnoreCase("horsefly"));
  }
}
```

INPUT – CLASS SCANNER

- Para já pouco sabemos ainda sobre classes
- Mas várias coisas já sabemos:
  - São amplamente utilizadas em Java
  - Como são variáveis do tipo de referência a primeira letra do seu tipo de dado começa com uma letra maiúscula
  - Iremos trabalhar com elas durante a formação

- Uma das classes com que se vai trabalhar vai ser a classe Scanner

INPUT – CLASSE SCANNER

- Esta classe vai permitir-nos obter dados que um utilizador insira nos nossos programas
- Para dizermos ao compilador de Java que vamos querer utilizar esta classe nos nossos programas, temos que “importar” esta classe para ela ficar disponível para a utilizarmos:

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

- Depois, temos que indicar dentro no método main a sua declaração e inicialização:

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

- Por fim já é possível ler e guardar o valor de uma variável utilizando o método nextLine:

```
<variable> = stdIn.nextLine();
```

INPUT – CLASSE SCANNER

```
import java.util.Scanner;

public class FriendlyHello
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String name;
    System.out.print("Enter your name: ");
    name = stdIn.nextLine();
    System.out.println("Hello " + name + "!");
  }
}
```

Estas duas instruções criam uma monitorização ao teclado

Usar o métdo print (sem "ln")

Obtemos uma linha de output

INPUT – CLASSE SCANNER

- Para além do método nextLine, a classe Scanner possui mais métodos para obter diferentes tipos de input, por exemplo:
  - nextInt() – Procura por um valor do tipo int, e devolve esse valor
  - nextLong() – Procura por um valor do tipo long, e devolve esse valor
  - nextFloat() – Procura por um valor do tipo float, e devolve esse valor
  - nextDouble() – Procura por um valor do tipo double, e devolve esse valor
  - next() – Procura por um valor do tipo String, e devolve esse valor

INPUT – CLASSE SCANNER

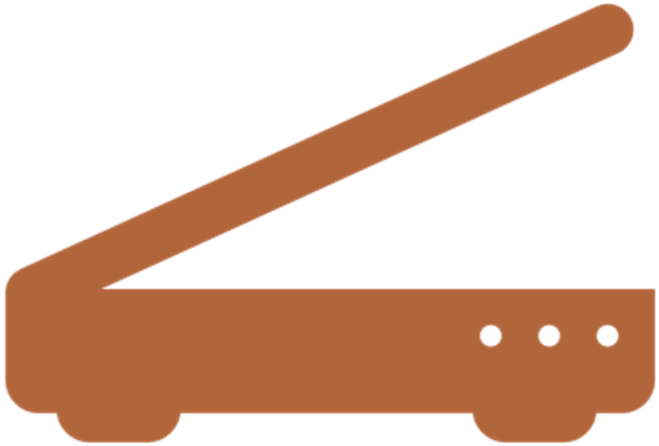
Exemplo de utilização do método nextDouble e nextInt:

```
import java.util.Scanner;

public class PrintPO
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    double price;  // price of purchase item
    int qty;       // number of items purchased

    System.out.print("Price of purchase item: ");
    price = stdIn.nextDouble();
    System.out.print("Quantity: ");
    qty = stdIn.nextInt();
    System.out.println("Total purchase order = $" + price *
  qty);
  }
}
```

INPUT – CLASSE SCANNER



Exemplo de utilização do método next:

```
import java.util.Scanner;

public class PrintInitials
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String first;  // first name
    String last;   // last name

    System.out.print("Enter first and last name separated by a space: ");
    first = stdIn.next();
    last = stdIn.next();
    System.out.println("Your initials are " +
      first.charAt(0) + last.charAt(0) + ".");
  }
}
```

OPERADORES RELACIONAIS

Condição	Operador	Exemplo
É igual a	==	int i=1; (i == 1)
É diferente de	!=	int i=2; (i != 1)
É menor que	<	int i=0; (i < 1)
É menor ou igual que	<=	int i=1; (i <= 1)
É maior que	>	int i=2; (i > 1)
É maior ou igual que	>=	int i=1; (i >= 1)

CONDIÇÕES

- A partir de agora iremos ver instruções condicionais if e switch, tal como ciclos que terão as suas condições dentro de parêntesis:

```
if (<condition>)
{
  ...
}

while (<condition>)
{
  ...
}
```

- Tipicamente cada condição involve algum tipo de comparação, e as comparações utilizam operadores relacionais

INSTRUÇÃO IF

- Usar uma instrução if se é necessário efetuar uma questão sobre o que se quer fazer a seguir

- Existem 3 formas diferentes para esta instrução:
  - if: Usar em problemas que se quer fazer alguma coisa ou nada
  - if, else: Usar em problemas que se quer fazer uma coisa ou outra coisa
  - if, else, if: Usar em problemas que se quer fazer uma coisa entre três possíveis

INSTRUÇÃO IF

- `if`:

```
if <condition>
   <statement(s)>
```

- `if, else`:

```
if <condition>
   <statement(s)>
else
   <statement(s)>
```

- `if`:

```
if (<condition>)
{
   <statement(s)>
}
```

- `if, else`:

```
if (<condition>)
{
   <statement(s)>
}
else
{
   <statement(s)>
}
```

INSTRUÇÃO IF-PSEUDOCÓDIGO

`if, else if:`

```
if <condition>
    <statement(s)>
else if <condition>
    <statement(s)>
  .
  .
  .
else
    <statement(s)>
```

mais else se forem necessários (opcional)

opcional

INSTRUÇÃO IF-JAVA

`if, else if, else:`

```
if (<condition>)
{
    <statement(s)>
}
else if (<condition>)
{
    <statement(s)>
}
 .
 .
 .
else
{
    <statement(s)>
}
```

mais else se forem necessários (opcional)

opcional

DESAFIO: INSTRUÇÃO IF

- Escrever um programa completo que peça ao utilizador para escrever uma frase e depois escrever uma mensagem de erro caso o último caracter não seja um ponto final

NOMENCLATURA IF

```
if (boolean_expression) {
      code_block;
}
```

NOMENCLATURA IF/ELSE

```
if (boolean_expression) {
                    <code_block1>
}
else if (boolean_expression){
                    <code_block2>
}
else {
                    <code_block3>
}
// programa continua aqui
```

NOMENCLATURA IF/ELSE ENCADEADOS

```
public void calculateNumDays() {
      if (month == 1 || month == 3 || month == 5 || month == 7 || month == 8 || month == 10 || month == 12) {
                System.out.println("There are 31 days in that month.");
      }
      else if (month == 2) {
        System.out.println("There are 28 days in that month.");
      }
       else if (month == 4 || month == 6 || month == 9 || month == 11) {
           System.out.println("There are 30 days in that month.");
       }
       else {
         System.out.println("Invalid month.");
       }
```

TESTE DE IGUALDADE ENTRE STRINGS

```
public class CarCollection {

  public String car1 = "Renault 4L";

  public String car2 = "Citroen 2CV";


  public void areCarModelsEqual() {

      if (car1.equals(car2)) {

        System.out.println("Same car.");

    }

      else {

      System.out.println("Different cars.");

      }

    }

}
```

OPERADORES LÓGICOS

Operação	Operador	Exemplo

Se uma condição **E** outra condição	&&	int i = 2;
		int j = 8;
		
		((i < 1) && (j > 6))

Se uma condição **OU** outra condição	| |	int i = 2;
		int j = 8;
		
		((i < 1) || (j > 10))

NEGAÇÃO	!	int i = 2; (! (i < 3))

OPERADOR LÓGICO &&

- Suponhamos que queremos imprimir “OK” se a temperatura está compreendida entre 50 e 90 graus e imprimir “not OK” se assim não for.

- A solução em pseudocódigo:

```
if temp ≥ 50 and ≤ 90
    print "OK"
else
    print "not OK"
```

OPERADOR LÓGICO &&

- E aqui está a solução usando Java:

```
if (temp >= 50 && temp <= 90)
{
    System.out.println("OK");
}
else
{
    System.out.println("not OK");
}
```

- Em Java, se dois critérios são necessários que têm de ser satisfeitos, então separar os dois critérios com o operador && (e). Se ambos os critérios usarem a mesma variável, deve ser incluída essa variável em ambos os lados do &&

OPERADOR LÓGICO &&

- O programa do próximo slide determina se um adepto num jogo de basquetebol vai ganhar batatas fritas. Se a equipa da casa ganhar e marcar pelo menos 100 pontos, então deverá aparecer uma mensagem a informar o adepto que ganhou as batatas fritas.

- No próximo slide, substituir *<inserir código aqui>* com o código apropriado.

OPERADOR LÓGICO &&

```
import java.util.Scanner;

public class FreeFries
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    int homePts;
    int opponentPts;

    System.out.print("Home team points scored: ");
    homePts = stdIn.nextInt();

    System.out.print("Opposing team points scored: ");
    opponentPts = stdIn.nextInt();

    <inserir código aqui>

  }
}
```

OPERADOR LÓGICO | |

- Escrever código que imprima “bye” se a variável “response” contiver apenas a letra q (de quit) em maiúscula ou minúscula.

```
if (response.equals("q") || response.equals("Q"))
{
  System.out.println("bye");
}
```

Quando se utiliza o operador ||, se ambos os critérios na condição ou, usam a mesma variável, ela tem de estar presente nos dois lados

OPERADOR LÓGICO | |

- Uma alternativa ao exemplo anterior é utilizar o método das strings “equalsIgnoreCase”

```
if (response.equalsIgnoreCase("q"))
{
  System.out.println("Bye");
}
```

OPERADOR LÓGICO !

- O operador ! (negação) inverte a verdade ou a falsidade de uma condição

```
if (!(reply == 'q' || reply == 'Q'))
{
  System.out.println("Let's get started....");
  ...
```

OPERADOR TERNÁRIO

Operação	Operador	Exemplo
Se <uma condição> é verdadeira, atribuir o valor de value1 ao resultado. Caso contrário, atribuir o valor de value2 ao resultado	? :	<uma condição> ? value1 : value2

```
int val1 = 10;
int val2 = 20;

int max = val1 >= val2 ? val1 : val2;
```

INSTRUÇÃO SWITCH

- Quando utilizar esta instrução? – Se precisa de fazer uma, de várias possibilidades

- Ou seja, a instrução switch é uma forma mais organizada de efeutar if/else encadeados

```
switch (<controlling-expression>)
{
  case <constant1>:
    <statements>;
    break;
  case <constant2>:
    <statements>;
    break;
  ...
  default:
    <statements>;
}
```

INSTRUÇÃO SWITCH

- Como funciona:
  - Salta para a expressão “case” que corresponde à expressão a avaliar (ou para a expressão “default” se não houver correspondência) e executa as instruções associadas até encontrar a expressão “break”
  - A expressão “break” faz com que se saia da instrução switch
  - Normalmente a expressão “break” encontra-se no fim de um bloco “case”, mas não é obrigatório que assim seja
  - Colocar : imediatamente a seguir de cada “case”
  - Não é necessário usar chavetas em cada “case”

INSTRUÇÃO SWITCH-EXEMPLO

```
i = stdIn.nextInt();
switch (i)
{
  case 1:
    System.out.print("A");
    break;
  case 2:
    System.out.print("B");
  case 3: case 4:
    System.out.print("C-D");
    break;
  default:
    System.out.print("E-Z");
}
```

INSTRUÇÃO SWITCH-DESAFIO

- Escrever um programa que leia o ZIP Code (o equivalente ao Código Postal nos EUA), e utilize o primeiro digito para dizer qual a área geográfica associada:

Se código ZIP começa com	imprimir esta mensagem
0, 2, 3	*<zip>* `is on the East Coast.`
4-6	*<zip>* `is in the Central Plains area.`
7	*<zip>* `is in the South.`
8-9	*<zip>* `is in the West.`
other	*<zip>* `is an invalid ZIP Code.`

INSTRUÇÃO SWITCH-DESAFIO

```
Scanner sc = new Scanner(System.in);
System.out.println("Insira o zip code:");
String number = sc.nextLine();
switch (number.substring(0,1)) {
    case "0":
    case "2":
    case "3":
        System.out.println("is on East Coast");
        break;
    case "4":
    case "5":
    case "6":
        System.out.println("is in the Central Plains area");
        break;
    case "7":
        System.out.println("is in the South");
        break;
    case "8":
    case "9":
        System.out.println("is in the west");
        break;
    default:
        System.out.println("is an invalid zip code.");
        break;
}
```

INSTRUÇÃO SWITCH

```
switch (variable) {
    case literal_value:
              <code_block>
              [break;]
    case another_literal_value:
        <code_block>
              [break;]
    [default:]
              <code_block>
}
```

INSTRUÇÃO SWITCH-EXEMPLO

```
public class SwitchDate {
              public int month = 10;
              public void calculateNumDays() {
                switch(month) {
                case 1:
                case 3:
                case 5:
                case 7:
                case 8:
                case 10:
                case 12:
                  System.out.println("There are 31 days in that month.");
                  break;
}
```

- Testes de igualdade
- Testes contra um valor singular, por exemplo “customerStatus”
- Testes contra um valor do tipo int, short, byte, char e String
- Testes contra um valor fixo conhecido em tempo de compilação

QUANDO UTILIZAR O SWITCH

INTRODUÇÃO A ARRAYS

Um array é um objeto que contém um grupo de valores de um dado tipo

Um valor num array pode ser do tipo primitivo ou um do tipo referência

O comprimento de um array é estabelecido quando o array é criado

Após a criação o comprimento do array não pode ser mudado

INTRODUÇÃO A ARRAYS

Cada item de um array é designado de elemento

Cada elemento é acedido por um índice numérico

O primeiro índice tem o valor 0 (zero)

ARRAYS UNI-DIMENSIONAIS

- Exemplo:

```
int ageOne = 27;

int ageTwo = 12;

int ageThree = 82;

int ageFour = 70;

int ageFive = 54;
```

```
int ageSix = 6;

int ageSeven = 1;

int ageEight = 30;

int ageNine = 34;

int ageTen = 42;
```

CRIAR ARRAYS UNI-DIMENSIONAIS

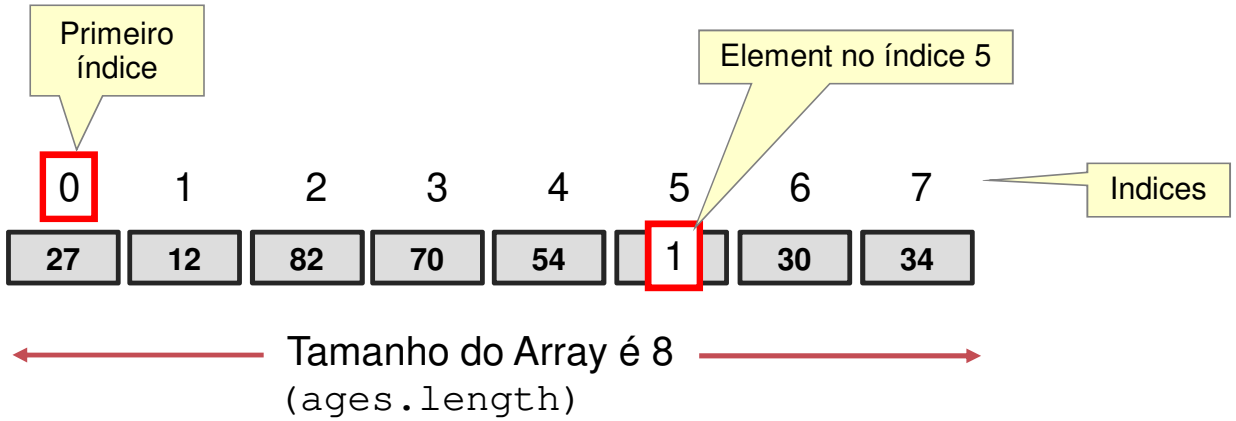
Array do tipo `int`

Array do tipo Shirt

Array do tipo String

ÍNDICES DE ARRAYS E COMPRIMENTO

- Array com o nome ages, com oito elementos



DECLARAR ARRAY UNI-DIMENSIONAL

- Sintaxe:

```
tipo_dado [] identificador_array;
```

- Declarar arrays do tipo char e int:

```
char [] status;
int [] ages;
```

- Declarar arrays de tipos de referência Shirt e String:

```
Shirt[] shirts;
String [] names;
```

INSTANCIAR ARRAY UNI-DIMENSIONAL

- Sintaxe:

```
identificador_array = new tipo_dado [comprimento];
```

- Exemplos:

```
status = new char[20];

ages  = new int[5];

names = new String[7];

shirts = new Shirt[3];
```

INICIALIZAR ARRAY UNI-DIMENSIONAL

- Sintaxe:

```
identificador_array[índice] = valor;
```

- Definir valores no array:

```
ages[0] = 19;
ages[1] = 42;
ages[2] = 92;
ages[3] = 33;
```

- Definir referências para os objetos Shirt no array shirts:

```
shirts[0] = new Shirt();
shirts[1] = new Shirt();
shirts[2] = new Shirt();
```

DECLARAR, INSTANCIAR, E INICIALIZAR ARRAYS UNI-DIMENSIONAIS

- Sintaxe:

```
tipo_dado [] identificador_array = { lista_valores_separados_por_vírgulas };
```

- Exemplos:

```
int [] ages = {19, 42, 92, 33, 46 };
Shirt [] shirts = { new Shirt(), new Shirt(), new Shirt() };
```

- NÃO PERMITIDO!!

```
int [] ages;
ages = {19, 42, 92, 33, 46 };
```

ACEDER A VALORES DE UM ARRAY

- Definir um valor:

```
status[0] = ‘3’;
names[1] = “Fred Smith”;
ages{[1] = 19;
```

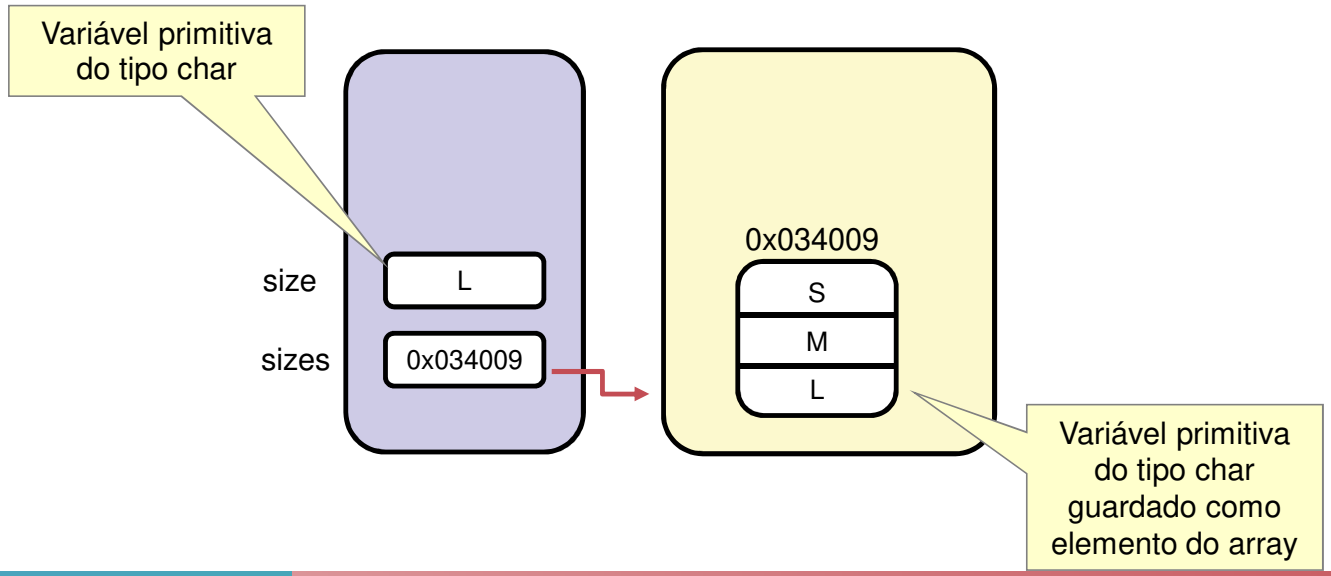
- Obter um valor:

```
char s = status[0];
string name = names[1];
int ages = ages[1];
```

GUARDAR ARRAYS EM MEMÓRIA

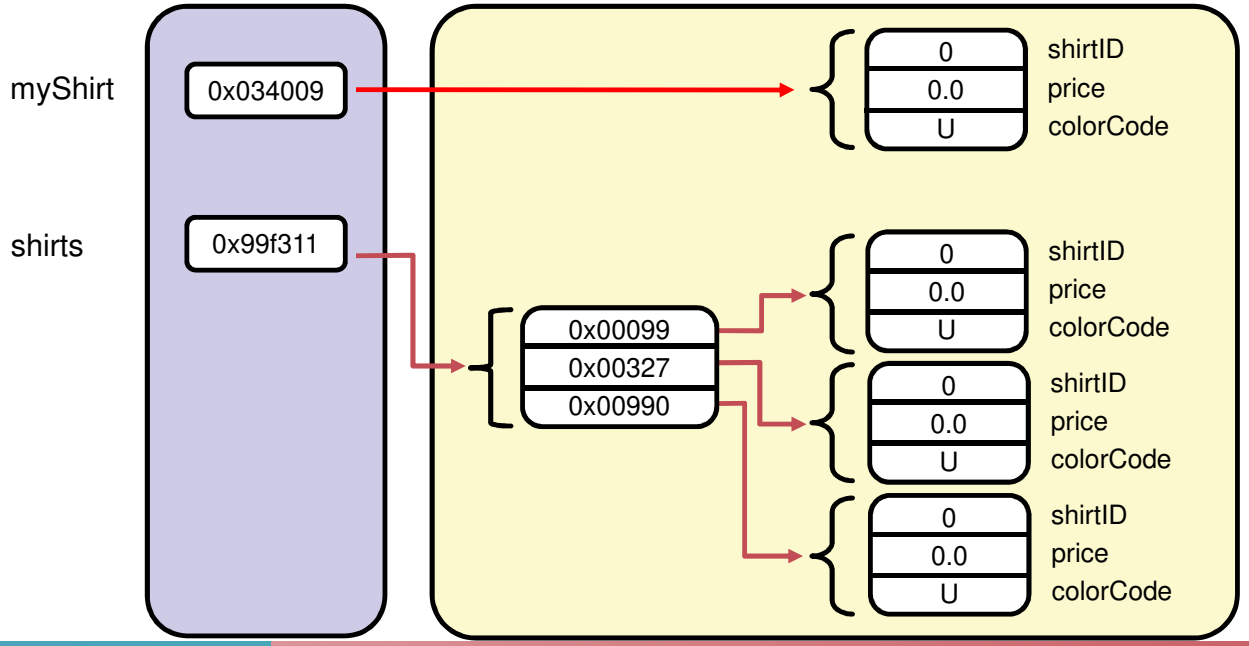
char size = 'L'

char[] sizes = {'S', 'M', 'L' };



GUARDAR ARRAYS DO TIPO REFERÊNCIA EM MEMÓRIA

```
Shirt myShirt = new Shirt();
Shirt[] shirts = { new Shirt(), new Shirt(), new Shirt() };
```



USAR O ARRAY ARGS NO MÉTODO MAIN

- Os parâmetros podem ser escritos na linha de comandos

```
> java ArgsTest Hello World!
   args[0] is Hello
   args[1] is World!
```

O segundo parâmetro vai em `args[1]`

O primeiro parâmetro vai em `args[0]`

- Código para obter esses parâmetros:

```
public class ArgsTest {
   public static void main (String[] args) {
      System.out.println("args[0] is " + args[0]);
      System.out.println("args[1] is " + args[1]);
   }
}
```

CONVERTER ARGUMENTOS STRING PARA OUTROS TIPOS

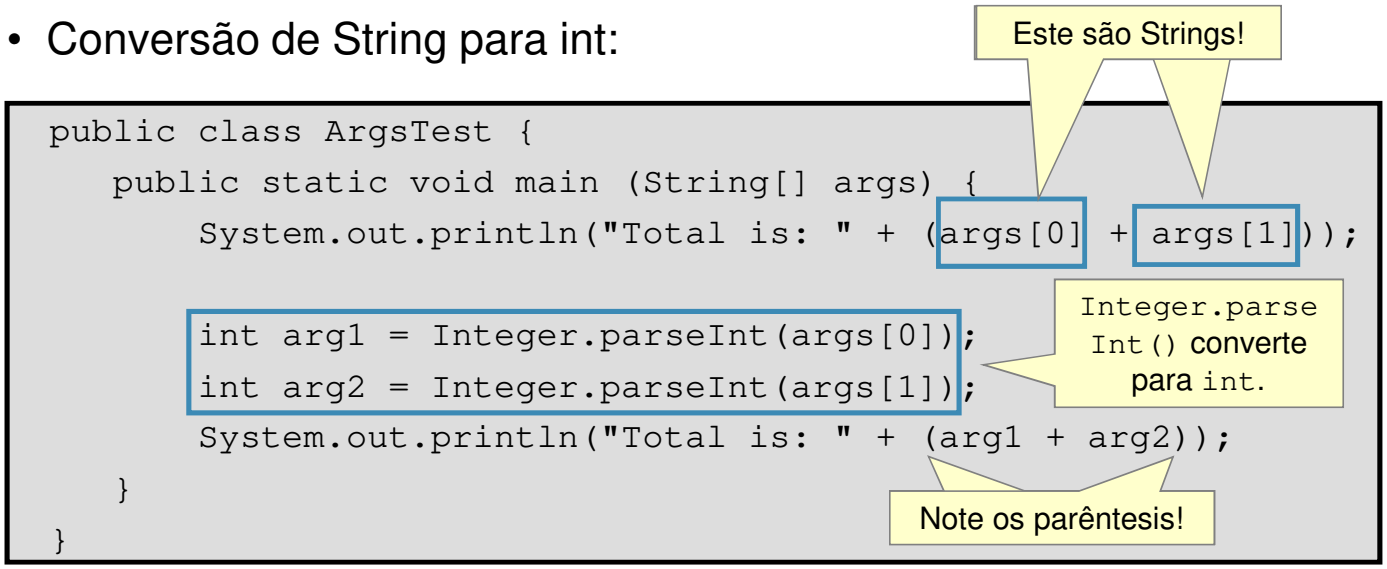
- Números podem ser escritos como parâmetros:

```
> java ArgsTest 2 3
Total is: 23
Total is: 5
```

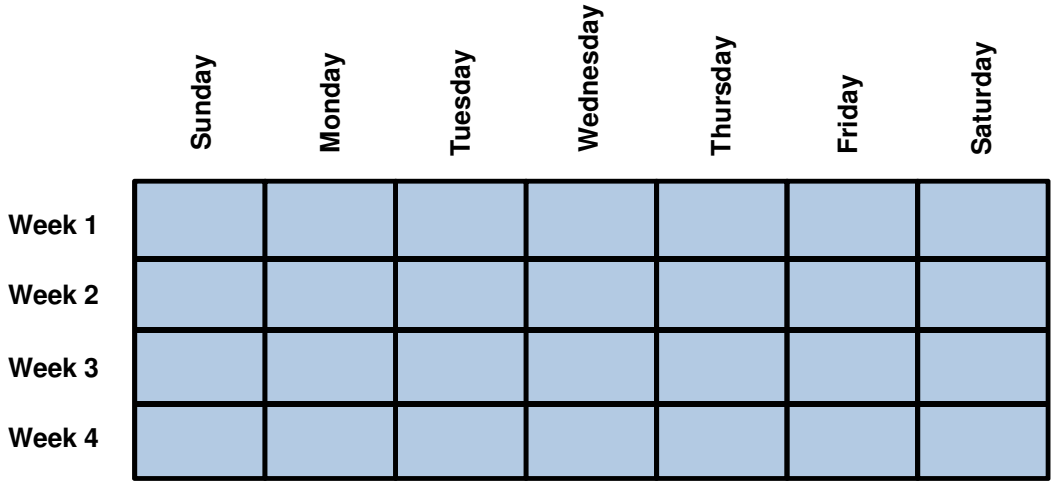
Concatenação, não adição!

CONVERTER ARGUMENTOS STRING PARA OUTROS TIPOS

- Conversão de String para int:



DESCREVER ARRAYS MULTI-DIMENSIONAIS



DECLARAR ARAYS MULTI-DIMENSIONAIS

- Sintaxe:

```
tipo_dado [][] identificador_array;
```

- Exemplo:

```
int [][] yearlySales;
```

INSTANCIAR ARRAY MULTI-DIMENSIONAL

- Syntax:

```
identificador_array = new tipo [numero_arrays] [tamanho];
```

- Exemplo:

```
// Instancia o 2D array: 5 arrays de 4 elementos cada

yearlySales = new int[5][4];
```

INSTANCIAR ARRAY MULTI-DIMENSIONAL

	Quarter 1	Quarter 2	Quarter 3	Quarter 4
Year 1				
Year 2				
Year 3				
Year 4				
Year 5				

INICIALIZAR UM ARRAY MULTI-DIMENSIONAL-EXEMPLO

```
• yearlySales[0][0] = 1000;
• yearlySales[0][1] = 1500;
• yearlySales[0][2] = 1800;
• yearlySales[1][0] = 1000;
• yearlySales[3][3] = 2000;
```

INICIALIZAR UM ARRAY MULTI-DIMENSIONAL-EXEMPLO

	Quarter 1	Quarter 2	Quarter 3	Quarter 4
Year 1	1000	1500	1800	
Year 2	1000			
Year 3				
Year 4				2000
Year 5				

CLASSE ARRAYLIST

- Os arrays não são a única maneira de guardar conjuntos de dados relacionados:
  - O ArrayList é uma das muitas classes possíveis de o fazer
  - O ArrayList tem um conjunto de métodos úteis para gerir os seus elementos:
    - add(), get(), remove(), indexOf(),...
  - Num ArrayList não precisa de especificar o tamanho deste quando o instancia:
    - O ArrayList vai crescendo (ou diminuindo) em tamanho, consoante a necessidade
    - Pode-se definir um tamanho inicial, mas não é obrigatório fazê-lo
  - Um ArrayList, só pode guardar objetos, e não tipos de dados primitivos

NOMES DE CLASSES E IMPORT

- O ArrayList está no package java.util

- Exemplo:

```
import java.util.ArrayList;
    public class ArrayListExample {
       public static void main (String[] args) {
           ArrayList myList;
       }
   }
```

```
ArrayList myList;
```

Declarar a referência.

```
myList = new ArrayList();
```

Instanciar o ArrayList.

```
myList.add("John");
myList.add("Ming");
myList.add("Mary");
myList.add("Prashant");
myList.add("Desmond");
```

Inicializar o ArrayList.

```
myList.remove(0);
myList.remove(myList.size()-1);
myList.remove("Mary");
```

Modificar the ArrayList.

```
System.out.println(myList);
```

TRABALHAR COM UM ARRAYLIST

CICLOS

- Ciclos são frequentemente utilizados em programas para repetir blocos de instruções até uma expressão ser falsa

- Existem três tipos principais de ciclos:
  - Ciclo **while**: Repete enquanto uma expressão for verdadeira
  - Ciclo **do**/**while**: Executa uma vez, e depois continua a repetir enquanto uma expressão for verdadeira
  - Ciclo **for**: Repete um número pré-determinado de vezes

CICLO WHILE

Sintaxe Pseudocódigo

```
while <condition>
  <statement(s)>
```

Sintaxte Java

```
while (<condition>)
{
   <statement(s)>
}
```

REPETIR INSTRUÇÕES


```
while (!areWeThereYet) {

  read book;
   argue with sibling;
   ask, "Are we there yet?";

}

Woohoo!;
Get out of car;
```

CICLO WHILE

- Escrever um método main que encontra a soma de números inseridos pelo utilizador onde o número -99999 será o número que interrompa essa soma, e apresenta-a.


```
public static void main(String[] args)
{
  Scanner stdIn = new Scanner(System.in);
  int sum = 0;          // sum of user-entered values
  int x;                // a user-entered value
  System.out.print("Enter an integer (-99999 to
   quit): ");
  x = stdIn.nextInt();
  while (x != -99999)
  {
    sum = sum + x;
    System.out.print("Enter an integer (-99999 to
   quit): ");

    x = stdIn.nextInt();
  }

  System.out.println("The sum is " + sum);
} // end main
```

CRIAR CICLOS WHILE

```
while (expressão booleana) {
        code_block;
}
// programa continua aqui
```

Se a expressão booleana é verdaeira, este bloco de código executa

Se a expressão booleana é falsa, o programa continua aqui

CICLO WHILE – EXEMPLO 1

```
float square = 4;    // number to find sq root of
   float squareRoot = square;    // first guess
   while (squareRoot * squareRoot - square > 0.001) { // How accurate?
       squareRoot = (squareRoot + square/squareRoot)/2;
       System.out.println("Next try will be " + squareRoot);
   }
   System.out.println("Square root of " + square + " is " + squareRoot);
```

CICLO WHILE – EXEMPLO 1

```
Next try will be 2.5
Next try will be 2.05
Next try will be 2.0006099
Next try will be 2.0
The square root of 4.0 is 2.0
```

CICLO WHILE-EXEMPLO 2

```
int initialSum = 500;
    int interest = 7;          // per cent
    int years = 0;
    int currentSum = initialSum * 100; // Convert to pennies
       while ( currentSum <= 100000 ) {
           currentSum += currentSum * interest/100;
           years++;
           System.out.println("Year " + years + ": " + currentSum/100);
       }
```

CICLO WHILE-EXEMPLO 2

```
... < some results not shown > ...
Year 9: 919
Year 10: 983
Year 11: 1052
```

CICLO DO-WHILE

- Quando utilizar o ciclod do-while:
  - Se o programador souber que pretende executar pelo menos uma vez um conjunto de instruções que pretende repetir

- Nota:
  - A condição está no fim do ciclo (ao contrário do ciclo while, em que a condição está no topo)
  - O compilador exige que “;” esteja no fim da condição do ciclo
  - A convenção diz que a condição deverá estar ao mesmo nível da última chaveta

CICLO DO-WHILE

```
do {
      bloco_código;
}
while (expressão_booleana);   // ponto e vírgula obrigatório
```

CICLO DO-WHILE

- Problema:
  - Como parte de um programa de arquitectura, escrever um método main que pergunte ao utilizador o comprimento e a altura de uma divisão de uma casa
  - Depois de questionar o comprimento e a largura, perguntar se existem mais divisões que pretenda adicionar
  - No fim, quando as divisões estiverem todas inseridas, apresentar o metros quadrados necessários

CICLO DO-WHILE

- Preencher com o código feito

CICLO FOR

- Quando utilizar:
  - Se sabemos o número extado de vezes que vamos repetir as instruções pretendida

- Por exemplo, usar um ciclo for para:
  - Imprimir uma contagem descrescente de apartir de 10
  - Encontrar o factorial de um número inserido por um utilizador

CICLO FOR

Sintaxe Java

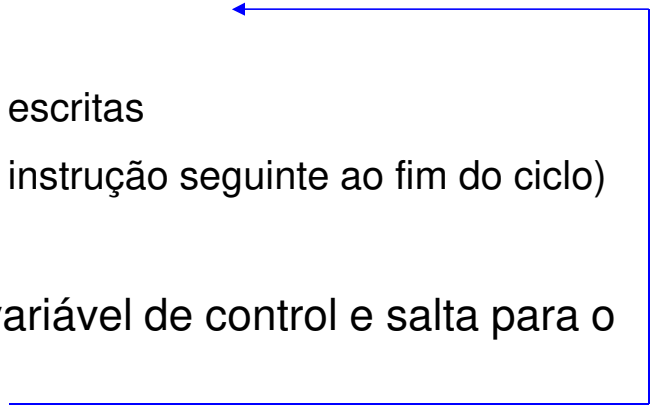
for (*<initialization>*; *<condition>*; *<update>*)
{
    <statement(s)>
}

Exemplo Java

```
for (int i=10; i>0; i--)
{
  System.out.print(i + " ");
}
System.out.println("Liftoff!");
```

CICLO FOR

- Semântica ciclo for:
  - Antes do início da primeira iteração, executar a inicialização
  - No início de cada iteração, avaliar a condição:
    - Se a condição for verdadeira, executar as instruções escritas
    - Se a condição for falsa, terminar o ciclo (salta para a instrução seguinte ao fim do ciclo)
  - No fim de cada iteração, executa a atualização da variável de control e salta para o início do ciclo




```
Scanner stdIn = new Scanner(System.in);
int number;
double factorial = 1.0;

System.out.print("Enter a whole number: ");
number = stdIn.nextInt();

for (int i=2; i<=number; i++) {
  factorial *= i;
}

System.out.println(number + "! = " + factorial);
```

CICLO FOR-DESAFIO

- Escrever um método main que imprima cada número impar entre 1 e 99
- <preencher com o código feito>

COMPARAÇÃO WHILE-FOR

```
int counter = 0;
while ( counter < 4 ) {
        System.out.println("    *");
        counter ++;
}
```

```
for (int counter = 0; counter < 4; counter++) {
  System.out.println("    *");
}
```

CICLOS ENCADEADOS

- Ciclos encadeados => um ciclo dentro de outro ciclo
- Desafio: Escrever um programa que imprime um rectângulo de caracteres onde o utilizador especifica a altura e o comprimento do rectângulo, e o caracter a apresentar
- Exemplo:

```
Enter height: 4
Enter width: 3
Enter character: <
<<<
<<<
<<<
<<<
```

CICLOS ENCADEADOS-FOR

```
int height = 4;
int width = 10;

for (int rowCount = 0; rowCount < height; rowCount++ ) {

   for (int colCount = 0; colCount < width; colCount++ ) {
       System.out.print("@");
   }
   System.out.println();
}
```

CICLOS ENCADEADOS-WHILE

```
String name = "Lenny";
  String guess = "";
  int numTries = 0;

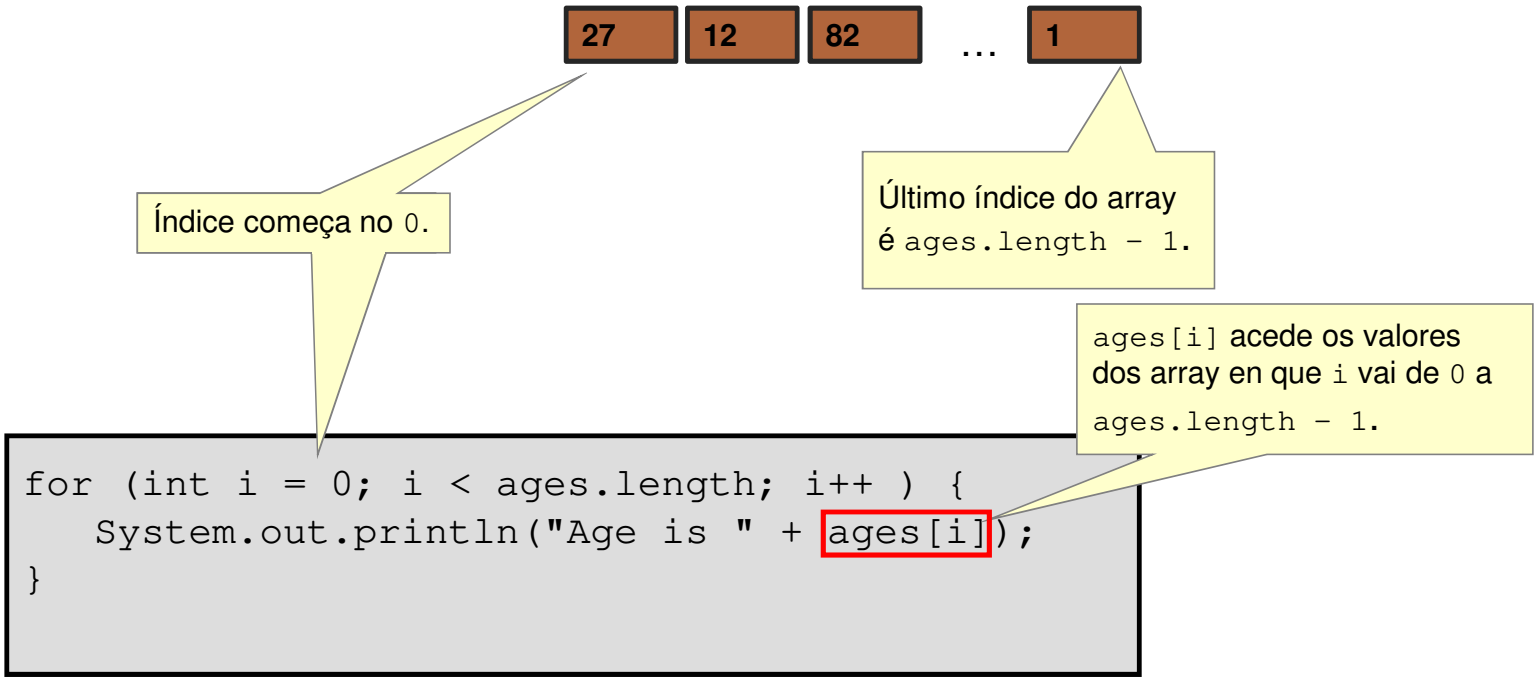
  while (!guess.equals(name.toLowerCase())) {
      guess = "";
      while (guess.length() < name.length()) {
         char asciiChar = (char)(Math.random() * 26 + 97);
         guess = guess + asciiChar;
      }
      numTries++;
  }
  System.out.println(name + " found after " + numTries + " tries!");
```

CICLOS E ARRAYS

- Um dos usos mais comuns para os ciclos é trabalhar com conjuntos de dados

- Todos os tipos de ciclo são úteis!!!

CICLOS E ARRAYS



DEFINIR VALORES NUM ARRAY

Ciclos acedem cada elemento de cada vez

```
for (int i = 0; int < ages.length; i++ ) {
    ages[i] = 10;
}
```

Cada elemento vai ter o seu valor definido para `10`.

"ENHANCED LOOP" COM ARRAYS

Ciclos acedem cada elemento de cada vez

Cada iteração devolve o elemento seguinte no array

```
for (int age : ages ) {
   System.out.println("Age is " + age);
}
```

FORMA BÁSICA DE UM MÉTODO

A palavara void indica que o método não devolve nenhum valor

Parêntesis vazios, indicam que nenhum argumento é passado para dentro do método

```
public   void   display ()   {

        System.out.println("Shirt description:" + description);

        System.out.println("Color Code: " + colorCode);

        System.out.println("Shirt price: " + price);

} // end of display method
```

PARÂMETROS EM MÉTODOS

```
public void calculate0(){
  System.out.println("No parameters");
}
```

```
public void calculate1(int x){
  System.out.println(x/2.0);
}
```

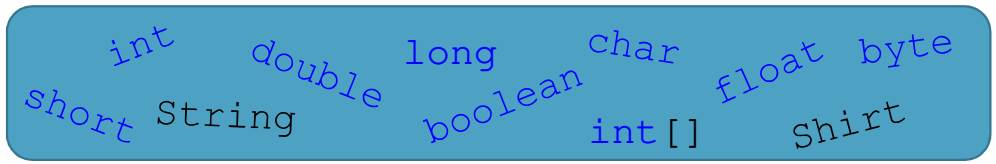
```
public void calculate2(int x, double y){
  System.out.println(x/y);
}
```

```
public void calculate3(int x, double y, int z){
  System.out.println(x/y +z);
}
```

TIPOS DE DADOS DEVOLVIDOS POR MÉTODOS

- As variáveis, como já sabemos, podem ser de diferentes tipos

- Um método pode também devolver diferentes tipos de dados:



TIPOS DE DADOS DEVOLVIDOS POR MÉTODOS

```
public void printString(){
  System.out.println("Hello");
}
```

```
public String returnString(){
  return("Hello");
}
```

```
public int sum(int x, int y){
  return(x + y);
}
```

```
public boolean isGreater(int x, int y){
  return(x > y);
}
```

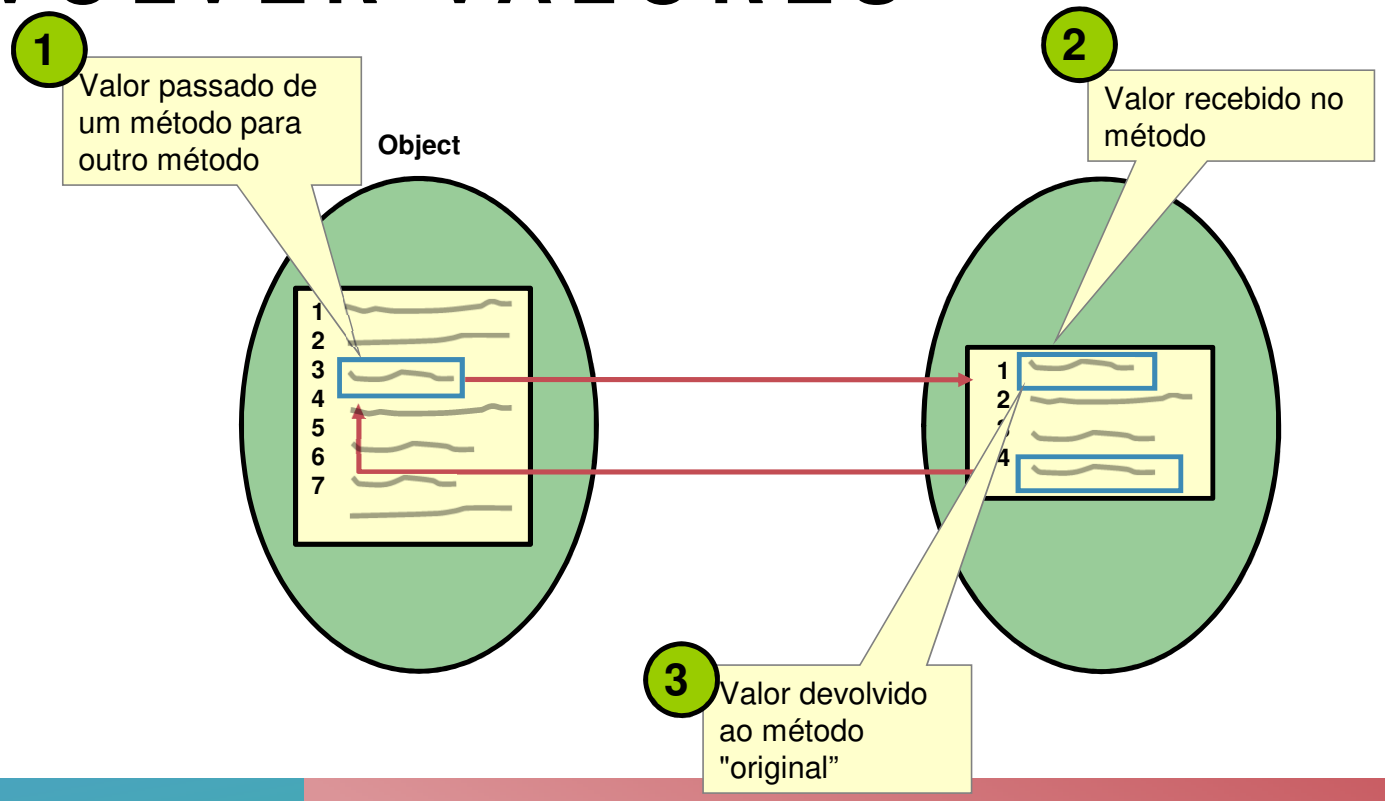
UM EXEMPLO...OU DOIS

```
public static void main(String[] args){
   int num1 = 1, num2 = 2;
   int result = num1 + num2;
   System.out.println(result);
}
```

```
public static void main(String[] args){
   int num1 = 1, num2 = 2;
   int result = sum(num1, num2);
   System.out.println(result);
}

public int sum(int x, int y){
   return(x + y);
}
```

PASSAR ARGUMENTOS E DEVOLVER VALORES



CÓDIGO SEM MÉTODOS

```
public static void main(String[] args){
      Shirt shirt01 = new Shirt();
      Shirt shirt02 = new Shirt();
      Shirt shirt03 = new Shirt();
      Shirt shirt04 = new Shirt();

      shirt01.description = "Sailor";
      shirt01.colorCode = 'B';
      shirt01.price = 30;

      shirt02.description = "Sweatshirt";
      shirt02.colorCode = 'G';
      shirt02.price = 25;

      shirt03.description = "Skull Tee";
      shirt03.colorCode = 'B';
      shirt03.price = 15;

      shirt04.description = "Tropical";
      shirt04.colorCode = 'R';
      shirt04.price = 20;
}
```

MELHOR CÓDIGO COM MÉTODOS

```
public static void main(String[] args){
        Shirt shirt01 = new Shirt();
        Shirt shirt02 = new Shirt();
        Shirt shirt03 = new Shirt();
        Shirt shirt04 = new Shirt();

        shirt01.setFields("Sailor", 'B', 30);
        shirt02.setFields("Sweatshirt", 'G', 25);
        shirt03.setFields("Skull Tee", 'B', 15);
        shirt04.setFields("Tropical", 'R', 20);
}
```

```
public class Shirt {
        public String description;
        public char colorCode;
        public double price;

        public void setFields(String desc, char color, double price){
                this.description = desc;
                this.colorCode = color;
                this.price = price;
        }
…
```

CÓDIGO AINDA MELHOR USANDO MÉTODOS

```
public static void main(String[] args){
        Shirt shirt01 = new Shirt("Sailor", "Blue", 30);
        Shirt shirt02 = new Shirt("SweatShirt", "Green", 25);
        Shirt shirt03 = new Shirt("Skull Tee", "Blue", 15);
        Shirt shirt04 = new Shirt("Tropical", "Red", 20);
}
```

```
public class Shirt {
        public String description;
        public char colorCode;
        public double price;

        //Constructor
        public Shirt(String desc, String color, double price){
                setFields(desc, price);
                setColor(color);
        }
        public void setColor (String theColor){
                if (theColor.length() > 0)
                        colorCode = theColor.charAt(0);
                }
        }
}
```

VANTAGENS DO USO DE MÉTODOS

- São reutilizáveis
- Fazem os programas mais pequenos e de mais fácil leitura
- Fazem o desenvolvimento e a manuntenção mais rápida e fácil
- Permitem objetos comunicar e distribuir o trabalho gerado num dado programa

MÉTODOS SYSTEM.OUT

- Do método System.out.println() considerar o seguinte:
  - **System** é uma classe (em Java.lang)
  - **out** é um campo da classe System
  - **out** é uma referência que permite invocar o método **println()** no objeto que o referencia

UTILIZANDO OS MÉTODOS PRINT() E PRINTLN()

- Método println:

```
System.out.println(<dados a imprimir>);
```

- Exemplo:

```
System.out.print("Carpe diem ");
System.out.println("Seize the day");
```

PROGRAMAÇÃO ORIENTADA EM OBJETOS

PARTE II

PROGRAMAÇÃO ORIENTADA A OBJETOS

- Paradigma da Programação Orientada a Objectos (POO)
- Classes e Objetos
- Propriedades e Métodos
- Numbers, Strings, Collections
- Polimorfismo

PROGRAMAÇÃO ORIENTADA A OBJETOS

- No início da programação a técnica de programação mais utilizada era a chamada "programação procedimental"
- Esta focava-se nos procedimentos ou tarefas que um programa tinha de realizar
- O programador desenhava o programa à volta do que julgava ser os procedimentos mais importantes
- Hoje em dia o paradigma mudou, e a técnica mais utilizada é a "programação orientada a objetos"
- Neste paradigma, em vez do foco ser os procedimentos mais importantes, são os elementos que constituem o programa em si – a esses elementos designamos de **objetos**

PROGRAMAÇÃO ORIENTADA A OBJETOS - OBJETOS

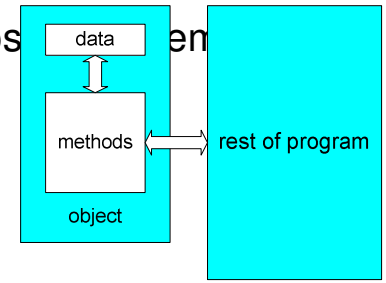
- Um objeto é:
  - Um conjunto de dados relacionados que identificam o estado atual do objeto em si
  - Um conjunto de comportamentos/ações

- Exemplos de objetos:

Entidades humanas	Objetos físicos	Entidades matemáticas
Empregados	Carros	Pontos em sistema de coordenadas
Clientes	Aviões	Números Complexos
Estudantes	Motas	Tempo

PROGRAMAÇÃO ORIENTADA A OBJETOS

- Benefícios da POO:
  - Programas mais fáceis de entender:
    - Visto que as pessoas tendem no mundo real a pensar e interagir com objetos, é mais fácil para as pessoas entenderem um programa desenvolvido com esse paradigma de programação
  - Menos erros:
    - Visto que os objetos neste paradigma de programação utilizam o conceito de **encapsulamento** (isolamento) para os dados, é mais difícil os dados [illegible] em

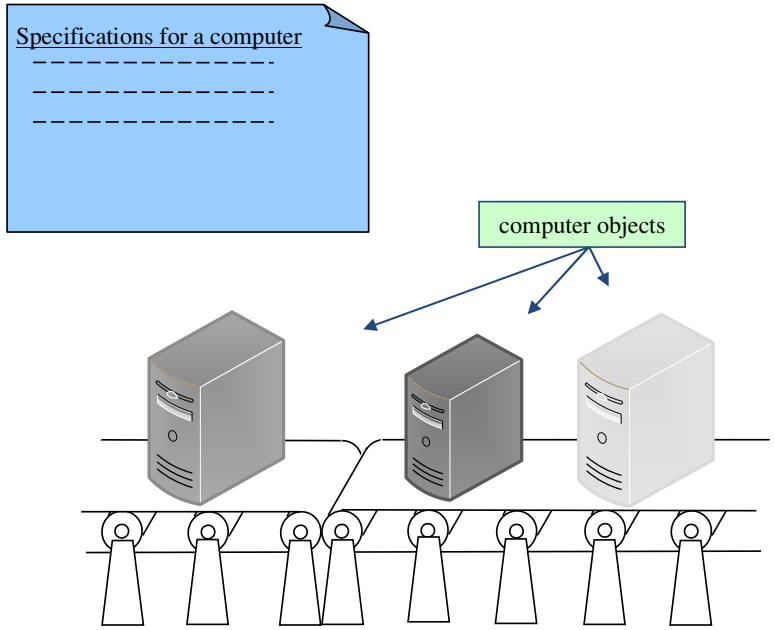


PROGRAMAÇÃO ORIENTADA A OBJETOS - CLASSES

- Uma **classe** é uma descrição de um conjunto de objetos
- No próximo slide veremos 3 computadores numa linha de montagem:
  - Os 3 computadores representam **objetos**, e o documento de especificação representa a **classe**
  - O documento de especificação é como se fosse uma planta que descreve os computadores: a sua lista de componentes, as suas características e instruções de montagem

- Pense num objeto com o um exemplo físico da descrição de uma classe.
- Formalmente dizemos que **um objeto é uma instância de uma classe**

PROGRAMAÇÃO ORIENTADA A OBJETOS - CLASSES



PROGRAMAÇÃO ORIENTADA A OBJETOS - CLASSES

Uma classe é uma descrição para um conjunto de objetos

A descrição consiste em:

- Um conjunto de variáveis
- Um conjunto de métodos

PROGRAMAÇÃO ORIENTADA A OBJETOS - CLASSES

- As classes definem 2 tipos de variáveis:
  - Variáveis de instância
  - Variáveis de classe

- As classes definem 2 tipos de métodos:
  - Métodos de instância
  - Métodos de classe

- Variáveis e métodos de instância são mais comuns do que as variáveis e métodos de classe

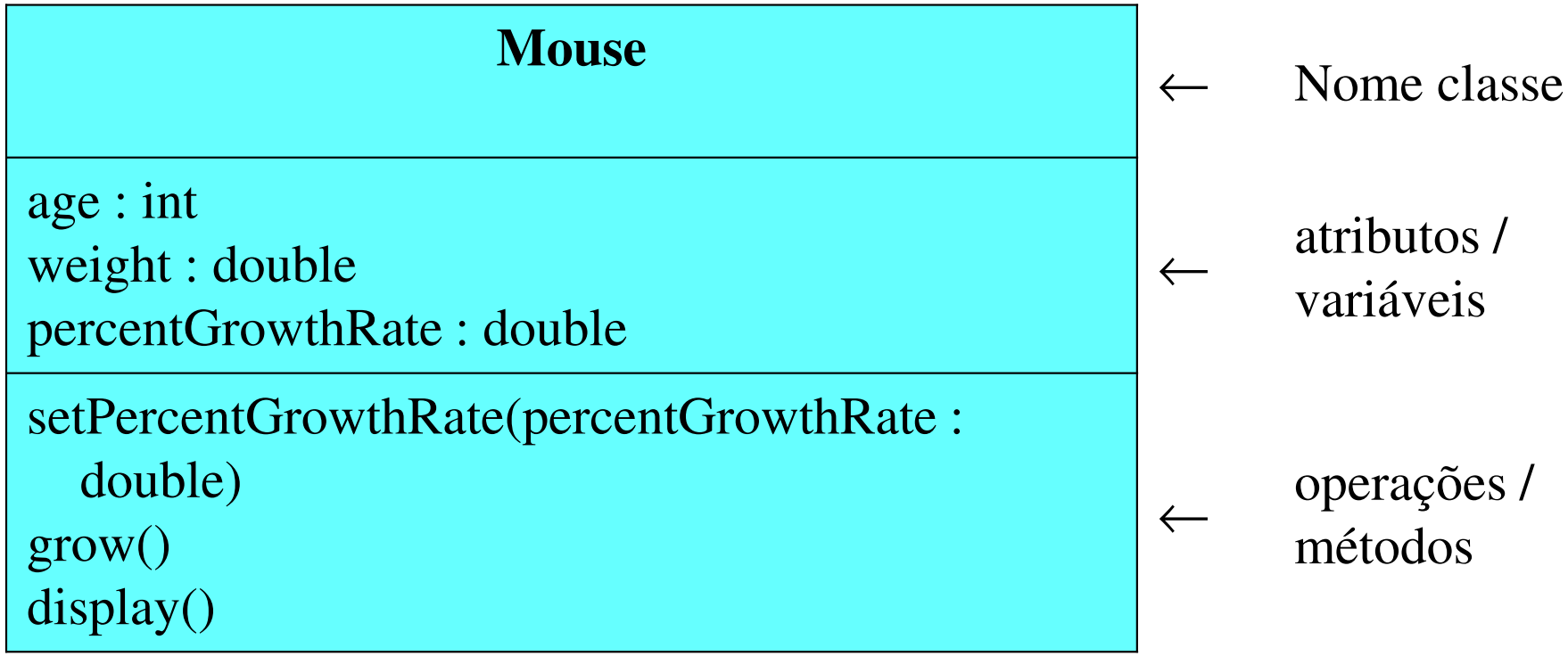
- Para já focar-nos-emos no primeiro tipo…

PROGRAMAÇÃO ORIENTADA A OBJETOS - CLASSES

As variáveis de instância de uma classe especificam os tipos de dados que um objeto pode ter

Os métodos de instância de uma classe especifica os comportamentos que um objeto pode ter

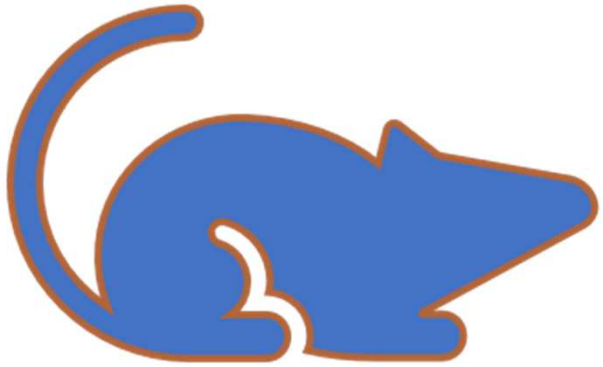
DIAGRAMA DE CLASSES-UML



A NOSSA PRIMEIRA CLASSE

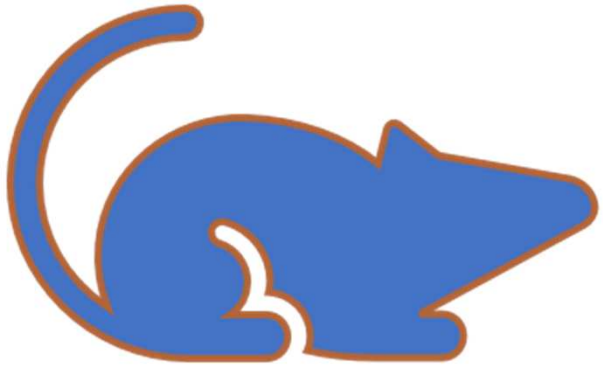
```
public class Mouse
{
  private int age = 0;               // idade em dias
  private double weight = 1.0;       // peso em gramas
  private double percentGrowthRate; // % aumento peso por dia

  // Este método define a percentage de crescimento
  public void setPercentGrowthRate(double percentGrowthRate)
  {
    this.percentGrowthRate = percentGrowthRate;
  }
```



A NOSSA PRIMEIRA CLASSE

```
 // Este método simula um dia de crescimento
  public void grow()
  {
    this.weight += (.01 * this.percentGrowthRate * this.weight);
    this.age++;
  }
  // Este método imprime a idade e o peso
  public void display()
  {
    System.out.printf(
      "Age = %d, weight = %.3f\n", this.age, this.weight);
  }
}
```



MODIFICADORES DE ACESSO

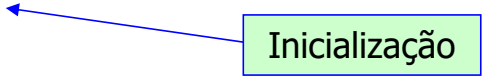
- **private** e **public** são **modificadores de acesso**
- Isto determina a facilidade com que um membro de uma classe/objeto pode ser acedido
- Se declarar um membro como privado, então o membro apenas pode ser acedido dentro da própria classe
- Se declarar um membro com público, então o membro pode ser acedido através de qualquer ponto da aplicação. Os métodos por norma são declarados como públicos pois normalmente quer invocá-los quando e onde forem necessários

VARIÁVEIS DE REFERÊNCIA E INSTANCIAÇÃO

- Para declarar uma variável do tipo referência (que contém o endereço de memória onde o objeto está guardado): **<nome classe> <variável referência>;**

- Para instanciar/criar um objeto e designar-lhe o seu endereço numa variável de referência: **<variável referência> = new <nome classe> ();**

- Exemplo:
  Mouse gus = new Mouse();



INVOCAR UM MÉTODO

- Depois de instanciar um objeto e designar-lhe o seu endereço para uma variável de referência, pode-se invocar um método usando a seguinte sintaxe:

<variável referência>.<nome método>(<lista argumentos separados por vírgula>);

- Exemplo:
  gus.display();

INVOCAR UM OBJETO

```
public static void main(String[] args)
{
  Scanner stdIn = new Scanner(System.in);
  double growthRate;
  Mouse gus = new Mouse();

  System.out.print("Enter growth rate as a percentage: ");
  growthRate = stdIn.nextDouble();
  gus.setPercentGrowthRate(growthRate);
  gus.grow();
  gus.display();
}
```

A REFERÊNCIA THIS

- Esta referência é utilizada dentro de um método para se referir ao objeto que invocou o método

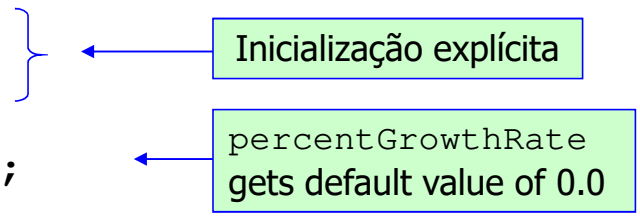
- Por outras palavras: a referência **this** refere-se ao objeto invocador

VALORES POR OMISSÃO

- O valor por omissão de uma variável é o valor que a variável tem quando não existe uma inicialização explícita

- Na classe Mouse vista anteriormente, as declarações das variáveis de instância:

```
private int age = 0;
private double weight = 1.0;
private double percentGrowthRate;
```



VALORES POR OMISSÃO

- Os valores por omissão para as variáveis de instância:
  - Numéricos inteiros - 0
  - Númericos vírgula flutuante – 0.0
  - Booleanos – false
  - Tipos de referência – null

VARIÁVEIS LOCAIS

- Uma variável local é uma variável que é declarada dentro de um método
- São diferentes de variáveis de instância, que são no topo de uma classe, fora dos métodos
- Uma variável local tem esse nome pois ela apenas pode ser usada dentro do método em que foi declarada

VARIÁVEIS LOCAIS

```
import java.util.Scanner;


public class Mouse2Driver
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    Mouse2 mickey = new Mouse2();
    int days;


    mickey.setPercentGrowthRate(10);
    System.out.print("Enter number of days to grow: ");
    days = stdIn.nextInt();
    mickey.grow(days);
    System.out.printf("Age = %d, weight = %.3f\n",
      mickey.getAge(), mickey.getWeight());
  } // end main
} // end class Mouse2Driver
```

Variáveis locais

CLASSE MOUSE2

```
import java.util.Scanner;


public class Mouse2
{
  private int age = 0;                // age in days
  private double weight = 1.0;        // weight in grams
  private double percentGrowthRate;  // % daily weight gain

  //*******************************************************

  public void setPercentGrowthRate(double percentGrowthRate)
  {
    this.percentGrowthRate = percentGrowthRate;
  } // end setPercentGrowthRate

  //*******************************************************

  public int getAge()
  {
    return this.age;
  } // end getAge
```

CLASSE MOUSE2

```
//*******************************************************

 public double getWeight()
 {
   return this.weight;
 } // end getWeight

 //*******************************************************

 public void grow(int days)
 {
   for (int i=0; i<days; i++)
   {
     this.weight +=
        (.01 * this.percentGrowthRate * this.weight);
   }
   this.age += days;
 } // end grow
} // end class Mouse2
```

Variável local

INSTRUÇÃO RETURN

- A instrução return permite enviar o valor de dentro de um método, de volta para o sítio onde o método foi invocado
- Da classe Mouse2:

```
public int getAge()
{
  return this.age;
} // end getAge
```

Tipo de retorno

Instrução return

- Da classe Mouse2Driver:

```
System.out.printf("Age = %d, weight = %.3f\n",
  mickey.getAge(), mickey.getWeight());
```

Invocação dos métodos

TIPO RETORNO VOID

- Como se verá no método em baixo, se um método não devolver um valor, então deve ser especificado **void** para o seu tipo de retorno

```
public void grow(int days)
{
  for (int i=0; i<days; i++)
  {
    this.weight += (0.01 * this.percentGrowthRate *
 this.weight);
  }
  this.age += days;
} // end grow
```

INSTRUÇÃO RETURN VAZIA

- Para métodos com o tipo de retorno void, é válido ter uma instrução **return** vazia
- A instrução return vazia faz o que se espera:
  - Termina o método atual, e o programa vai para o local onde o método foi invocado

```
public void grow(int days)
{
  int endAge;
  endAge = this.age + days;
  while (this.age < endAge)
  {
    if (this.age >= 100)
    {
      return;
    }
    this.weight += .01 * this.percentGrowthRate *
   this.weight;
    this.age++;
  }  // end while
} // end grow
```

Instrução return vazia

VALORES POR OMISSÃO VARIÁVEIS LOCAIS

- O valor por omissão de um variável local é lixo
- Lixo significa que o valor de uma variável é desconhecido
- Se um programa tenta aceder a uma variável que contém lixo, o compilador gera um erro:

```
public void grow(int days)
{
  for (int i; i<days; i++)
  {
    this.weight +=
      (.01 * this.percentGrowthRate *
    this.weight);
  }
  this.age += days;
} // end grow
```

PERSISTÊNCIA DE VARIÁVEIS LOCAIS

- Variáveis locais persistem apenas durante a duração de um método ou de um ciclo, no qual a variável foi definida

- Na vez seguinte em que o método, ou ciclo, for executado a variável local é redefinida com o seu valor inicial

PASSAGEM DE ARGUMENTOS

- Qual é o output do seguinte código?

```
public class Mouse3Driver
{
  public static void main(String[] args)
  {
    Mouse3 minnie = new Mouse3();
    int days = 365;
    minnie.grow(days);
    System.out.println("# of days aged =
  " + days);
  } // end main
} // end class Mouse3Driver
```

PASSAGEM DE ARGUMENTOS

```
public class Mouse3
{
  private int age = 0;                    // age in days
  private double weight = 1.0;            // weight in grams
  private double percentGrowthRate = 10; // % daily weight gain

  public void grow(int days)
  {
    this.age += days;
    while (days > 0)
    {
      this.weight +=
        (.01 * this.percentGrowthRate * this.weight);
      days--;
    }
  } // end grow
} // end class Mouse3
```

PASSAGEM DE ARGUMENTOS

- O Java usa o mecanismo **pass-by-value** para passar argumentos para os métodos

- Pass-by-value significa que a JVM passa uma **cópia** do valor do argumento (não o argumento em si) para o parâmetro

- Se o valor do parâmetro mudar dentro do método, o argumento no método invocador não é afetado

MÉTODOS ESPECIALIZADOS

Métodos assessores:

- Permitem aceder ao valor de uma variável de instância

```
public int getAge()
{
  return this.age;
}
```

Métodos mutadores:

- Permitem alterar o valor de uma variável de instância

```
public void
  setPercentGrowthRate(double
  percentGrowthRate)
{
  this.percentGrowthRate =
  percentGrowthRate;
}
```

MÉTODOS ESPECIALIZADOS

- Métodos booleanos:
  - Permitem verificar a verdade ou falsidade de uma dada condição
  - Devolvem sempre o valor booleano
  - Normalmente começam com a palavra **is**

MÉTODOS ESPECIALIZADOS-BOOLEANOS

```
public boolean isAdolescent()
{
  if (this.age <= 100)
  {
    return true;
  }
  else
  {
    return false;
  }
} // end isAdolescent
```

```
Mouse pinky = new Mouse();
...
if (pinky.isAdolescent() ==
  false)
{
  System.out.println(
    "The Mouse's growth is no
  longer" +
    " being simulated - too
  old.");
```

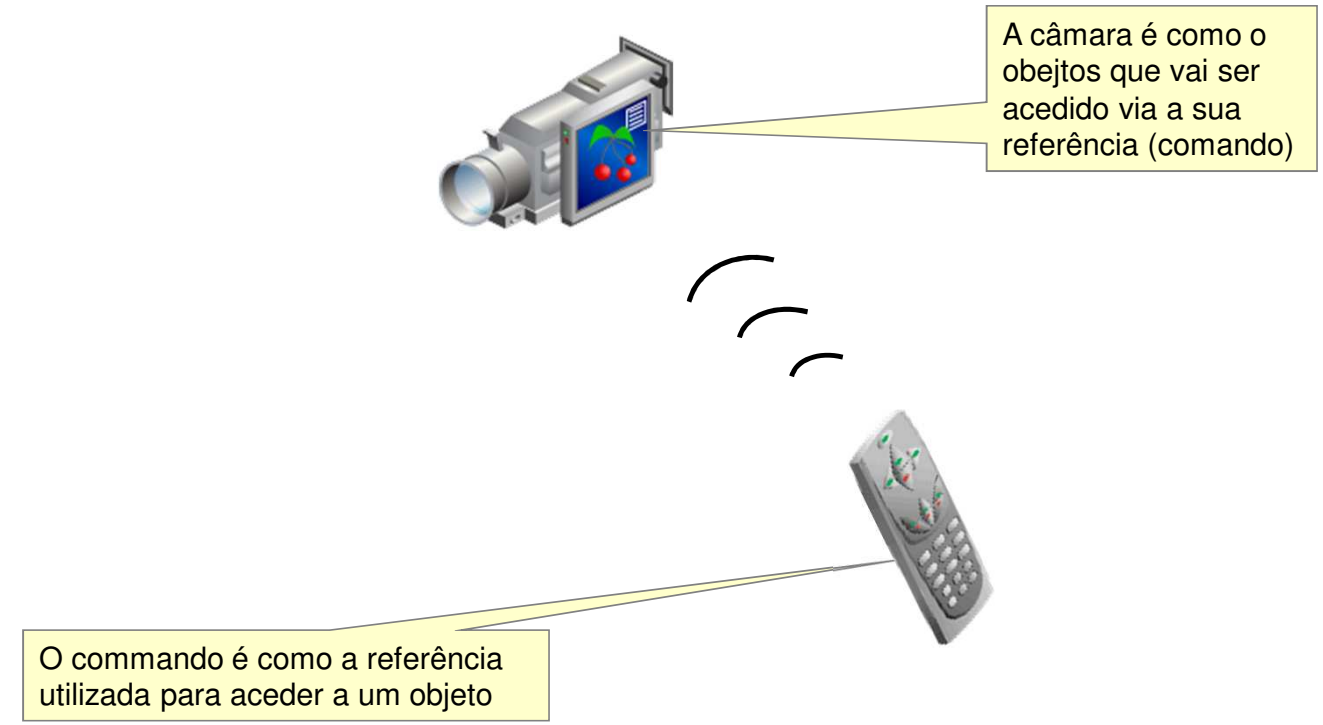
TRABALHAR COM OBJETOS

PARTE III

TRABALHAR COM OBJETOS: INTRODUÇÃO

- Objetos são acedidos através de referências
- Objetos são versões instanciadas das suas classes
- Objetos consistem num conjunto de atributos e operações
- Em Java atributos e operações designam-se de campos e métodos

ACEDER A OBJETOS USANDO UMA REFERÊNCIA




```
public class Shirt {
  public int shirtID = 0;
  public String description =
        "-description required-";// The color
   codes are R=Red, B=Blue, G=Green, U=Unset
  public char colorCode = 'U';
  public double price = 0.0;

  public void display() {
    System.out.println("Item ID: " +
   shirtID);
    System.out.println("Item description:" +
   description);
    System.out.println("Color Code: " +
   colorCode);
    System.out.println("Item price: " +
   price);
  }
}
```

TRABALHAR COM VARIÁVEIS DE REFERÊNCIA

Declaração:

```
NomeClasse identificador;
```

Instanciação:

```
new NomeClasse();
```

Esta linha de código é a que cria o objeto.

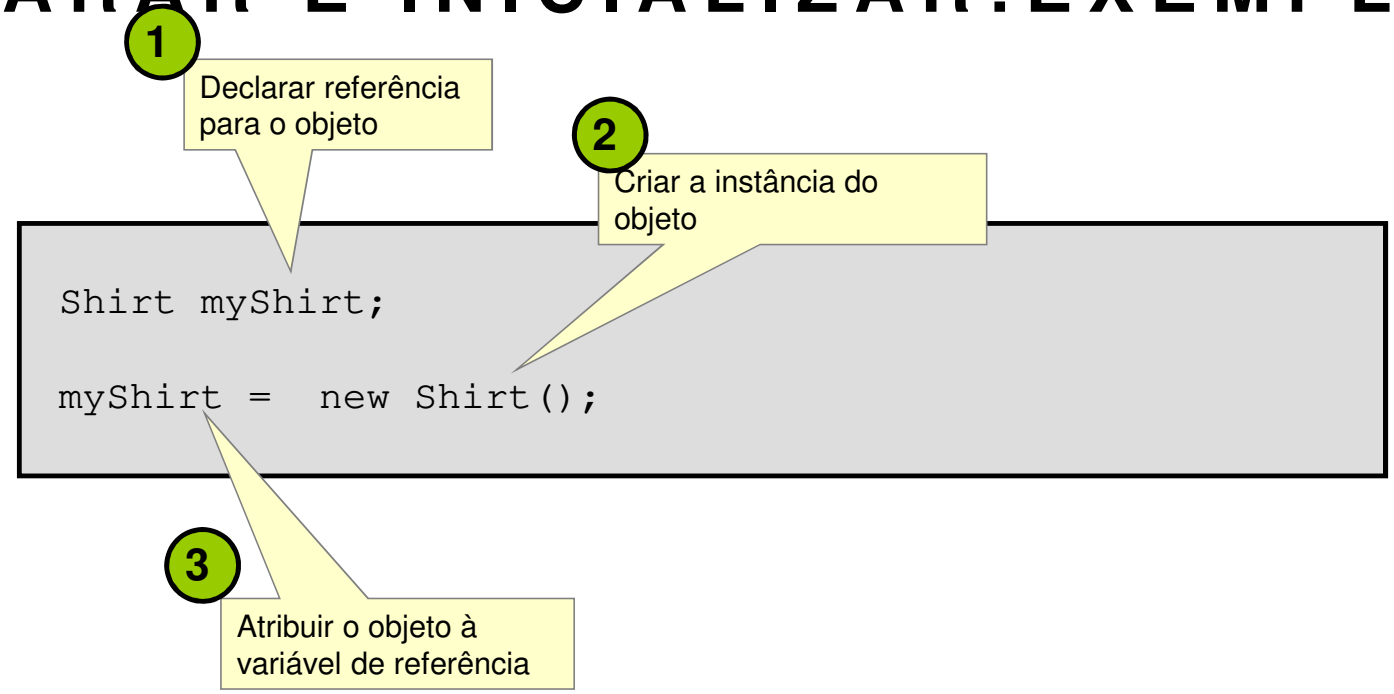
Atribuição:

```
Objeto referência = new NomeClasse();
```

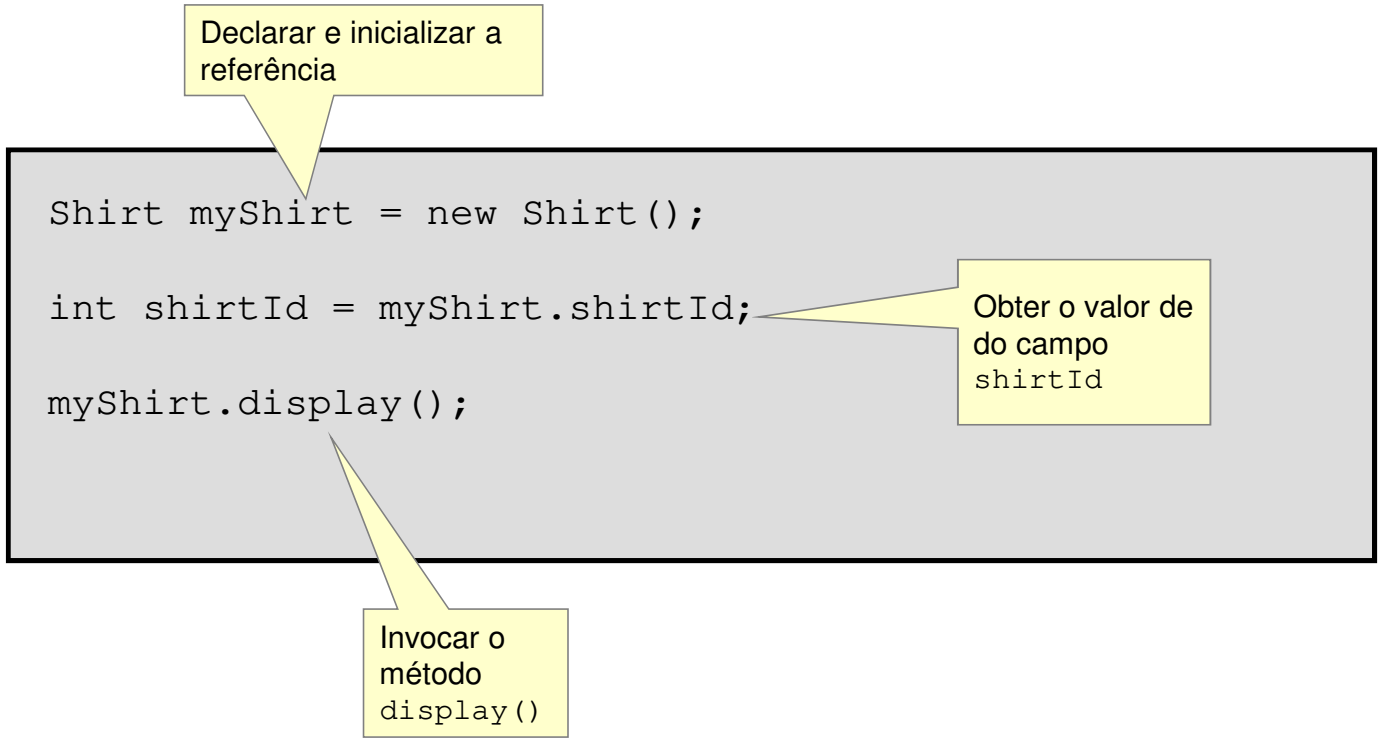
O identificador da declaração

O operador de atribuição

DECLARAR E INICIALIZAR:EXEMPLO



REFERÊNCIAS DE OBJETOS



1 Pegar no comando e aceder à câmara

2 Pressionar os botões do comando para efetuarem ações

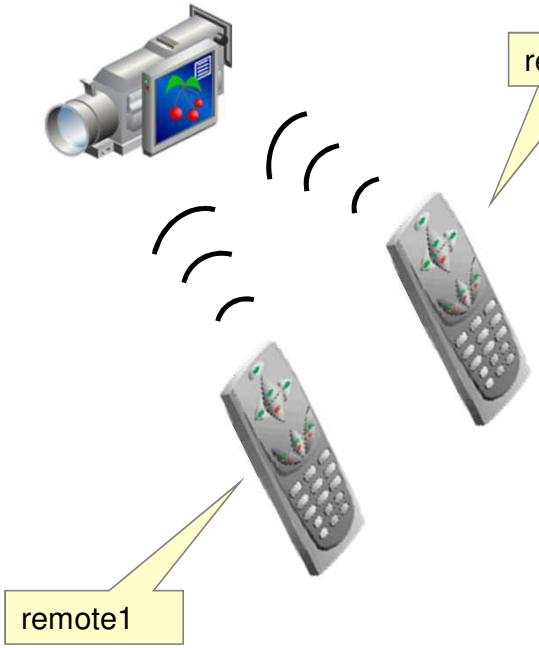
1 Criar o objeto Shirt e obter uma referêcnia para ele

```
Shirt myShirt = new Shirt();

myShirt.display();
```

2 Invocar um método para o objeto Shirt fazer algo

REFERÊNCIAS DE OBJETOS



Só existe um objeto Camera

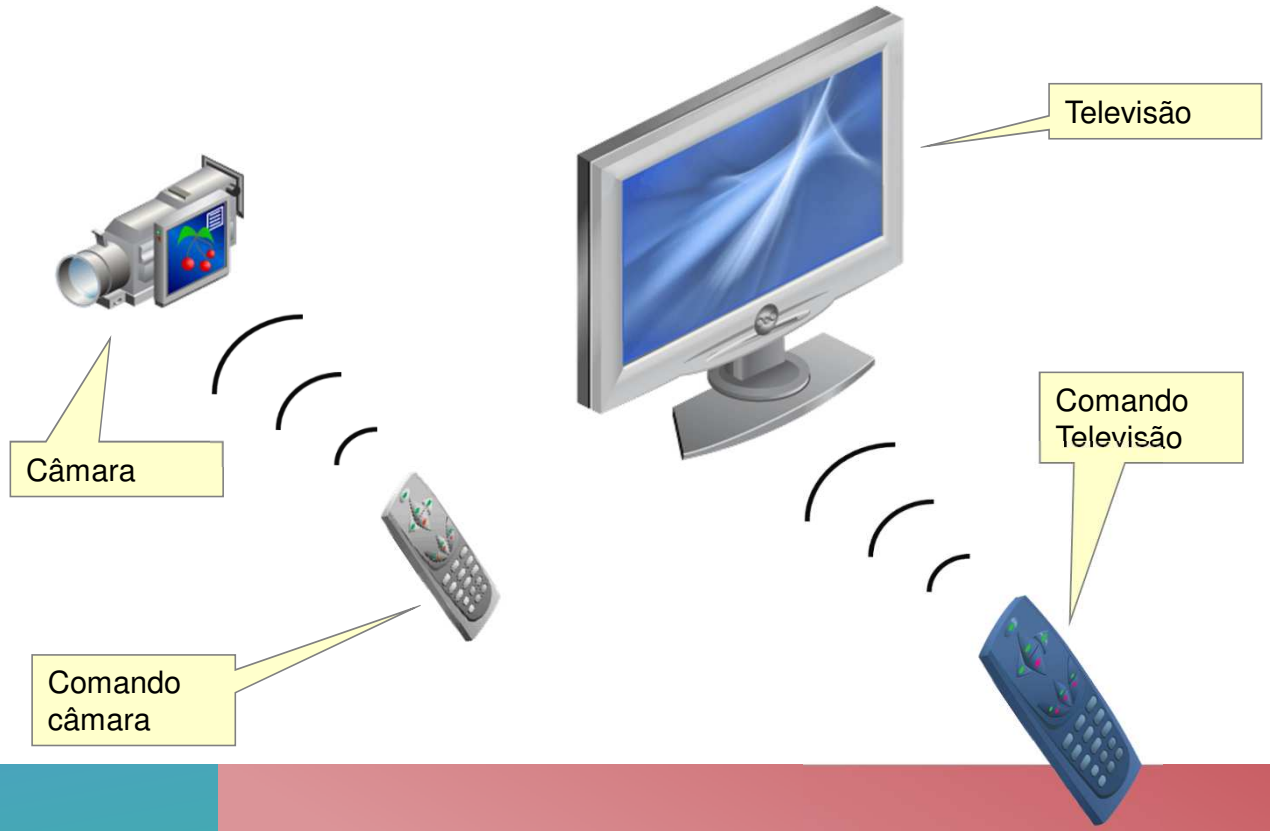
```
Camera remote1 = new Camera();

Camera remote2 = remote1;

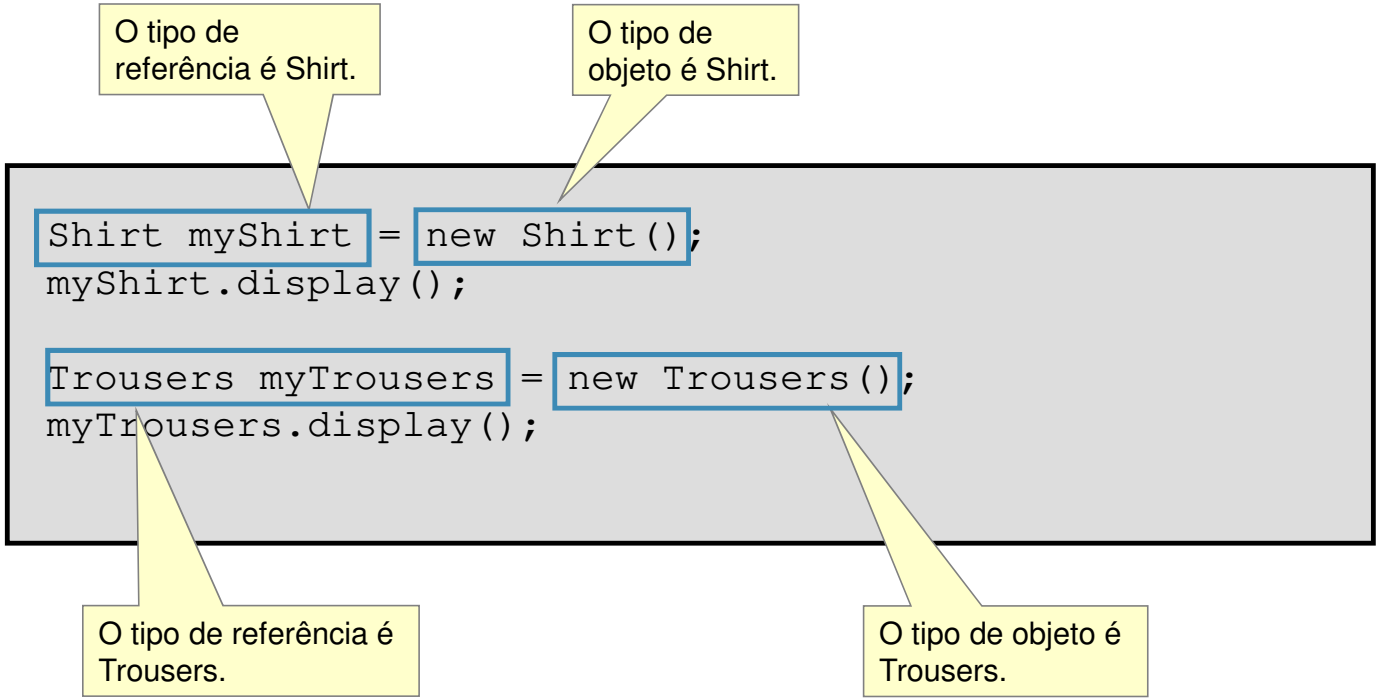
remote1.play();

remote2.stop();
```

REFERÊNCIA PARA DIFERENTES OBJETOS

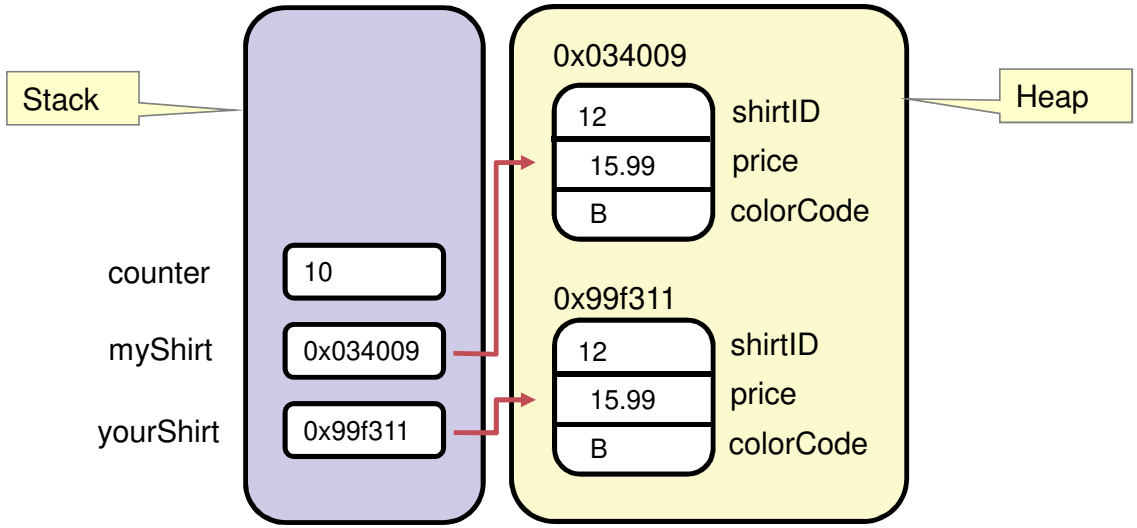


REFERÊNCIA PARA DIFERENTES OBJETOS



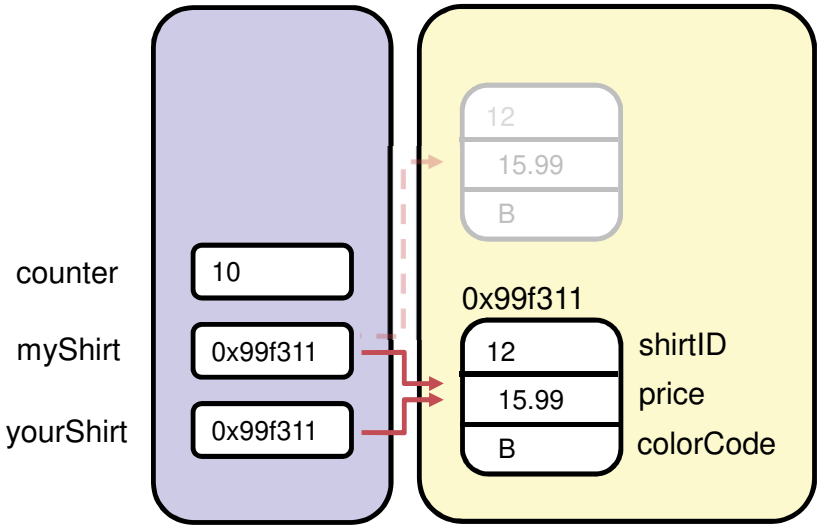
REFERÊNCIAS E OBJETOS EM MEMÓRIA

```
int counter = 10;
Shirt myShirt = new Shirt();
Shirt yourShirt = new Shirt();
```



ATRIBUIR UMA REFERÊNCIA A OUTRA REFERÊNCIA

```
myShirt = yourShirt;
```



DUAS REFERÊNCIAS-UM OBJETO

Código:

```
Shirt myShirt = new Shirt();
Shirt yourShirt = new Shirt();

myShirt = yourShirt;

myShirt.colorCode = 'R';
yourShirt.colorCode = 'G';

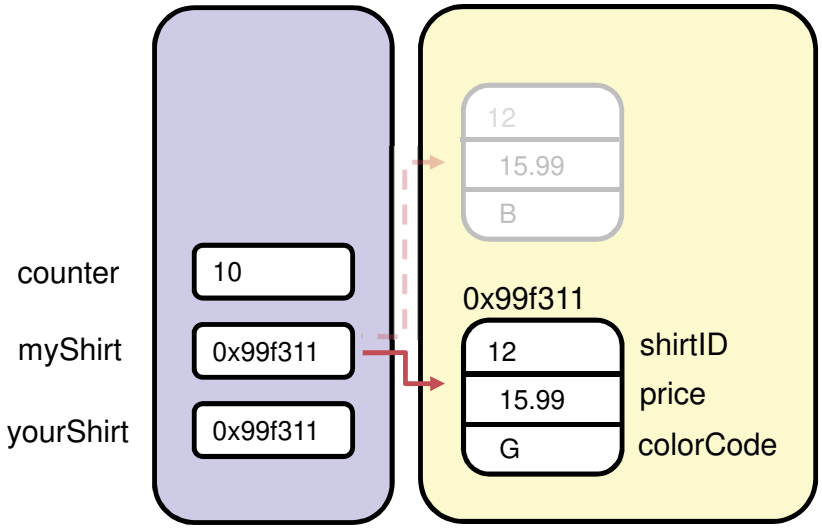
System.out.println("Shirt color: " + myShirt.colorCode);
```

Output do código:

```
 Shirt color: G
```

ATRIBUIR UMA REFERÊNCIA A OUTRA REFERÊNCIA

```
myShirt.colorCode = 'R';
yourShirt.colorCode = 'G';
```



A CLASSE STRING

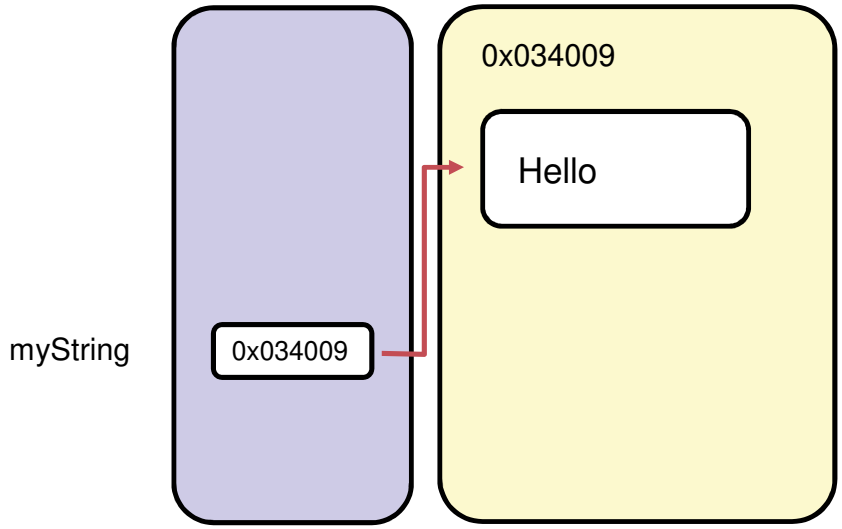
- A classe String suporta uma sintaxe diferente da sintaxe standard
- Pode ser instanciada sem usar a **palavra reservada new**
  String hisName = “António Sousa”;
- A palavra reservada **new** pode ser usada, **mas não é uma boa prática**
  String herName = new String(“Maria Manuela”);
- **O objeto String é imutável**…ou seja, o seu valor não pode ser mudado
- Um objeto String **pode ser usado com o sinal +**, para concatenação

CONCATENAÇÃO DE STRINGS

- Quando se utiliza um literal do tipo String em Java, ele é instanciado e obtém-se um referência String
- Concatenar Strings:
  - ```
    String name1 = "Fred";
    theirNames = name1 + " and " + "Anne Smith";
    ```
- A concatenação cria uma nova String, e a referência `theirNames` agora aponta para uma nova String
    ```

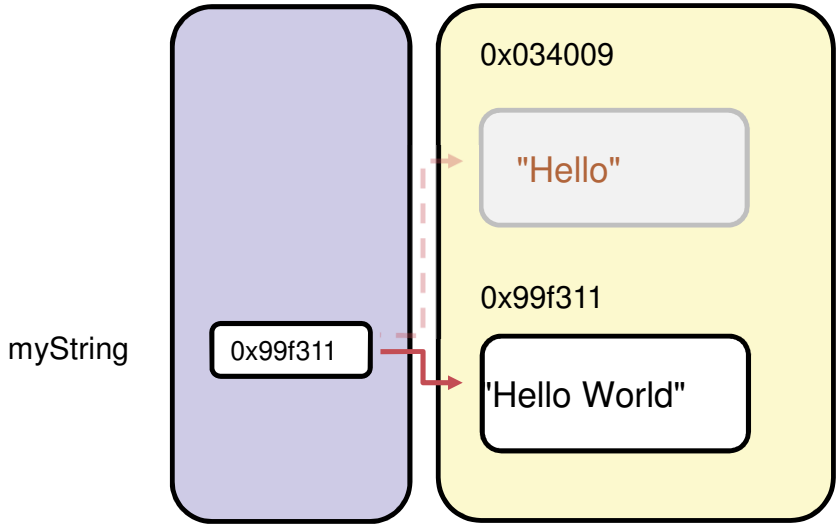
CONCATENAÇÃO DE STRINGS

```
String myString = "Hello";
```



CONCATENAÇÃO DE STRINGS

```
String myString = "Hello";
myString = myString.concat(" World");
```



MÉTODOS DA CLASSE STRING COM DIFERENTES TIPOS DE RETORNO

- Um método pode devolver um valor único de qualquer tipo

- Exemplo:

```
String hello = "Hello World";
int stringLength = hello.length();


String greet = " HOW ".trim();
String lc = greet + "DY".toLowerCase();
```

MÉTODOS DA CLASSE STRING COM ARGUMENTOS

- Os métodos, como falado anteriormente, podem necessitar de nenhum, um, ou mais do que um argumento
- Passando um argumento do tipo primitivo:

```
String theString = "Hello World";
String partString = theString.substring(6);
```

- Passando um argumento do tipo objeto:
  - `boolean endWorld = "Hello World".endsWith("World");`

CLASSE STRINGBUILDER

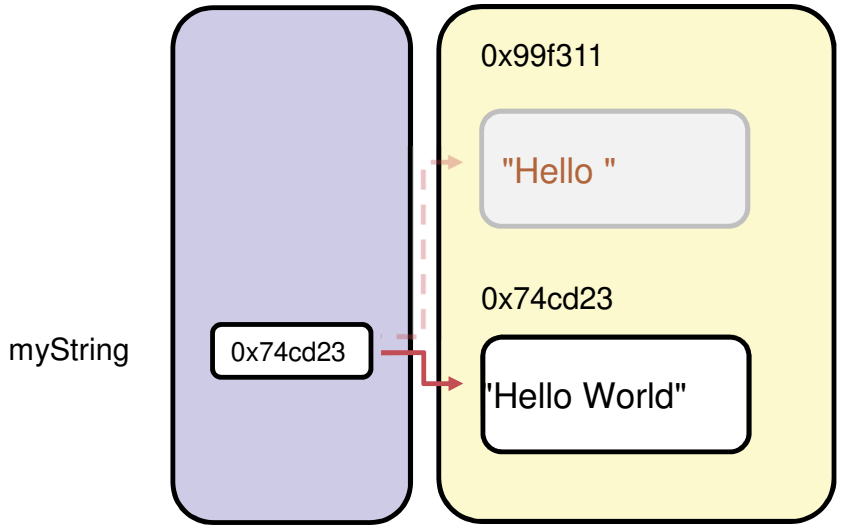
- A classe StringBuilder é uma alternativa mutável à classe String
- StringBuilder:
  - É uma classe normal. Utiliza a palavra new para ser instanciada
  - Tem uma extensiva panóplia de métodos para acrescentar, inserir ou apagar
  - Tem muitos métodos para devolver referências do próprio objeto, não havendo "custo" de instanciação
  - Pode ser iniciada com uma capacidade inicial, ficando com o tamanho exato para as necessidades

MAS AFINAL E A CLASSE STRING?

- Continua a ser necessária!!
- Normalmente é mais seguro usar um objeto imutável
- Por norma várias classes em Java requerem uma String em vez de StringBuilder
- Possui mais métodos do que a classe StringBuilder

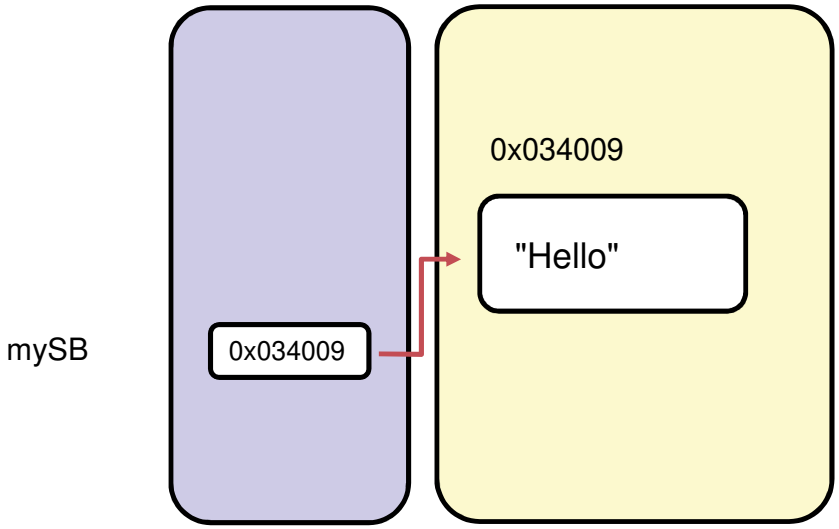
STRINGBUILDER-VANTAGENS SOBRE A CONCATENAÇÃO DE STRINGS

```
String myString = "Hello";
myString = myString.concat(" World);
```



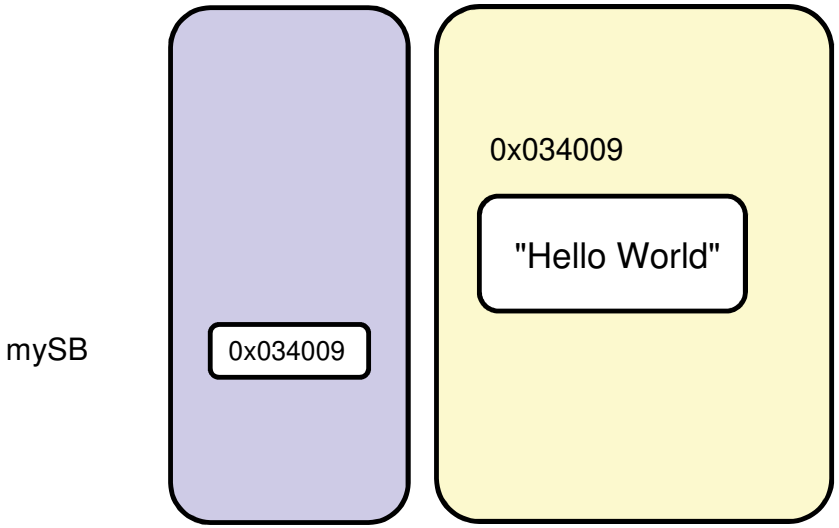
STRINGBUILDER-DECLARAR E INSTANCIAR

```
StringBuilder mySB = new StringBuilder("Hello");
```



STRINGBUILDER-ACRESCENTAR

```
StringBuilder mySB = new StringBuilder("Hello");
mySB.append(" World");
```



RESUMO

- Objetos são acedidos através de referências
- Objetos são versões instanciadas das suas classes
- Objetos consistem em atributos e operações:
  Em java, designamos de atributos e métodos
- Para aceder aos atributos de métodos de um objeto, obter a variável referência do objeto
- Uma referência já existente de um objeto, pode ser redifinida para uma nova variável de referência
- A palavra reservada **new** instancia um novo objeto e devolve uma referência

CRIAR E USAR MÉTODOS

PARTE IV

FORMA BÁSICA DE UM MÉTODO

A palavra reservada void keyword indica que o método não devolve nenhum valor

Parêntesis vazio indicam que nenhum argumento é passado para o método

```
public  void  display ()  {
        System.out.println("Shirt description:" + description);
        System.out.println("Color Code: " + colorCode);
        System.out.println("Shirt price: " + price);
}
```

INVOCAR UM MÉTODO ATRAVÉS DE OUTRA CLASSE

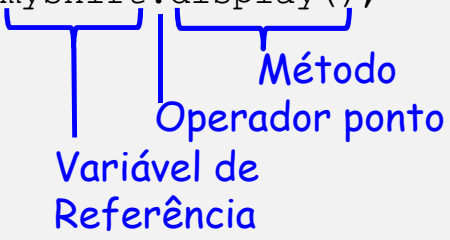
```
public class ShoppingCart {

        public static void main (String[] args) {

                Shirt myShirt = new Shirt();

                myShirt.display();

        }
}
```



ARGUMENTOS E PARÂMETROS DE MÉTODOS

- Um **argumento** é um valor que é passado durante a invocação de um método:

```
Calculator calc = new Calculator();
double denominator = 2.0
                     Argumentos
calc.calculate(3, denominator);
```

- Um **parâmetro** é uma variável definida na declaração de um método:

```
                                3          2.0
public void calculate(int x, double y){
  System.out.println(x/y);
}
                     Parâmetros
```

- Métodos podem ter qualquer número ou tipo de parâmetros:

```
public void calculate0(){
  System.out.println("No parameters");
}
```

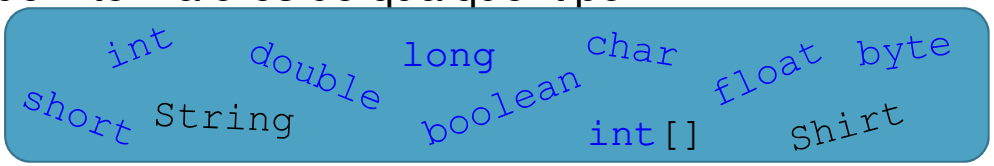
```
public void calculate1(int x){
  System.out.println(x/2.0);
}
```

```
public void calculate2(int x, double y){
  System.out.println(x/y);
}
```

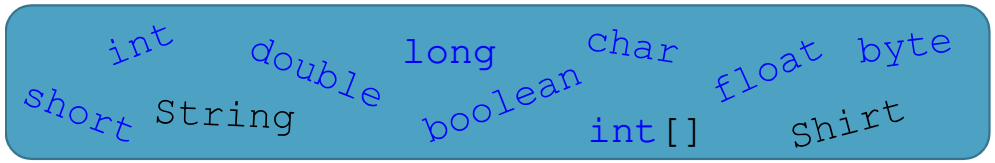
```
public void calculate3(int x, double y, int z){
  System.out.println(x/y +z);
}
```

TIPOS DE RETORNO DE MÉTODOS

- Variáveis podem ter valores de qualquer tipo



- Os métodos também podem devolver valores de qualquer tipo



- Como fazer com que um método devolva um valor:
  - Declarar o método de forma a ser do tipo “non-void return”
  - Usar a palavra reservada return dentro do método, seguido de um valor

EXEMPLOS

- Métodos devem devolver dados que sejam do mesmo tipo que o o seu `return`:

```
public void printString(){
  System.out.println("Hello");
}
```

Métodos void não podem devolver valores em Java

```
public String returnString(){
  return("Hello");
}
```

```
public int sum(int x, int y){
  return(x + y);
}
```

```
public boolean isGreater(int x, int y){
  return(x > y);
}
```

EXEMPLOS

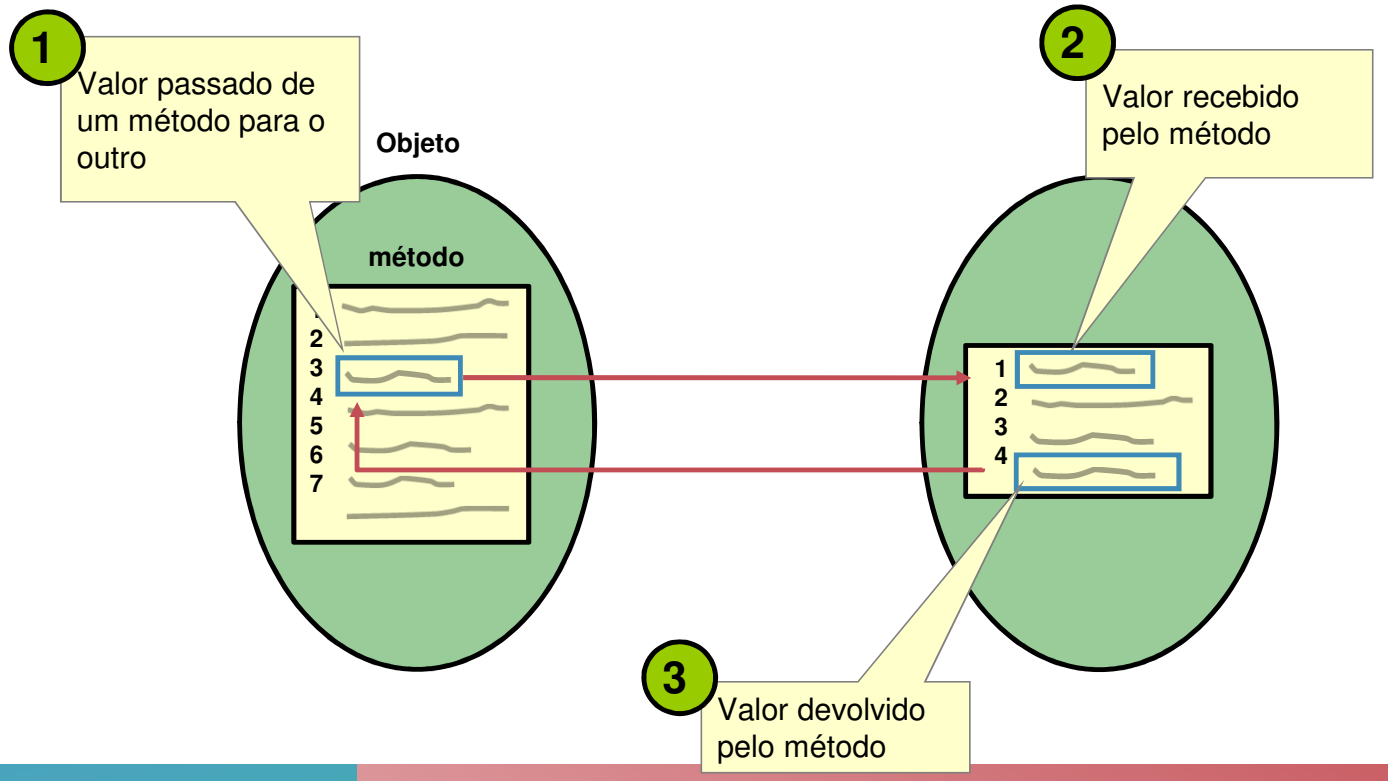
- O código seguinte produz o mesmo resultado:

```
public static void main(String[] args){
   int num1 = 1, num2 = 2;
   int result = num1 + num2;
   System.out.println(result);
}
```

```
public static void main(String[] args){
   int num1 = 1, num2 = 2;
   int result = sum(num1, num2);
   System.out.println(result);
}

public int sum(int x, int y){
   return(x + y);
}
```

PASSAR ARGUMENTOS E DEVOLVER VALORES



CÓDIGO SEM MÉTODOS

```
  public static void main(String[] args){
    Shirt shirt01 = new Shirt();
    Shirt shirt02 = new Shirt();
    Shirt shirt03 = new Shirt();
    Shirt shirt04 = new Shirt();

    shirt01.description = "Sailor";
    shirt01.colorCode = 'B';
    shirt01.price = 30;

    shirt02.description = "Sweatshirt";
    shirt02.colorCode = 'G';
    shirt02.price = 25;

    shirt03.description = "Skull Tee";
    shirt03.colorCode = 'B';
    shirt03.price = 15;

    shirt04.description = "Tropical";
    shirt04.colorCode = 'R';
    shirt04.price = 20;
}
```

CÓDIGO MELHOR COM MÉTODOS

```
  public static void main(String[] args){
    Shirt shirt01 = new Shirt();
    Shirt shirt02 = new Shirt();
    Shirt shirt03 = new Shirt();
    Shirt shirt04 = new Shirt();

    shirt01.setFields("Sailor", 'B', 30);
    shirt02.setFields("Sweatshirt", 'G', 25);
    shirt03.setFields("Skull Tee", 'B', 15);
    shirt04.setFields("Tropical", 'R', 20);
  }
```

```
  public class Shirt {
    public String description;
    public char colorCode;
    public double price;

    public void setFields(String desc, char color, double price){
        this.description = desc;
        this.colorCode = color;
        this.price = price;
  }
 …
```

CÓDIGO AINDA MELHOR COM MÉTODOS

```
public static void main(String[] args){
   Shirt shirt01 = new Shirt("Sailor", "Blue", 30);
   Shirt shirt02 = new Shirt("SweatShirt", "Green", 25);
   Shirt shirt03 = new Shirt("Skull Tee", "Blue", 15);
   Shirt shirt04 = new Shirt("Tropical", "Red", 20);
}
```

```
public class Shirt {
  public String description;
  public char colorCode;
  public double price;

  //Constructor
  public Shirt(String desc, String color, double price){
      setFields(desc, price);
      setColor(color);
  }
  public void setColor (String theColor){
     if (theColor.length() > 0)
         colorCode = theColor.charAt(0);
     }
  }
}
```

ÂMBITO DE VARIÁVEIS

```
public class Shirt {
  public String description;
  public char colorCode;
  public double price;

  public void setColor (String theColor){
     if (theColor.length() > 0)
        colorCode = theColor.charAt(0);
     }
  }

  public String getColor(){
     return theColor; //Cannot find symbol
  }

}
```

Variável de instância (campo)

Variável local

âmbito de theColor

Não é o âmbito de theColor

VANTAGENS DE USAR MÉTODOS

- São reutilizáveis
- Fazem os programas mais pequenos e mais fáceis de ler
- Tornam o desenvolvimento e respetiva manutenção mais rápidos
- Permitem que objetos separados comuniquem e distribuam o trabalho efetuado pelo programa

VARIÁVEIS E MÉTODOS STATIC

- O modificador static é aplicado a um método ou a uma variável
- Significa que o método/variável:
  - Pertence à classe e é partilhado por todos os objetos daquela classe
  - Não é única da instância de um objeto
  - Pode ser acedida sem instanciar a classe
- Comparação:
  - Uma **variável static** é partilhada por todos os objetos de uma classe
  - Uma **variável de instância** é única de um objeto

EXEMPLO

```
public class ItemSizes {
     static final String mSmall = "Men's Small";
     static final String mMed = "Men's Medium";
}
```

```
   Item item1 = new Item();
   item1.setSize(ItemSizes.mMed);
```

```
   public class Item {
       public String size;
       public void setSize(String sizeArg){
          this.size = sizeArg;
       }
   }
```

CRIAR E ACEDER A MEMBROS STATIC

- Para criar uma variável e um método **static**:
  - Static String mSmall;
  - Static void setMSmall(String desc);
- Para aceder a uma variável ou método **static**:
  - De outra classe:
    - ItemSizes.mSmall;
    - ItemSizes.setMSmall(“Men’s Small”);
  - Na própria classe:
    - mSmall;
    - setMSmall(“Men’s Small”);

ANDO LIZAR DOS E AMPOS STATIC

- Realizar operações num objeto específico ou quando associar a variável com tipo específico de objeto não é importante
- Quando for importante aceder ao método ou variável antes de instanciar um objeto
- O método ou variável não pertence, em termos lógicos, a um objeto, mas possivelmente pertence a uma classe utilitária
- Usar valores constantes

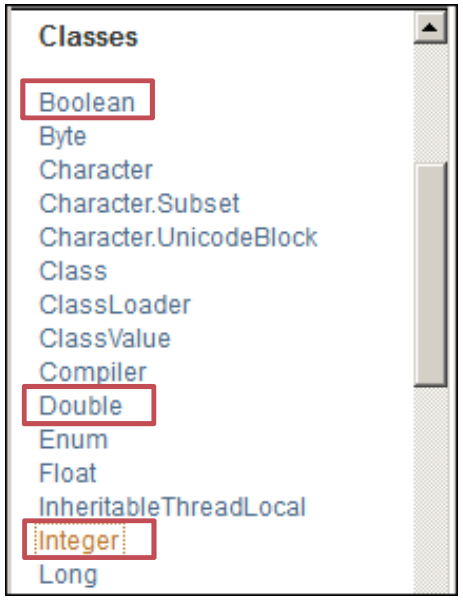
ALGUMAS REGRAS SOBRE MÉTODOS E CAMPOS STATIC

- Métodos de instância conseguem aceder a campos ou métodos static
- Métodos static não conseguem aceder métodos ou campos de instância

```
1 public class Item{
2   int itemID;
3   public Item(){
4       setId();
5   }
6   static int getID(){
7       // itemID de quem??
8   }
```

MÉTODOS E CAMPOS STATIC NA API DO JAVA

- Existem **wrapper classes** para cada tipo de dado primitivo:
  - **Boolean**: Contém um campo único do tipo **boolean**
  - **Double**: Contém um campo único do tipo **double**
  - **Integer**: Contém um campo único do tipo **int**
- Elas também fornecem métodos para trabalhar com dados



CONVERTER DADOS

- Métodos necessários para converter um argumento para um tipo de dado diferente
- Existem vários métodos para efetuar essas conversões

Exemplos:

- Converter uma String para um `int`

```
int myInt1 = Integer.parseInt(s_Num);
```

- Converter uma String para um `double`

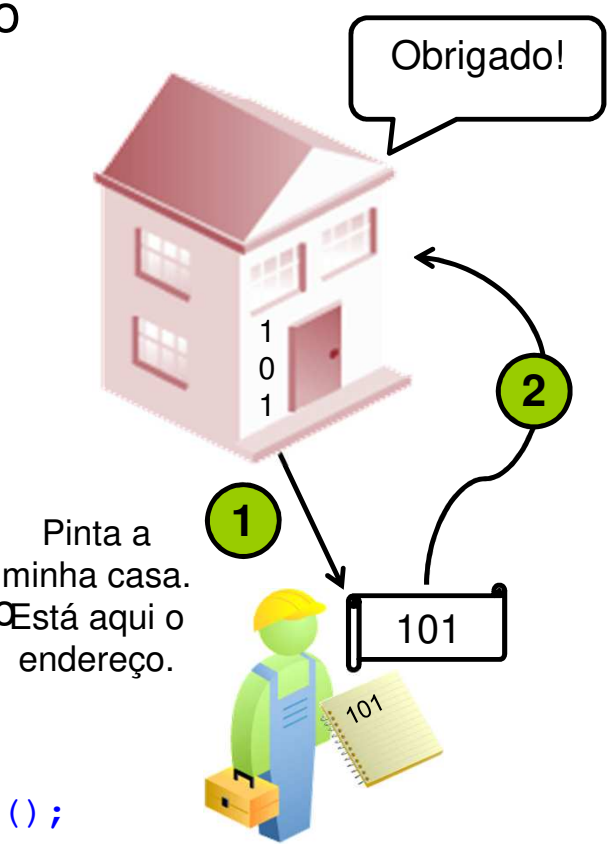
```
double myDbl = Double.parseDouble(s_Num);
```

- Converter uma String para um `boolean`

```
boolean myBool = Boolean.valueOf(s_Bool);
```

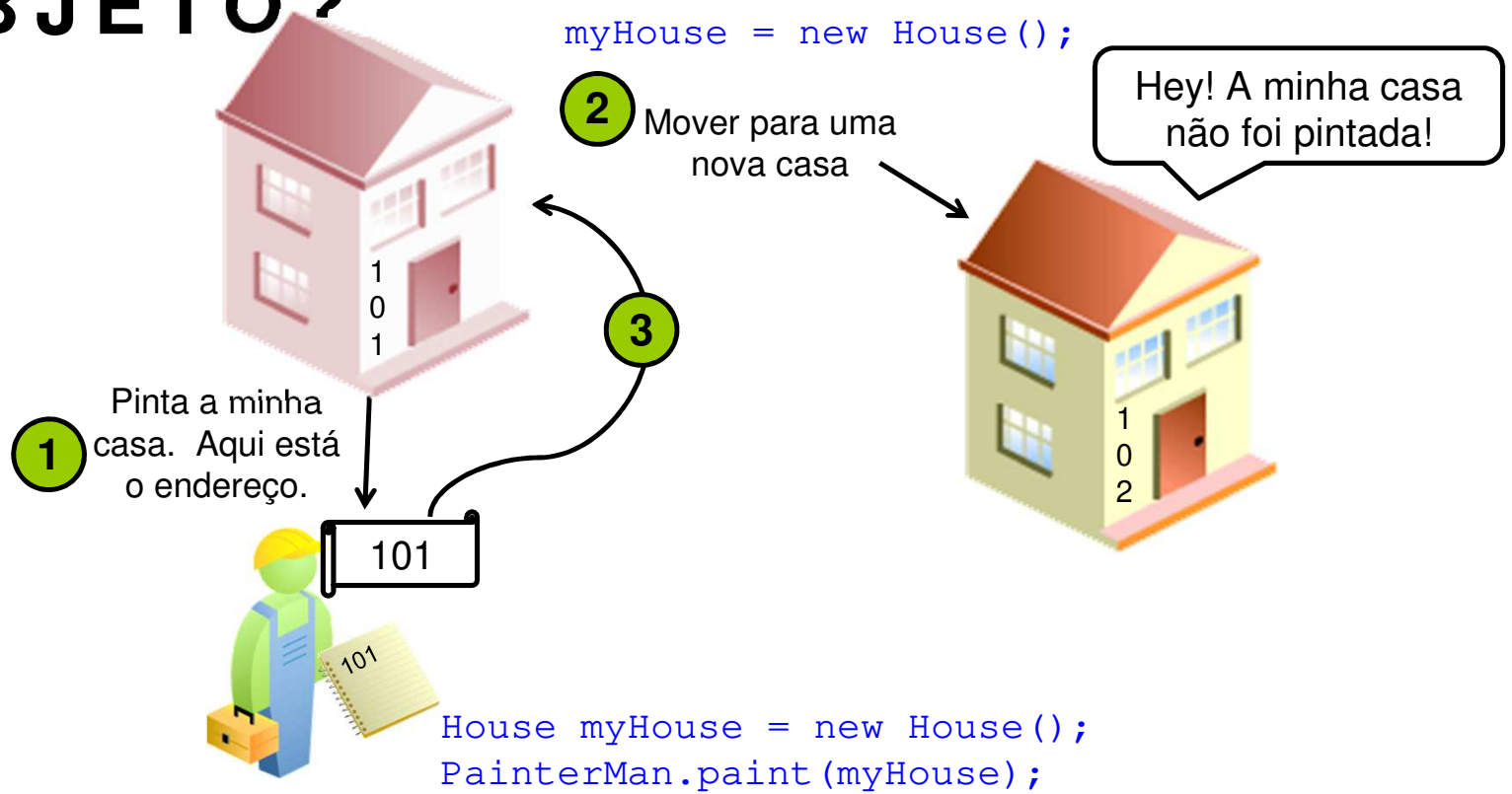
PASSAR UMA REFERÊNCIA DE UM OBJETO

- Uma referência a um objeto é similar ao endereço de uma casa. Quando é passado para um método:
  - O objeto em si não é passado
  - O método consegue aceder ao objeto através da sua referência
  - O método consegue atuar no objeto




```
House myHouse = new House();
PainterMan.paint(myHouse);
```

MAS E SE EXISTIR UM NOVO OBJETO?



EXEMPLO CARRINHO DE COMPRAS

```
public class ShoppingCart {
    public static void main (String[] args) {
        Shirt myShirt = new Shirt();
        System.out.println("Shirt color: " + myShirt.colorCode);
        changeShirtColor(myShirt, 'B');
        System.out.println("Shirt color: " + myShirt.colorCode);
    }
    public static void changeShirtColor(Shirt theShirt, char color) {
        theShirt.colorCode = color;     }
}
```

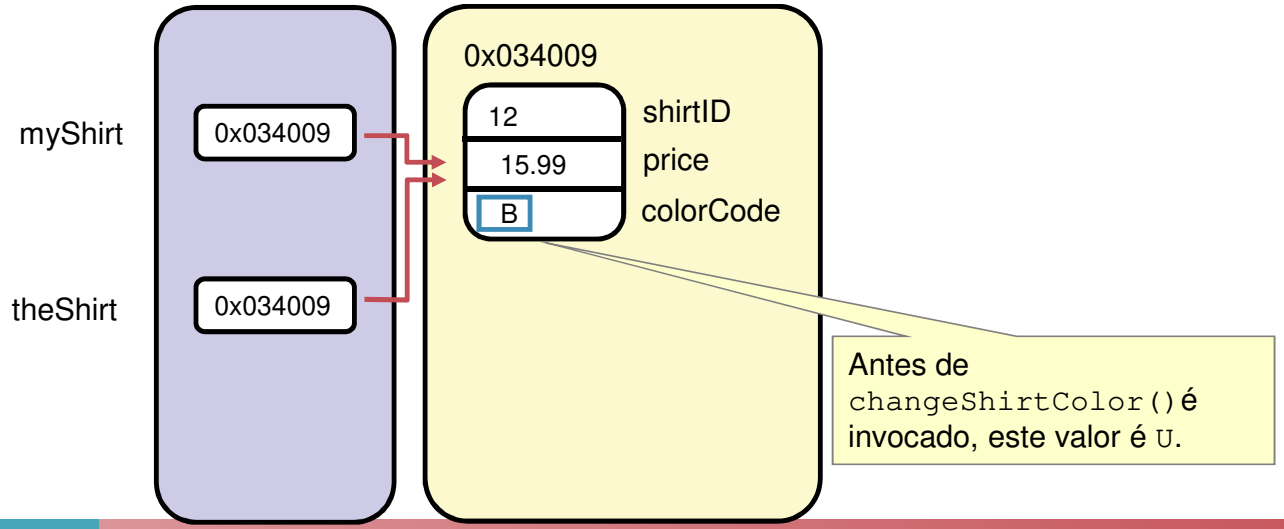
theShirt é uma nova referência do tipo Shirt

Output:

```
Shirt color: U
Shirt color: B
```

PASSAR POR VALOR

```
Shirt myShirt = new Shirt();
changeShirtColor(myShirt, 'B');
```



REATRIBUIR A REFERÊNCIA

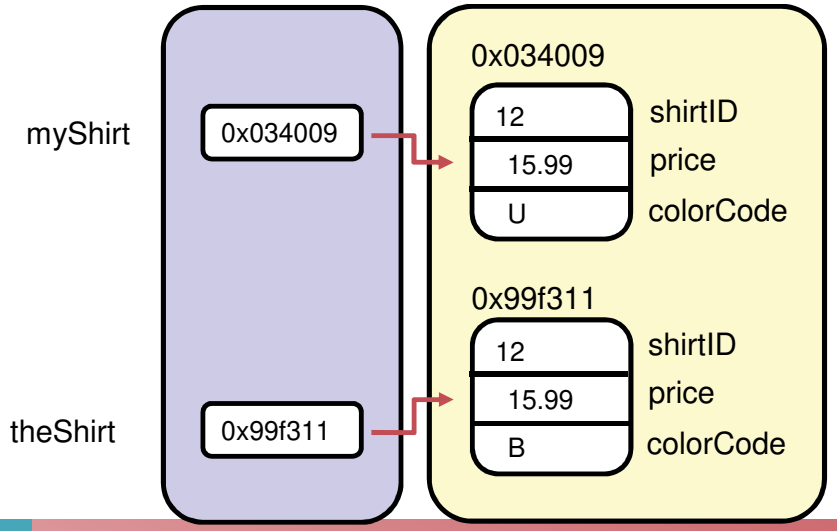
```
 1 public class ShoppingCart {
 2     public static void main (String[] args) {
 3         Shirt myShirt = new Shirt();
 4         System.out.println("Shirt color: " + myShirt.colorCode);
 5         changeShirtColor(myShirt, 'B');
 6          System.out.println("Shirt color: " + myShirt.colorCode);
 7     }
 9     public static void changeShirtColor(Shirt theShirt, char color) {
10         theShirt = new Shirt();
11         theShirt.colorCode = color;
12 }
```

Output:

```
Shirt color: U
Shirt color: U
```

PASSAR POR VALOR

```
Shirt myShirt = new Shirt();
changeShirtColor(myShirt, 'B');
```



OVERLOADING DE MÉTODOS

- Têm o mesmo nome

- Têm assinaturas diferentes:
  - **Número** de parâmetros
  - **Tipo** de parâmetros
  - **Ordem** dos parâmetros

- Podem ter funcionalidades diferentes ou similares

- São largamente utilizados em classes fundamentais do Java

OVERLOADING DE MÉTODOS

Tipo de método

Assinatura do método

```
public final class Calculator {
        public static int sum(int num1, int num2){
                System.out.println("Method One");
                return num1 + num2;
        }

        public static float sum(float num1, float num2) {
                System.out.println("Method Two");
                return num1 + num2;
        }
        public static float sum(int num1, float num2) {
                System.out.println("Method Three");
                return num1 + numb2;
        }
}
```

OVERLOADING DE MÉTODOS

```
public class CalculatorTest {


        public static void main(String[] args) {
                int totalOne = Calculator.sum(2, 3);
                System.out.println("The total is " + totalOne);


                float totalTwo = Calculator.sum(15.99F, 12.85F);
                System.out.println(totalTwo);


                float totalThree = Calculator.sum(2, 12.85F);
                System.out.println(totalThree);
        }
}
```

ENCAPSULAMENTO

- Significa esconder atributos de objetos, colocando todos os campos como **private**:
  - Usar métodos **getters** e **setters**
  - Nos métodos **setters**, utilizar código para garantir que os valores são válidos

- O encapsulamento sugere programar para "**interfaces**":
  - O tipo de dado do campo é irrelevante para o método invocador
  - A classe pode ser modificada desde que a "**interface**" continue a mesma

- Encoraja um bom desenho de programação orientada a objetos

MODIFICADOR PUBLIC

```
public class Elevator {
          public boolean doorOpen=false;
          public int currentFloor = 1;
          public final int TOP_FLOOR = 10;
          public final int MIN_FLOOR = 1;

          ... < Cdigo omitido> ...

          public void goUp() {
                    if (currentFloor == TOP_FLOOR) {
                              System.out.println("Cannot go up further!");
                    }
                    if (currentFloor < TOP_FLOOR) {
                              currentFloor++;
                              System.out.println("Floor: " + currentFloor);
                    }
          }
}
```

PERIGOS DE DEFINIR UM CAMPO PÚBLICO

```
Elevator theElevator = new Elevator();


theElevator.currentFloor = 15;
```


Pode causar problemas!

MODIFICADOR PRIVATE

```
public class Elevator {
        private boolean doorOpen=false;
        private int currentFloor = 1;
        private final int TOP_FLOOR = 10;
        private final int MIN_FLOOR = 1;

        ... < código omitido > ...

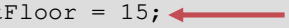
        public void goUp() {
                if (currentFloor == TOP_FLOOR) {
                        System.out.println("Cannot go up further!");
                }
                if (currentFloor < TOP_FLOOR) {
                        currentFloor++;
                        System.out.println("Floor: " + currentFloor);
                }
        }
}
```

Nenhum destes campos pode agora ser acedido através de outra classe, utilizando o operador .

TENTAR ACEDER A UM CAMPO PRIVATE

```
Elevator theElevator = new Elevator();


theElevator.currentFloor = 15;
```



NetBeans vai apresentar um erro

```
1  public class ElevatorTest {
2
3      public static void main(String args[]) {
4  currentFloorIndex has private access in Loops_Elevator
5  --                                                    Elevator();
6  (Alt-Enter shows hints)
7          theElevator.currentFloorIndex = 17;
8
9
10
```

MODIFICADOR PRIVATE EM MÉTODOS

```
public class Elevator {

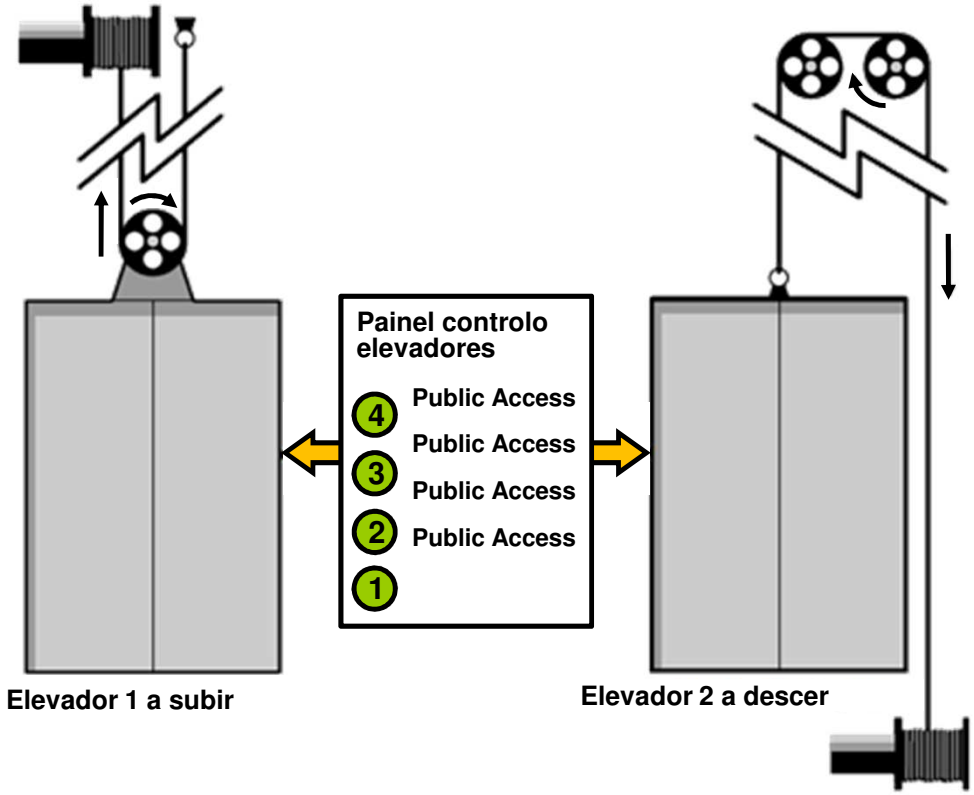
        ... < código omitido > ...


      private void setFloor() {
                int desiredFloor = 5;
                while (  currentFloor != desiredFloor  ){
                        if (currentFloor < desiredFloor) {
                                goUp();
                        } else {
                                goDown();
                        }
                }
        }


        public void requestFloor(int desiredFloor) {
                ... < contém Código para adicionar o pedido para um andar
                numa queue > ...
        }
```

Devia este método ser private?

E R F A C E ” E
M E N T A Ç Ã O



MÉTODOS GETTERS E SETTERS

```
public class Shirt {
        private int shirtID = 0; // Default ID for the shirt
        private String description = "-description required-";
        private char colorCode = 'U';
        private double price = 0.0; // Default price for all items

        public char getColorCode() {
                return colorCode;
        }

        public void setColorCode(char newCode) {
                colorCode = newCode;
        }
        // Métodos adicionais get/set para shirtID, description e price
}
```

UTILIZANDO MÉTODOS GETTERS E SETTERS

```
public class ShirtTest {

        public static void main (String[] args) {

        Shirt theShirt = new Shirt();

        char colorCode;


        theShirt.setColorCode('R');

        colorCode = theShirt.getColorCode();

        System.out.println("Color Code: " + colorCode);


        theShirt.setColorCode('Z');

        colorCode = theShirt.getColorCode();

        // The ShirtTest class can set and get an invalid colorCode

        System.out.println("Color Code: " + colorCode);

}
```

código de cor inválido

MÉTODO SETTER COM VERIFICAÇÃO

```
public void setColorCode(char newCode) {
          switch (newCode) {
                    case 'R':
                    case 'G':
                    case 'B':
                              colorCode = newCode;
                              break;
                    default:
                              System.out.println("Invalid colorCode. Use R, G, or B");
          }
}
```

USAR MÉTODOS SETTER E GETTER

```
public class ShirtTest {

          public static void main (String[] args) {

          Shirt theShirt = new Shirt();

          System.out.println("Color Code: " + theShirt.getColorCode());


          Shirt1.setColorCode('Z');

          System.out.println("Color Code: " + theShirt.getColorCode());

}
```

código de cor inválido

```
Color Code: U
Invalid colorCode. Use R, G, or B
Color Code: U
```

Antes de invocar `setColorCode()` - mostra valor por omissão

Invocar `setColorCode` mostra mensagem erro

`colorCode` não é modificado por causa de argumetno inválido passado para `setColorCode()`

INICIALIZAR OBJETOS

```
public class ShirtTest {

        public static void main (String[] args) {
                Shirt theShirt = new Shirt();


                theShirt.setColorCode('R');
                theShirt.setDescription("Outdoors shirt");
                theShirt.price(39.99);
        }
}
```

CONSTRUTORES

- São métodos de uma classe:
  - Têm o mesmo nome da classe
  - Usados habitualmente para inicializar campos de um objeto
  - Podem ter argumentos
  - Podem ser alvo de overload
- Todas as classes têm pelo menos um construtor:
  - Se não for criado nenhuma, o compilador do Java cria um construtor sem argumentos em tempo de compilação
  - Se houver um ou mais construtores criados, o Java não efetua a ação descrita em cima

CRIAR CONSTRUTORES - SINTAXE

```
[modificadores] class NomeClasse {


          [modifiers] NomeClasse([argumentos]) {

                    bloco_código

          }

}
```

CRIAR CONSTRUTORES

```
public class Shirt {

        public int shirtID = 0;

        public String description = "-description required-";

        private char colorCode = 'U';

        public double price = 0.0;


        public Shirt(char colorCode) {

                setColorCode(colorCode);

        }

}
```

INICIALIZAR UM OBJETO DO TIPO SHIRT COM CONSTRUTOR

```
public class ShirtTest {

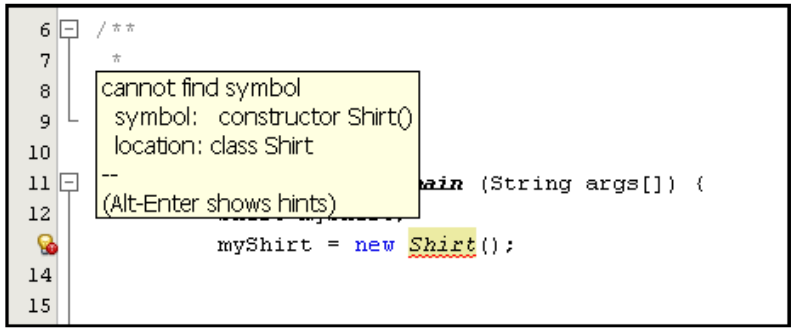
          public static void main (String[] args) {

                    Shirt theShirt = new Shirt('G');

                    theShirt.display();

          }

}
```



VÁRIOS CONSTRUTORES

```
public class Shirt {
        ... < declarations for field omitted > ...

        public Shirt() {
                // Pode-se adicionar código aqui
        }
        // Este constructor leva um argumento
        public Shirt(char colorCode) {
                setColorCode(colorCode);
        }

        public Shirt(char colorCode, double price) {
                this(colorCode);
                setPrice(price);
        }
}
```

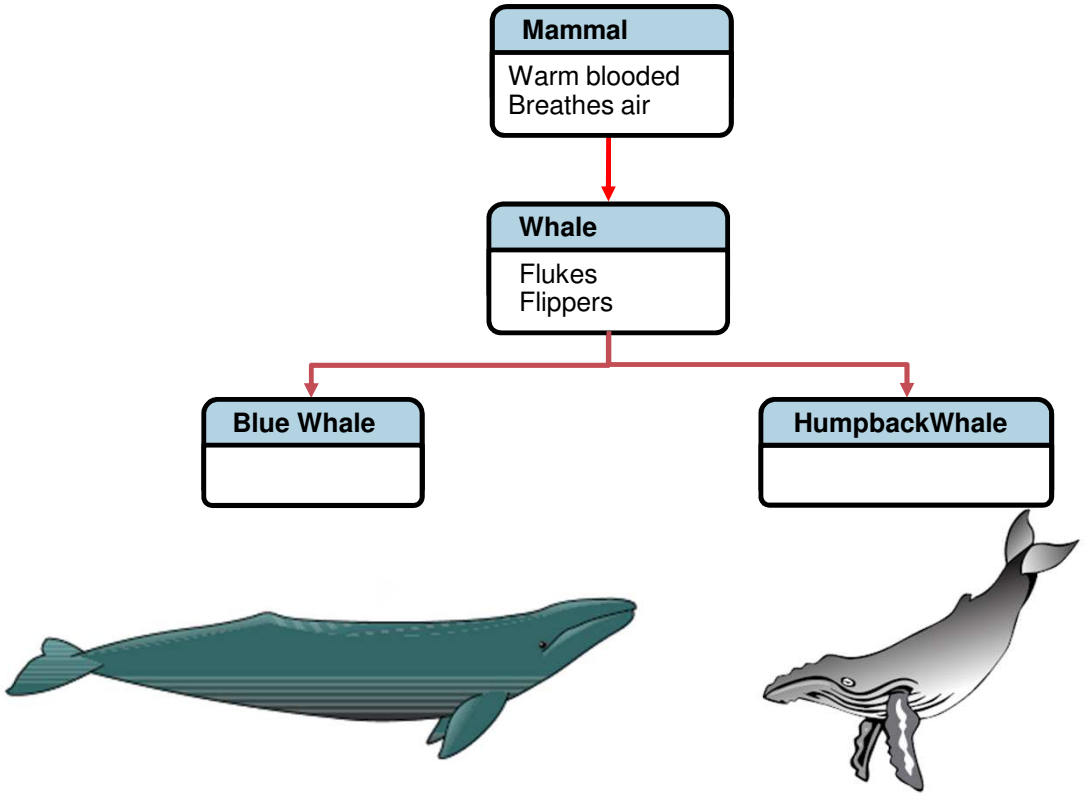
Se necessário, tem de ser escrito

Encadeamento de construtores

HERANÇA DE CLASSES

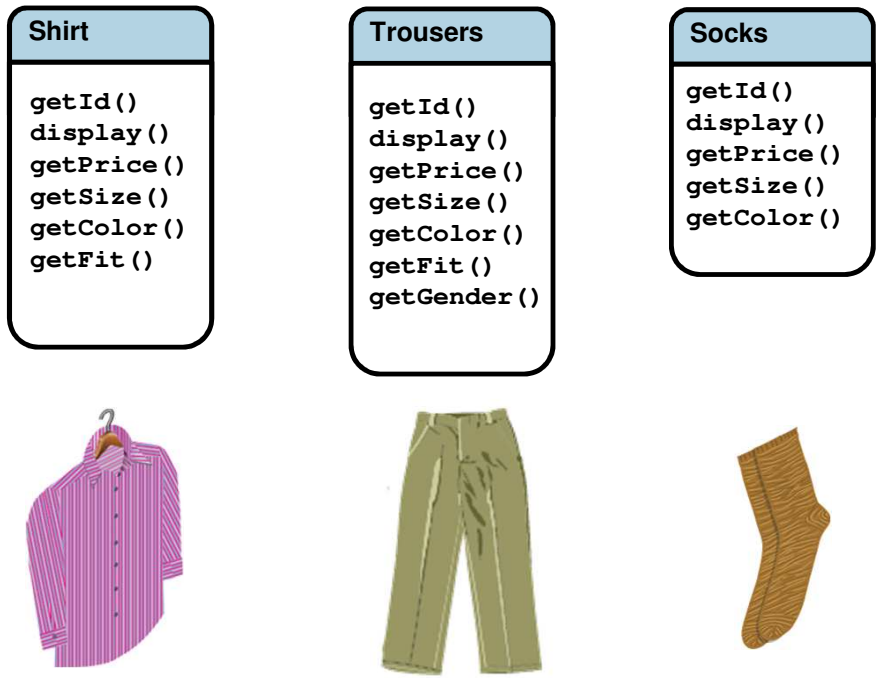
PARTE V

HIERARQUIA DE CLASSES

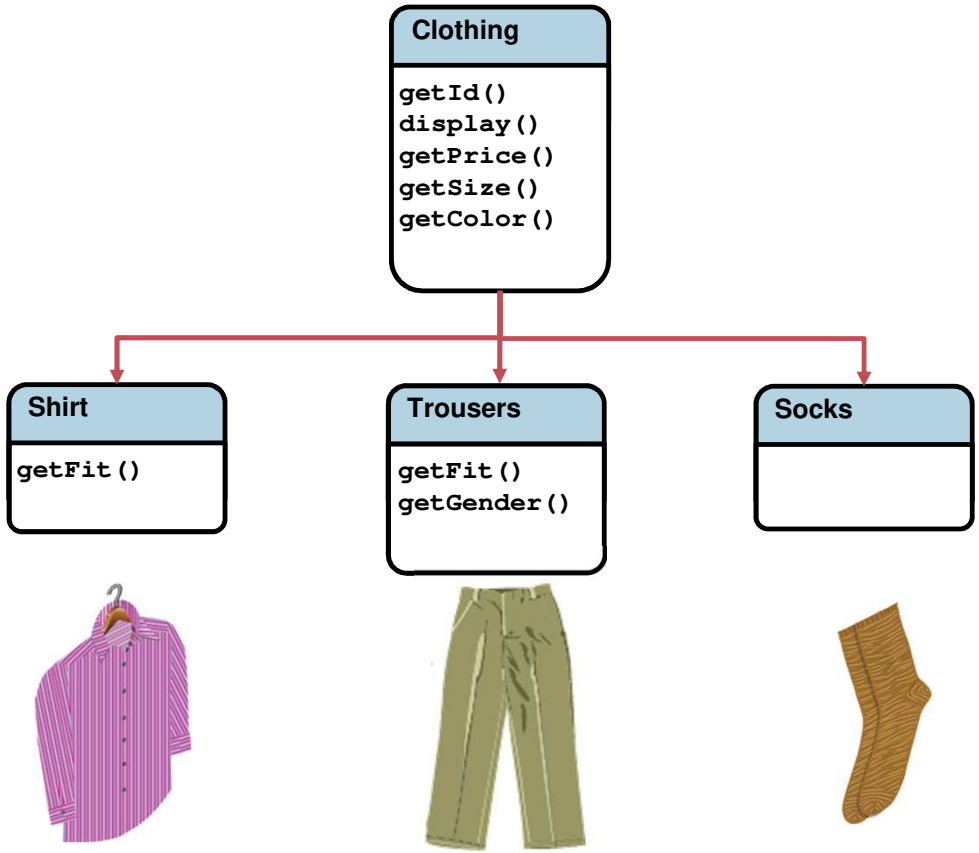


Shirt	Trousers
getId() getPrice() getSize() getColor() getFit()	getId() getPrice() getSize() getColor() getFit() getGender()
setId() setPrice() setSize() setColor() setFit()	setId() setPrice() setSize() setColor() setFit() setGender()
display()	display()

DUPLICAÇÃO DE CÓDIGO



HERANÇA


Clothing
getId()
display()
getPrice()
getSize()
getColor()
Shirt
getFit()
Trousers
getFit()
getGender()
Socks

FAZER OVERRIDING A MÉTODOS DA SUPERCLASSE

- Métodos que existam numa superclasse podem:
  - Não são implementados na subclasse
    - O método declarado na superclasse é usado em tempo de execução
  - Serem implementados na subclasse
    - O método declarado na subclasse é usado em tempo de execução

SUPERCLASS E CLOTHING – 1/3

```
public class Clothing  {

        private int itemID = 0;

        private String description = "-description required-";

        private char colorCode = 'U';

        private double price = 0.0;


        public Clothing(int itemID, String description, char colorCode,
        double price) {

                this.itemID = itemID;

                this.description = description;

                this.colorCode = colorCode;

                this.price = price;

        }
```

SUPERCLASS E CLOTHING – 2/3

```
public void display() {
        System.out.println("Item ID: " + getItemID());
        System.out.println("Item description: " + description);
        System.out.println("Item price: " + getPrice());
        System.out.println("Color code: " + getColorCode());
}

public String getDescription(){
        return description;
}

public double getPrice() {
        return price;
}

public int getItemID() {
        return itemID;
}
```

SUPERCLASS E CLOTHING – 3/3

```
        public char getColorCode() {
                return colorCode;
        }
        public void setItemID(int itemID) {
                this.itemID = itemID;
        }
        public void setDescription(String description) {
                this.description = description;
        }
        public void setColorCode(char colorCode) {
                this.colorCode = colorCode;
        }
        public void setPrice(double price) {
                this.price = price;
        }
}
```

DECLARAR UMA SUBCLASSE - SINTAXE

- *[modificador] class NomeSubclasse extends NomeSuperclasse*

DECLARAR UMA SUBCLASSE – 1/2

```
public class Shirt extends Clothing {

  private char fit = 'U';

  public Shirt(int itemID, String description, char colorCode,
               double price, char fit) {
     super(itemID, description, colorCode, price);

     this.fit = fit;
  }

  public char getFit() {
      return fit;
  }
  public void setFit(char fit) {
      this.fit = fit;
  }
```

Assegura que Shirt herda os membros de Clothing

`super` é a referência para os métodos e atributos da superclasse

`this` é a referência para este objeto

DECLARAR UMA SUBCLASSE – 2/2

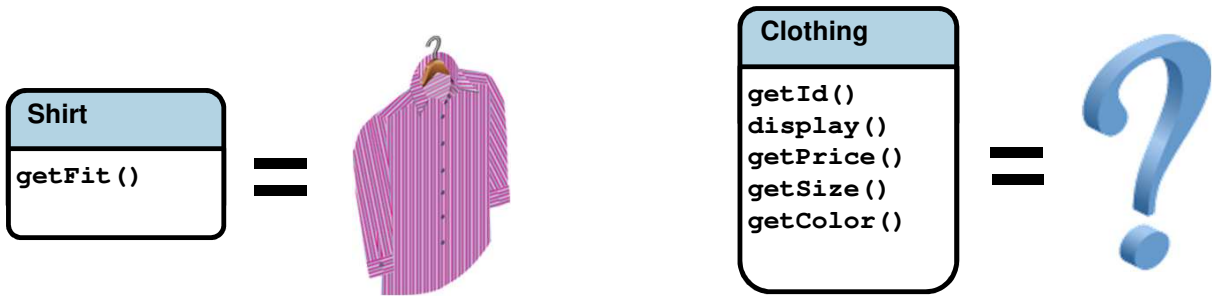
```
 //Este método faz override ao método display na superclasse Clothing
 public void display() {
   System.out.println("Shirt ID: " + getItemID());
   System.out.println("Shirt description: " + description);
   System.out.println("Shirt price: " + getPrice());
   System.out.println("Color code: " + getColorCode());
   System.out.println("Fit: " + getFit());
 }

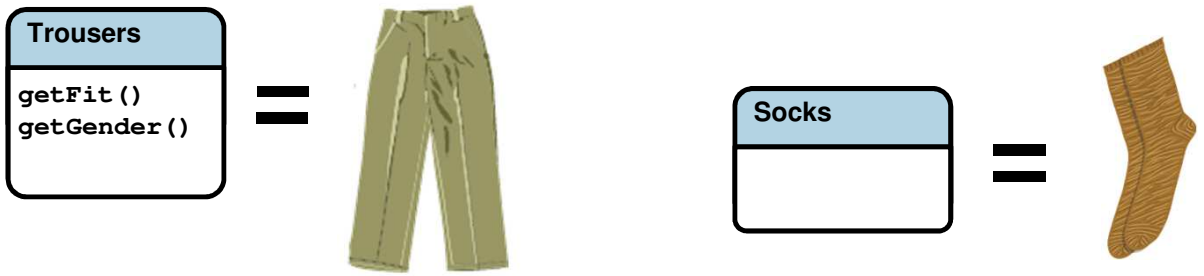
 // Este método faz override ao método setColorCode na superclasse
 public void setColorCode(char colorCode) {

     ... incluir código aqui para verificar um código de cor correto ...

     this.colorCode = colorCode;
 }
}
```

CLASSES ABSTRATAS





CLASSE ABSTRATA CLOTHING – 1/2

```
public abstract class Clothing  {
  private int itemID = 0; // Default ID for all clothing items
  private String description = "-description required-"; // default
  private char colorCode = 'U'; //'U' is Unset
  private double price = 0.0; // Default price for all items

  public Clothing(int itemID, String description, char colorCode,
    double price, int quantityInStock) {
    this.itemID = itemID;
    this.description = description;
    this.colorCode = colorCode;
    this.price = price;
  }
```

A palavra reservada `abstract` assegura que a classe não pode ser instanciada

CLASSE ABSTRATA CLOTHING – 2/2

```
  public abstract char getColorCode() ;


  public abstract void setColorCode(char colorCode);


   ... Outros métods não apresentados aqui ...

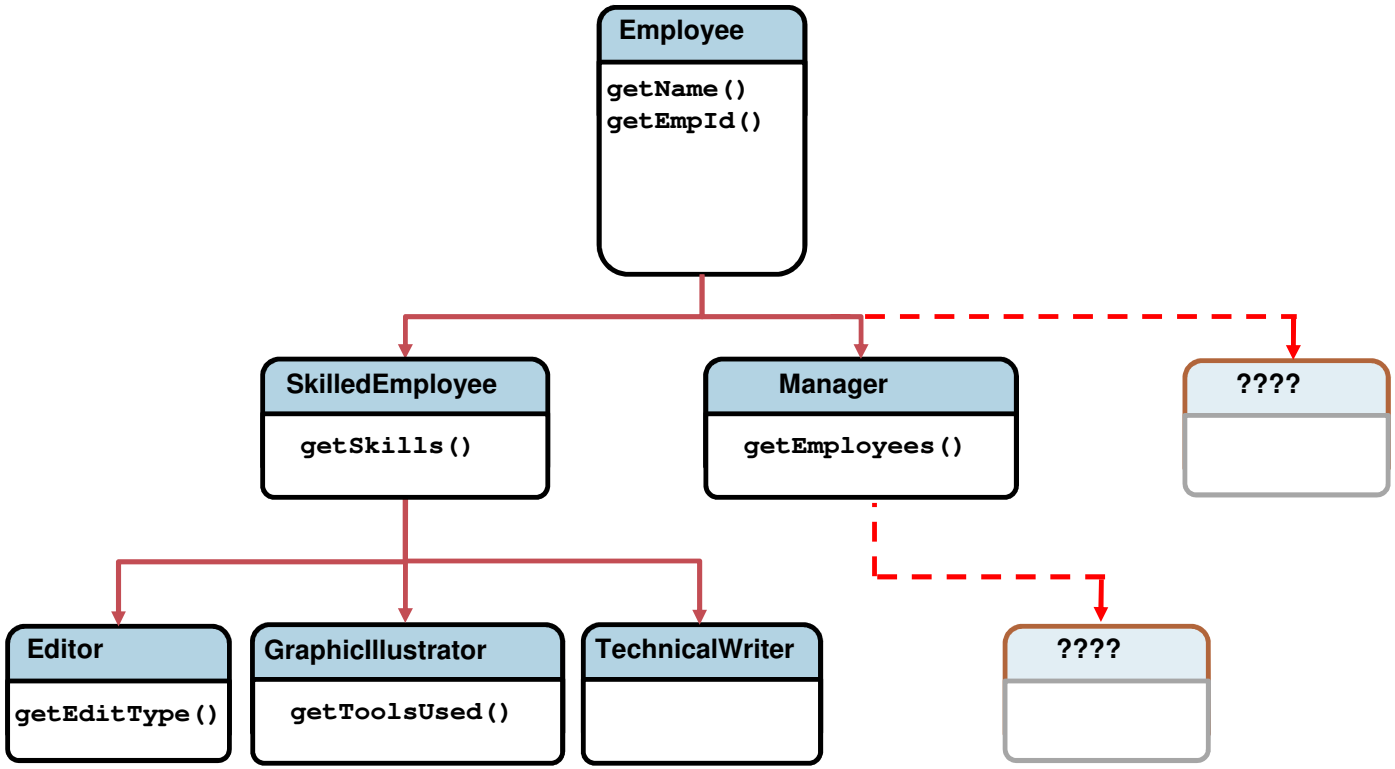

}
```

A palavra reservada `abstract` assegura que estes métodos têm de ser reescritos na subclasse

RELAÇÕES ENTRE SUPERCLASSE E SUBCLASSE

- É muito importante considerar o melhor uso da herança:
  - Usar herança apenas quando é completamente válido e necessário
  - Verificar a questão da herança com uma frase do tipo “is a”:
    - A frase “a shirt is a piece of clothing” expressa uma relação válida de herança
    - A frase “an hat is a sock” expressa uma relação invalidade de herança

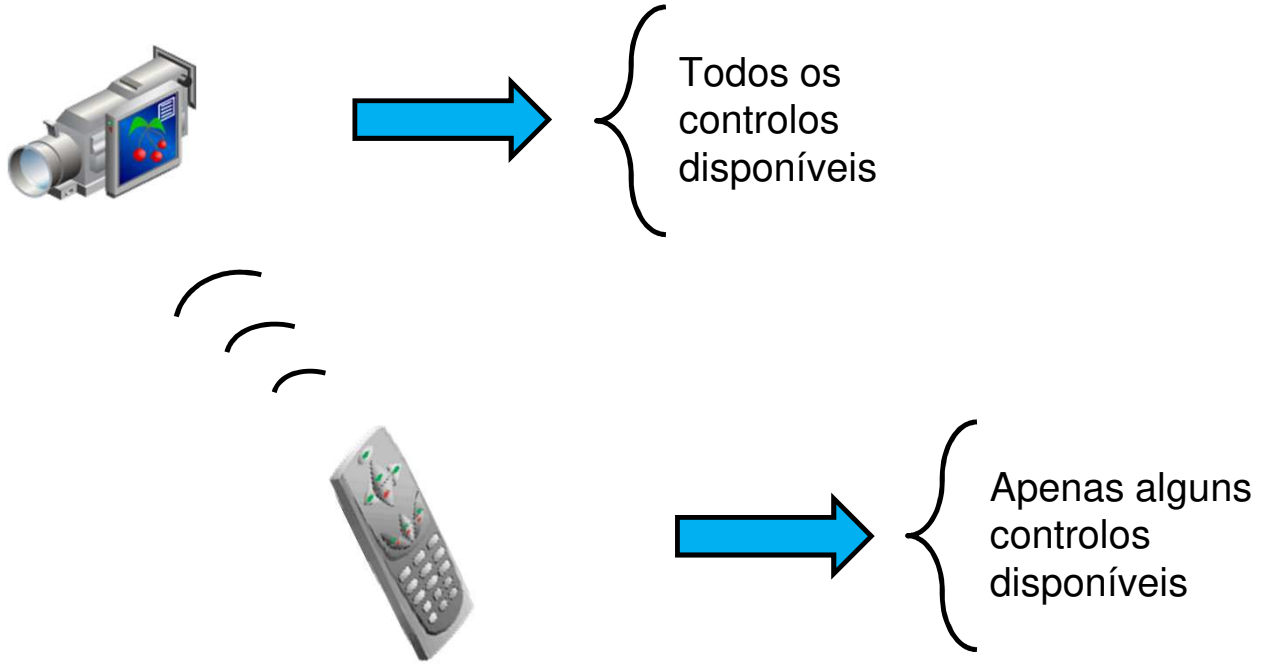
RELAÇÕES ENTRE SUPERCLASSE E SUBCLASSE



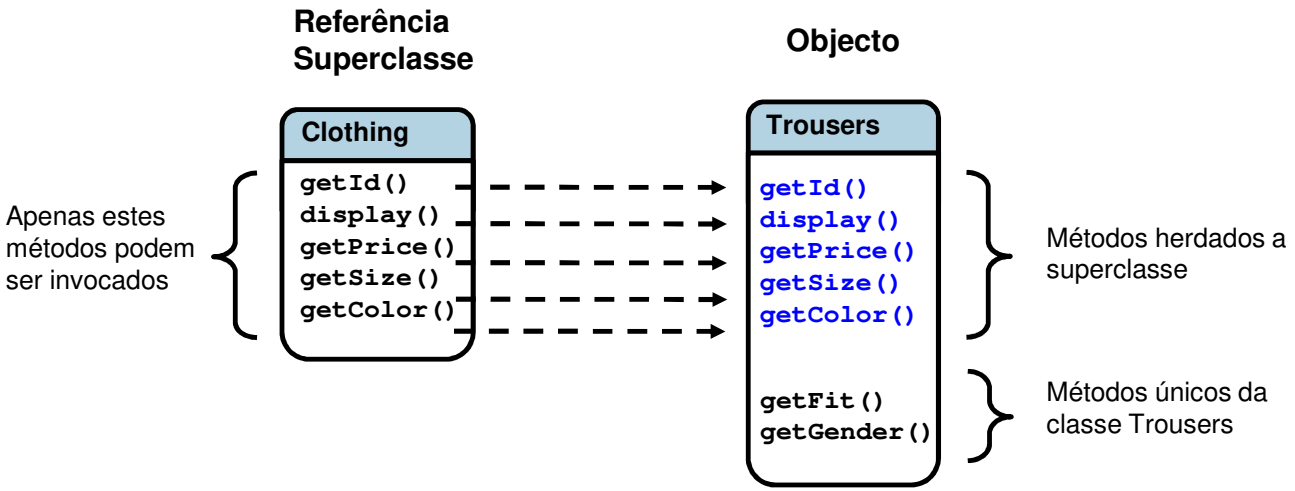
TIPOS DE REFERÊNCIA SUPERCLASSE

- Até agora viu-se a classe a utilizar referências do tipo do objeto criado:
  - Para usar a classe Shirt como tipo de referência do objeto Shirt:
    - Shirt myShirt = new Shirt();
  - Mas também é possível utilizar a superclasse como referência:
    - Clothing clothingItem1 = new Shirt();
    - Clothing clothingItem2 = new Trousers();

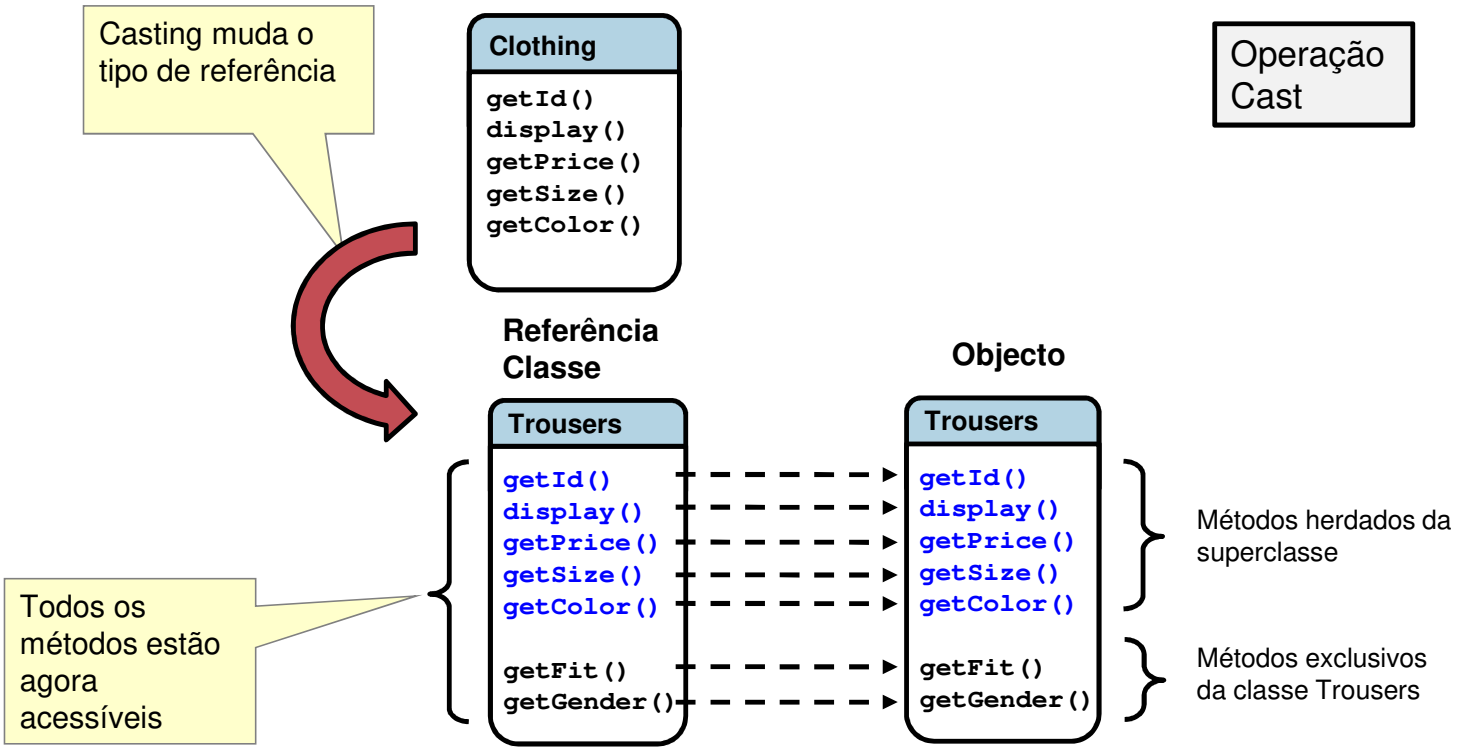
ACEDER ÀS FUNCIONALIDADES DO OBJETO



ACEDER MÉTODOS DA CLASSE APARTIR DA SUPERCLASSE



FAZER CASTING AO TIPO REFERÊNCIA



FAZER CASTING

```
Clothing cl = new Trousers(123, "Dress Trousers", 'B', 17.00, 4, 'S');

cl.display();

char fitCode = cl.getFit(); // Isto não compila

char fitCode = ((Trousers)cl).getFit(); // Isto compila
```

Os parêntesis à volta do `cl` asseguram que o cast vai ser aplicado a esta referência

A sintaxe para fazer cast é o tipo para o qual se quer fazer cast dentro de parêntesis colocados antes da referência alvo do cast

OPERADOR INSTANCEOF

Possível erro de casting:

```
    public static void displayDetails(Clothing cl) {

        cl.display();

        char fitCode = ((Trousers) cl).getFitCode();

        System.out.println("Fit: " + fitCode);

}
```

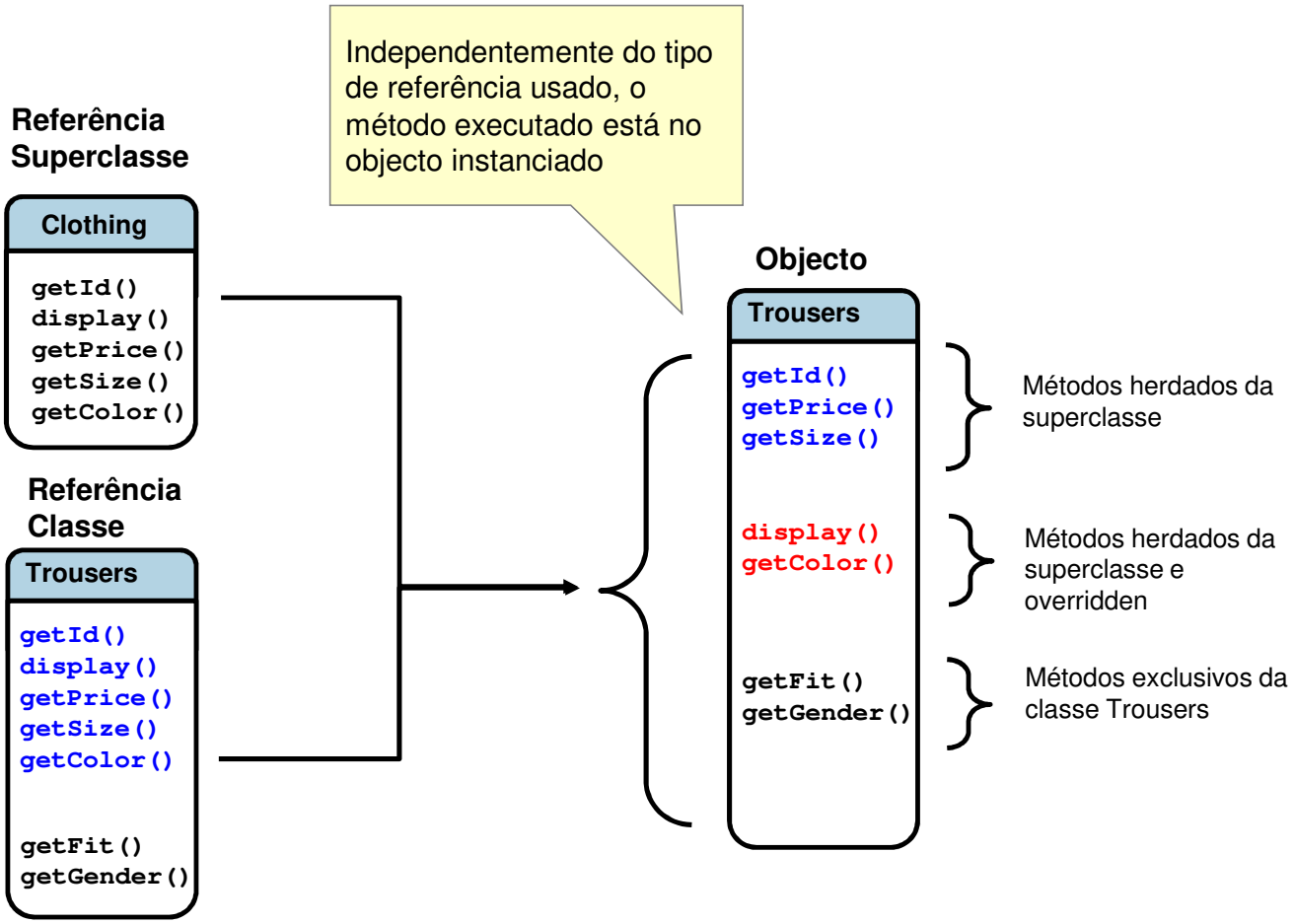
operador `instanceof` usado para assegurar que não ja erro de casting:

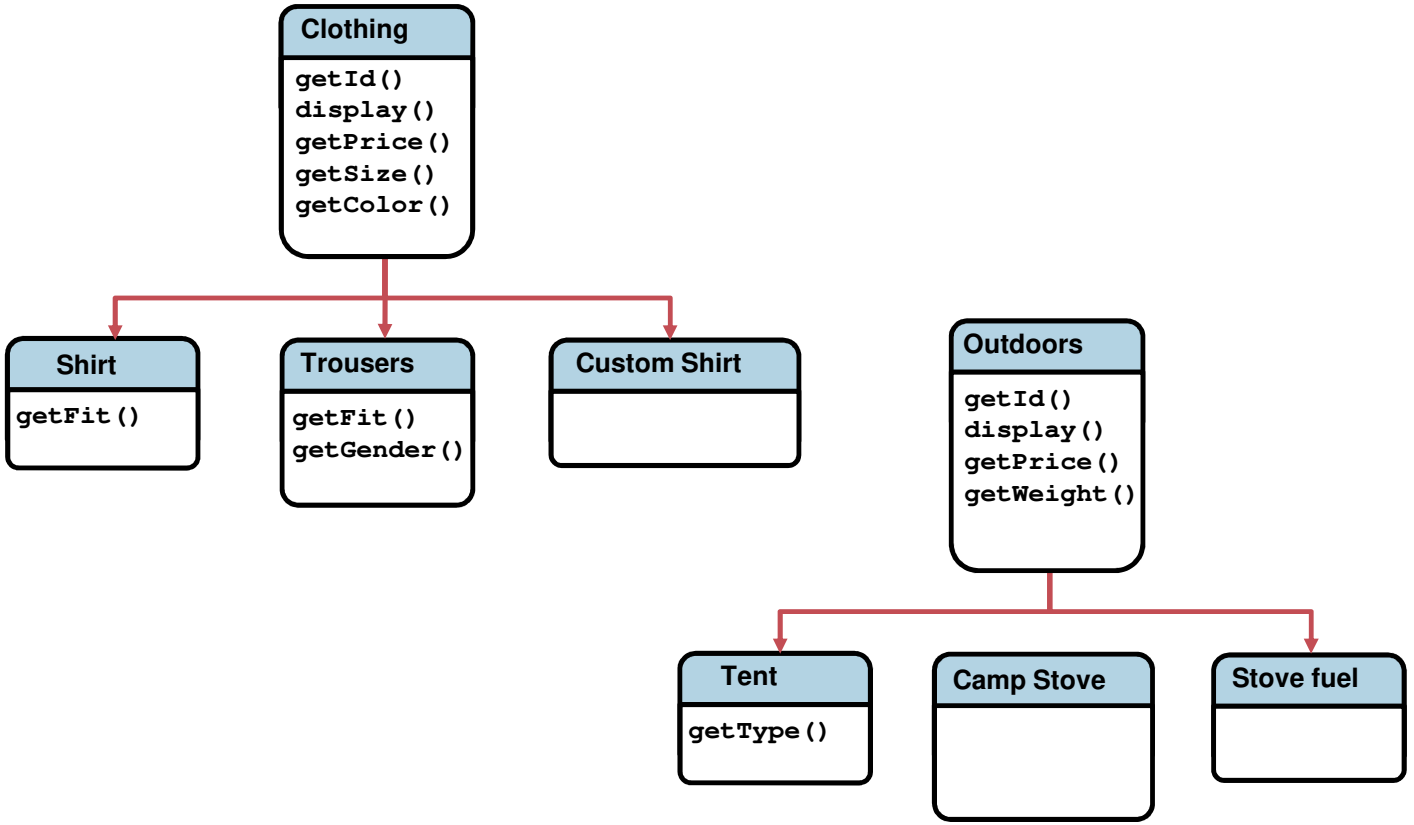
```
    public static void displayDetails(Clothing cl) {
        cl.display();
        if (cl instanceof Trousers) {
            char fitCode = ((Trousers) cl).getFitCode();
            System.out.println("Fit: " + fitCode);
        }

        else { // Faz outra ação }
```

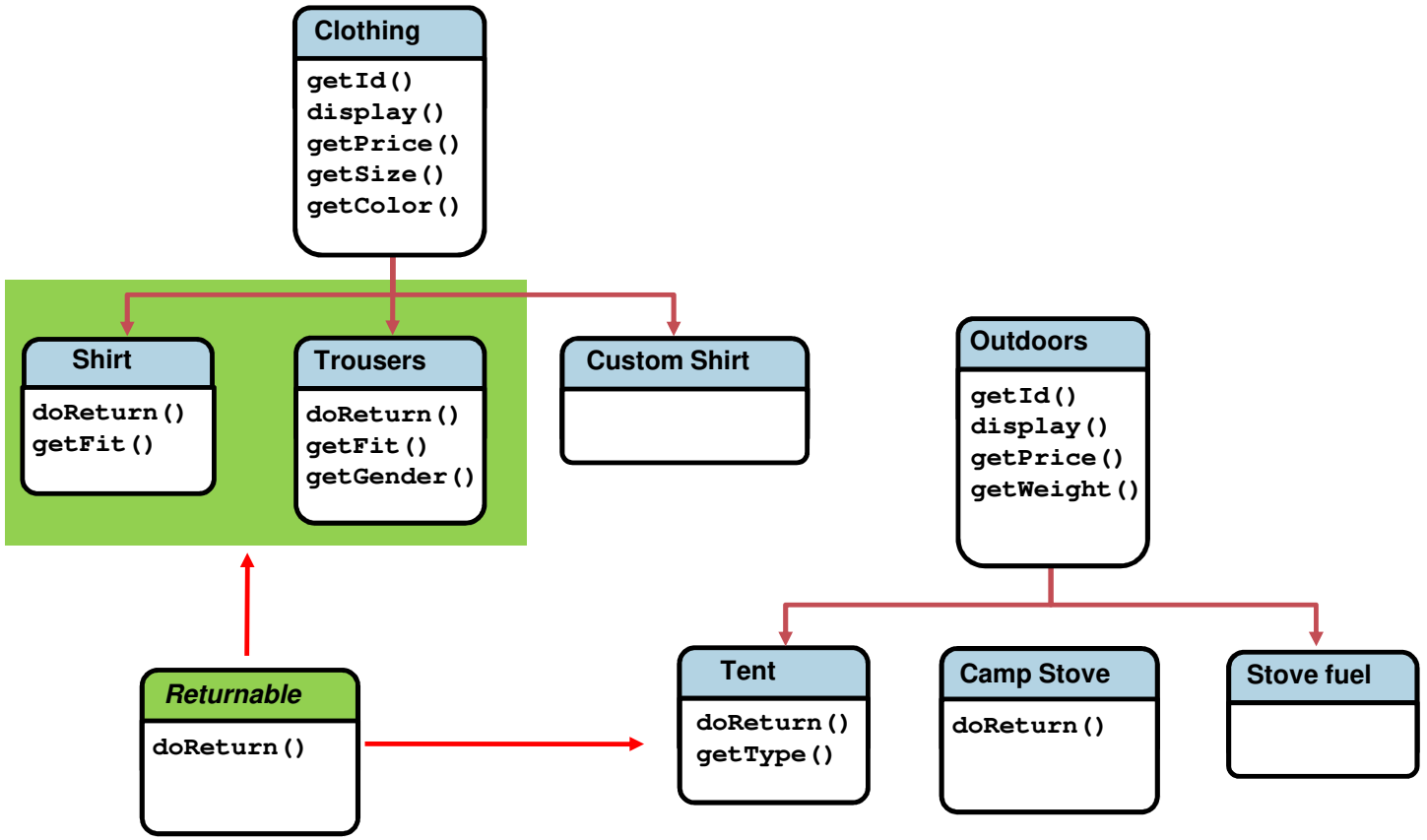
operador `instanceof` devolve `true`, se o objeto referenciado por `cl` é um objeto Trousers

INVOCAÇÃO POLIMÓRFICA DE MÉTODOS





HERANÇA MÚLTIPLA


Clothing
getId()
display()
getPrice()
getSize()
getColor()
Shirt
doReturn()
getFit()
Trousers
doReturn()
getFit()
getGender()
Custom Shirt
Outdoors
getId()
display()
getPrice()
getWeight()
Returnable
doReturn()
Tent
doReturn()
getType()
Camp Stove
doReturn()
Stove fuel

INTERFACES

IMPLEMENTAR A INTERFACE RETURNABLE

Interface Returnable

```
public interface Returnable {

  public String doReturn();

}
```

Como um método abstract, apenas tem assinatura

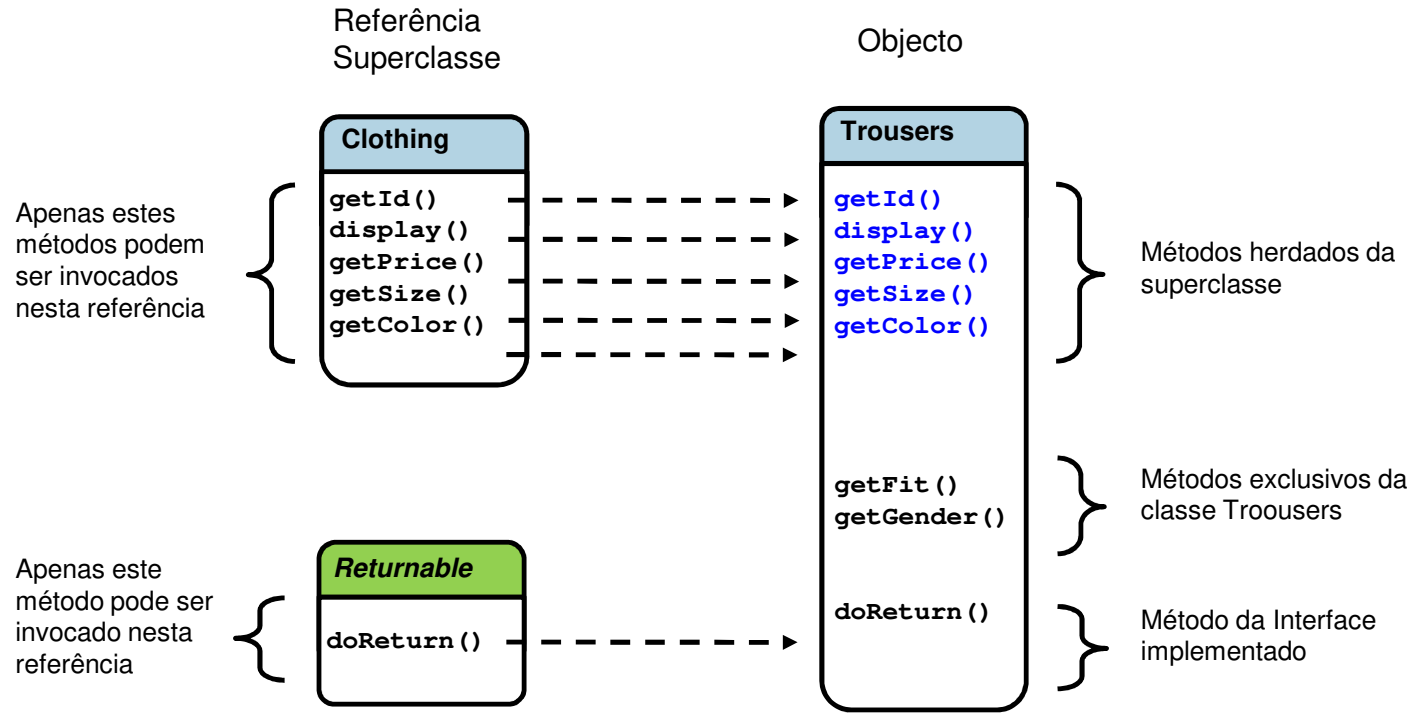
Classe Shirt

```
public class Shirt extends Clothing implements Returnable {
   public Shirt(int itemID, String description, char colorCode,
                double price, char fit) {
      super(itemID, description, colorCode, price);
      this.fit = fit;
   }
 public String doReturn() {
    // Ver notas em baixo
    return "Suit returns must be within 3 days";
 }
 ...< outros métdos por msostar > ...       }
```

Assegura que Shirt tem de implementar todos os métodos de Returnable

Método declarado na interface Returnable

ACEDER A MÉTODOS DO OBJETO ATRAVÉS DA INTERFACE



TRATAMENTO DE ERROS E EXCEPÇÕES

PARTE VI

REPORTAR EXCEPÇÕES

Erro de codificação:

```
int[] intArray = new int[5];
intArray[5] = 27;
```

Output na consola:

```
Exception in thread "main"
    java.lang.ArrayIndexOutOfBoundsException: 5
        at TestErrors.main(TestErrors.java:17)
```

REPORTAR EXCEPÇÕES

- Calling code in `main()`:

```
TestArray myTestArray = new TestArray(5);
myTestArray.addElement(5, 23);
```

Classe TestArray:

```
public class TestArray {
   int[] intArray;
   public TestArray (int size) {
      intArray = new int[size];
   }
   public void addElement(int index, int value) {
     intArray[index] = value;
   }
}
```

COMO AS EXCEPÇÕES SÃO LANÇADAS

- Execução normal de programa:
  1. Um método invoca o método que irá efetuar a tarefa
  2. O método que ia executar a tarefa efetua a tarefa
  3. O método completa a tarefa, e devolve o resultado da execução para o método que o invocou
- Quando uma excepção ocorre, a sequência muda:
  - A excepção é lançada e:
    - Um objecto Exception é passada para um método na secção catch no método atual **ou…**
    - A execução é devolvida para o método que foi invocou a instrução

TIPOS DE EXCEPÇÕES

- Três tipos principais do tipo Throwable:
  - **Erro**
    - Tipicamente erros externos irrecuperáveis
    - Do tipo **unchecked**
  - **RuntimeException**
    - Tipicamente erros de programação
    - Do tipo **unchecked**
  - **Exception**
    - Erro recuperável
    - Do tipo **checked** (têm de ser apanhados ou lançados)

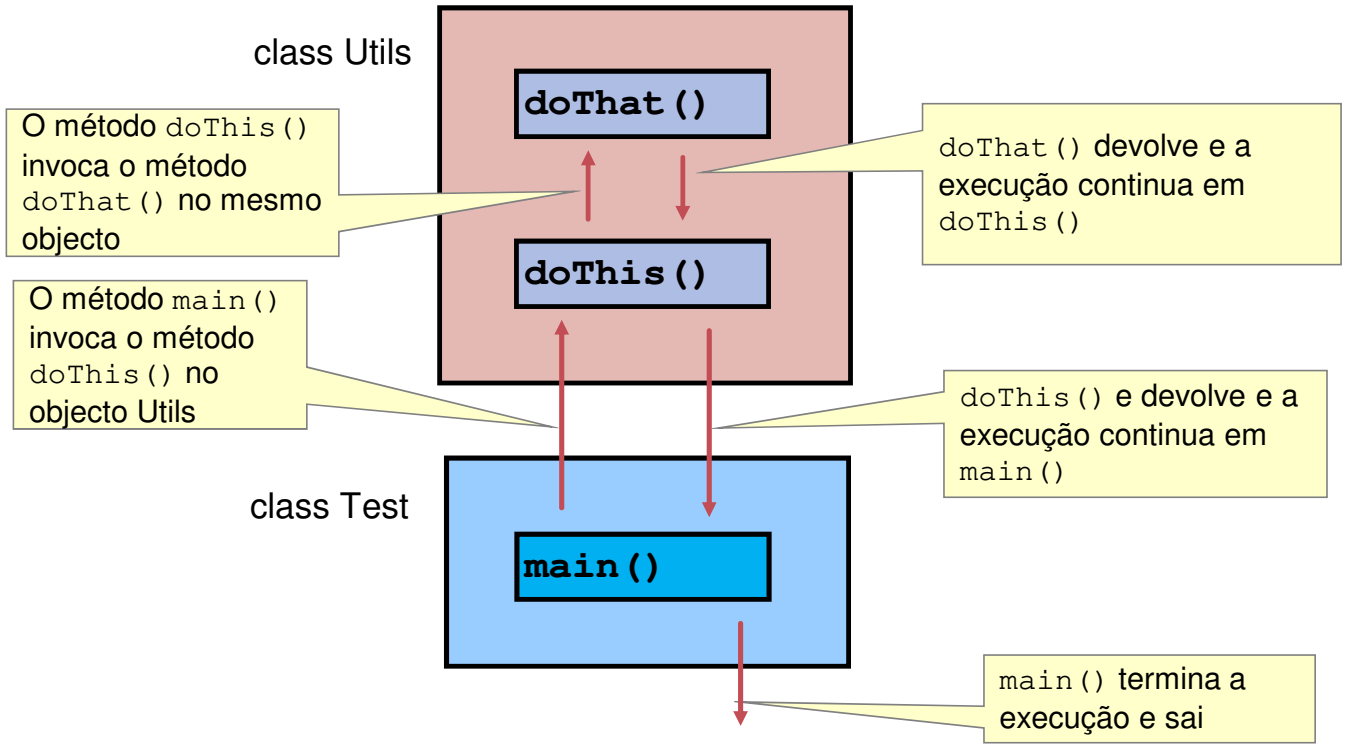
OUTOFMEMORY ERROR

Erro de programação:

```
ArrayList theList = new ArrayList();
while(true) {
   String theString = "A test String";
   theList.add(theString);
   if (theList.size()% 1000000 == 0) {
      System.out.println("List now has " +
            theList.size()/100000 + " million elements!");
   }
}
```

Output:

```
List now has 240 million elements!
List now has 250 million elements!
Exception in thread "main" java.lang.OutOfMemoryError: Java
    heap space
```



STACK DOS MÉTODOS

Classe Test:

```
public static void main (String args[]) {
   Utils theUtils = new Utils();
   theUtils.doThis();
}
```

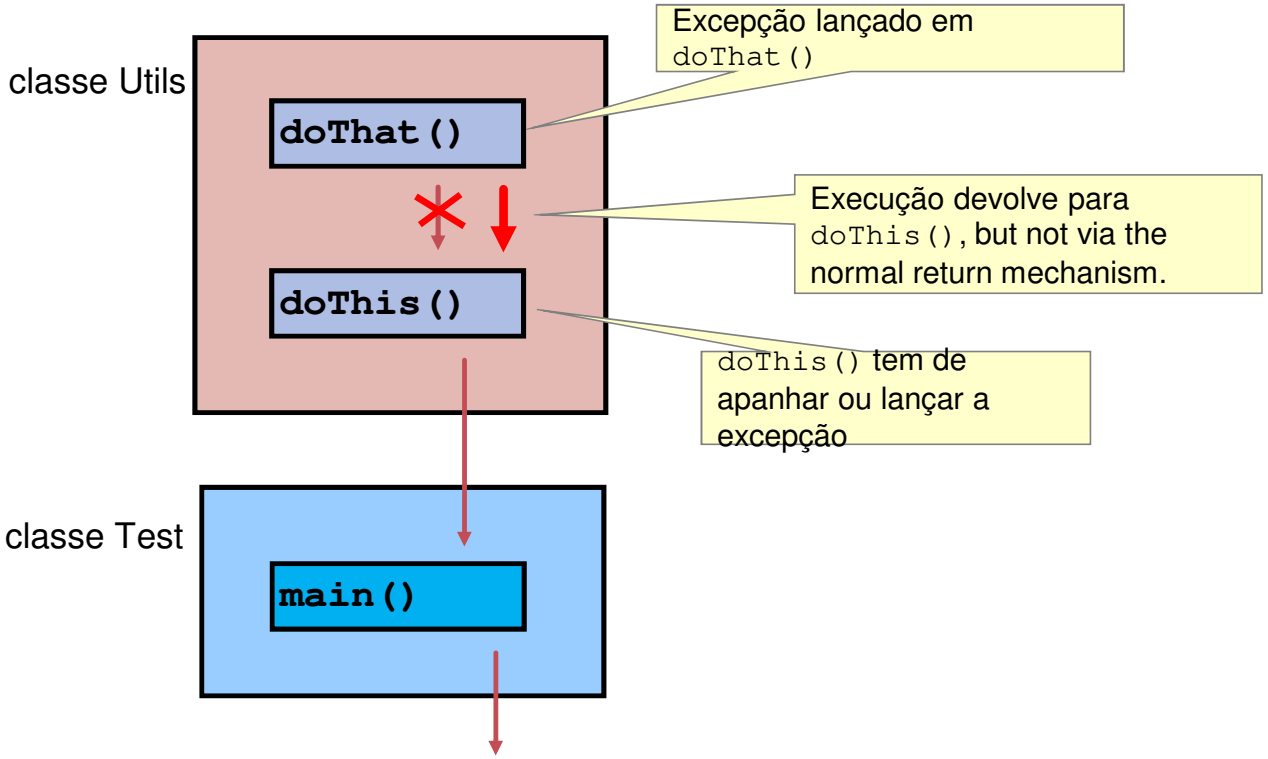
Classe Util:

```
  public void doThis() {
     ...< código para fazer alguma coisa >...
     doThat();
  return;

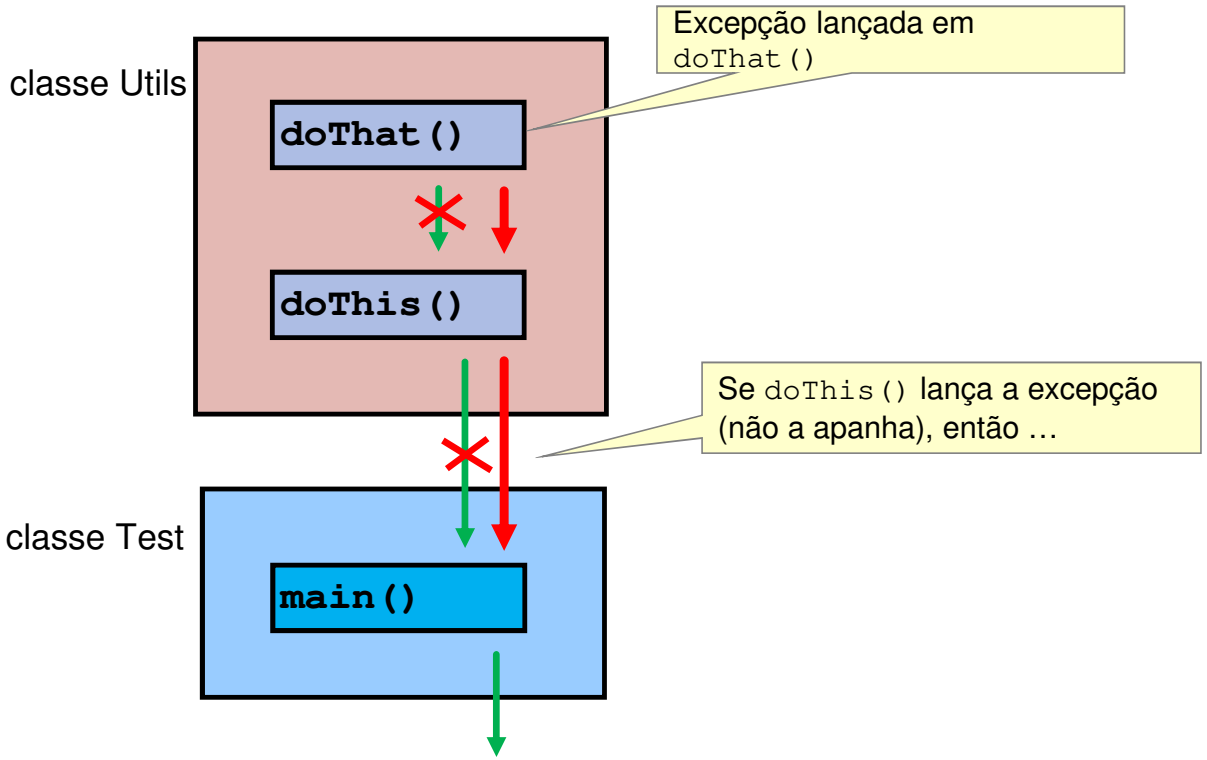
  public void doThat() throws Exception{
     ...< código para fazer alguma coisa >...
     if (algum_problema) throw new Exception();
  return;
```

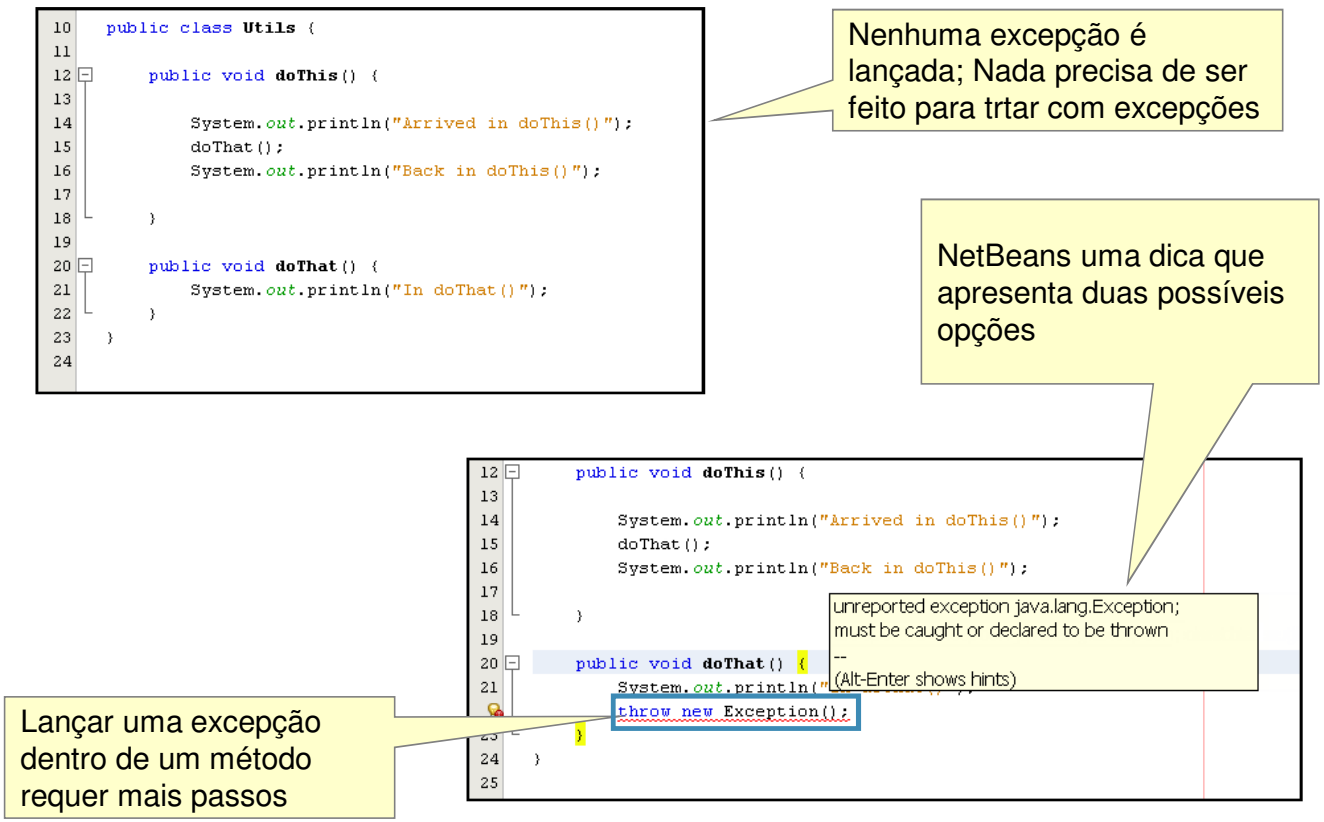
STACK DA INVOCAÇÃO:EXEMPLO

LANÇANDO THROWABLES

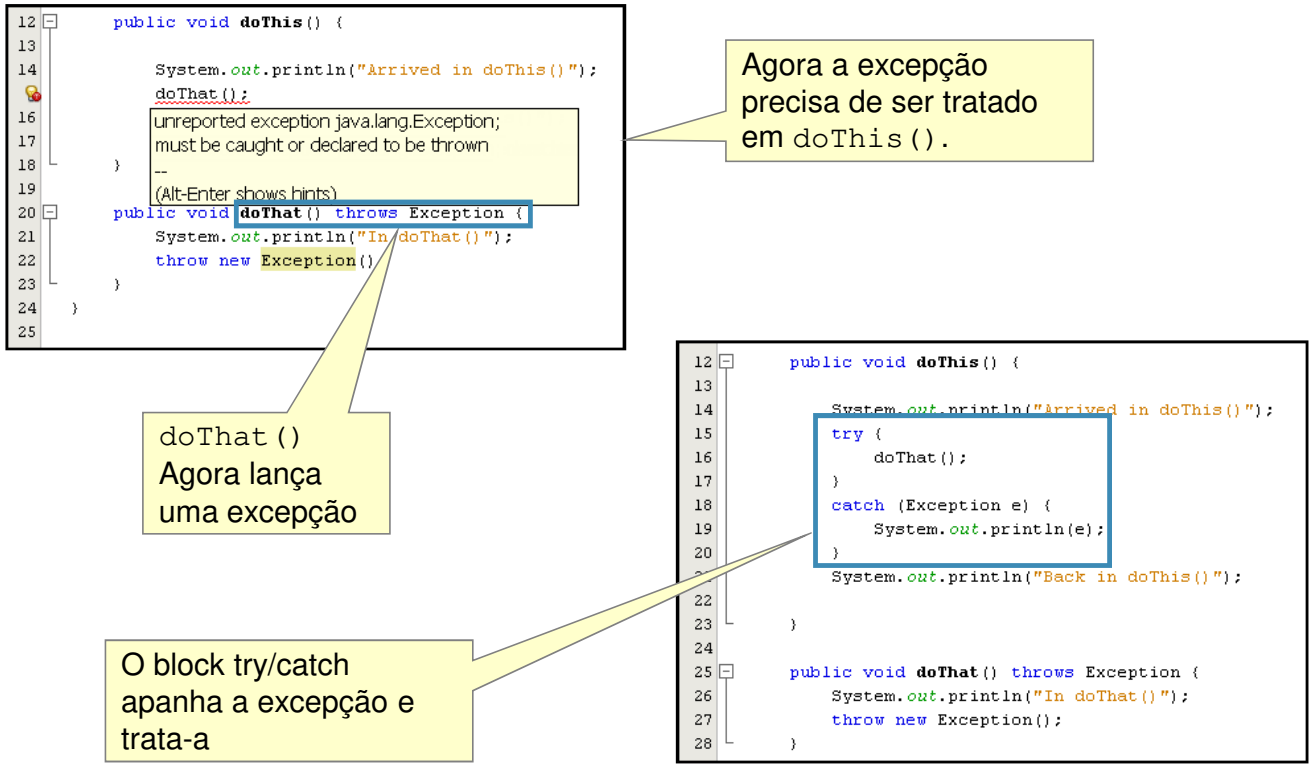


LANÇANDO THROWABLES



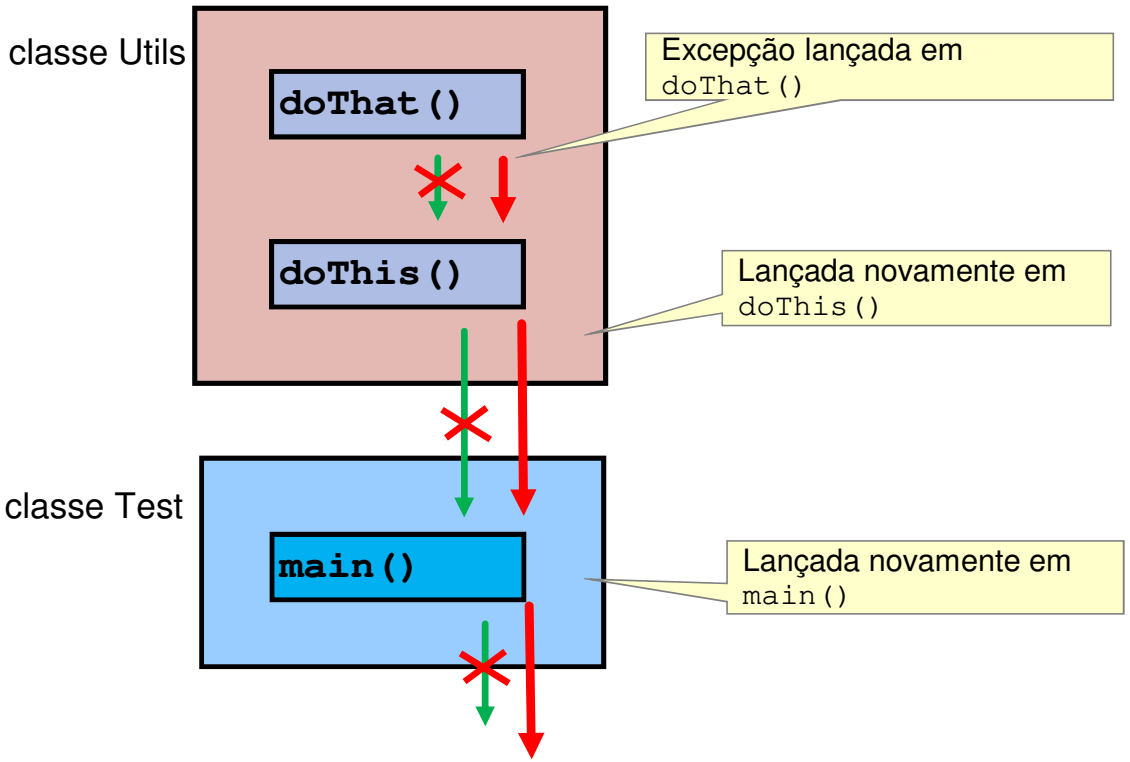


TRABALHAR COM EXCEPÇÕES NO NETBEANS



APANHANDO UMA EXCEPÇÃO

EXCEPÇÃO NÃO TRATADA



EXCEPÇÃO NA IMPRESSA NA CONSOLA

- Exemplo de `main()` a lançar uma excepção

```
10  public class Test {
11
12      public static void main (String args[]) throws Exception {
13
14          System.out.println("Started in main()");
15          Utils myUtils = new Utils();
16          myUtils.doThis();
17          System.out.println("Back in main()");
18      }
19
20  }
```

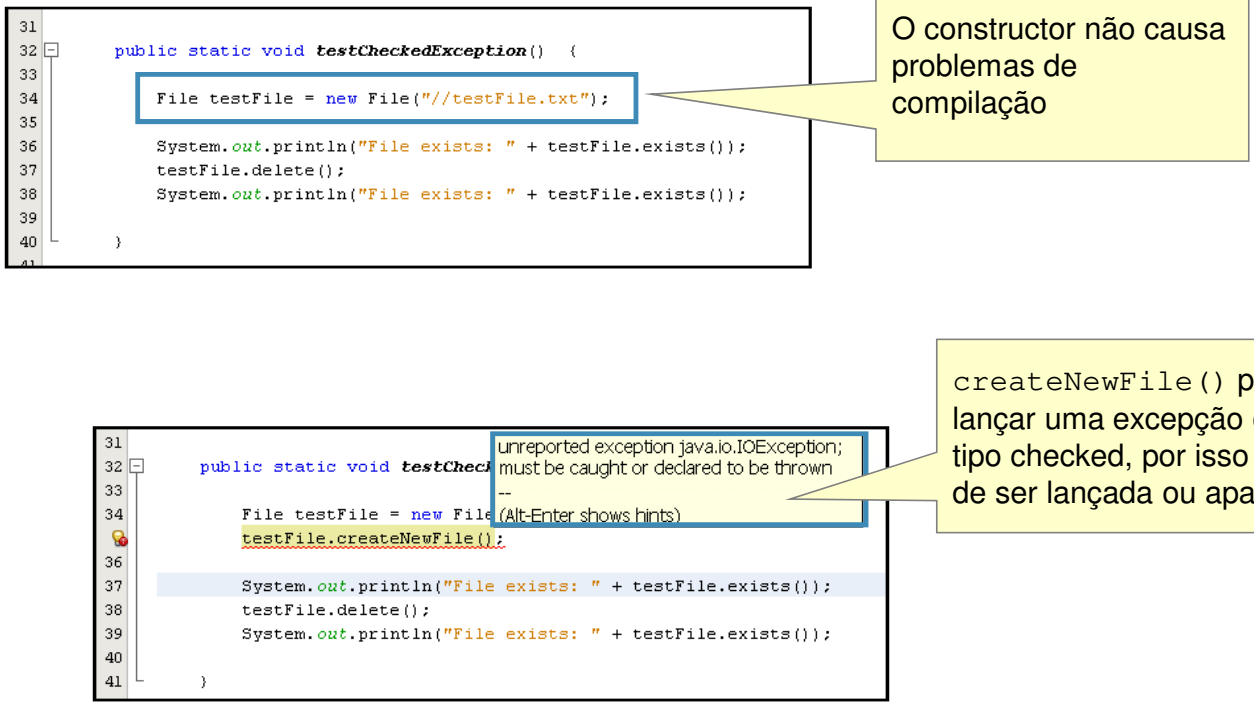
```
Output - TestCode (run)   Tasks
run:
Started in main()
Arrived in doThis()
In doThat()
Exception in thread "main" java.lang.Exception
        at Utils.doThat(Utils.java:27)
        at Utils.doThis(Utils.java:16)
        at Test.main(Test.java:16)
Java Result: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

`main()` está agora configurada para lançar a excepção

Porque `main()` lança a excepção, agora imprime a ordem de execução na consola

RESUMO DO TIPO DE EXCEPÇÕES

- **Throwable** é um tipo especial de objeto Java:
  - Apenas objetos deste tipo são usados como argumento na secção **catch**
  - Apenas objetos deste podem ser “lançados” para o **método invocador**
  - Tem duas subclasses:
    - **Error**
      - Lançado automaticamente para o método invocador, se criado
    - **Exception**
      - Tem de ser lançado explicitamente para o método invocador
        OU
      - Apanhado utilizando o bloco try/catch
      - Tem a subclasse RuntimeException que é automaticamente lançada no método invocador



INVOCAR UM MÉTODO QUE LANÇA UMA EXCEPÇÃO

Apanhar uma IOException:

```
public static void main(String args[]) {
   try {
      testCheckedException();
   }
   catch (IOException e) {
      System.out.println(e);
   }
 }

public static void testCheckedException() throws IOException{
   File testFile = new File("//testFile.txt");
   testFile.createNewFile();
   System.out.println("File exists: " + testFile.exists());

}
```

TRABALHAR COM EXCEPÇÕES DO TIPO CHECKED

BOAS PRÁTICAS

- Apanhar a excepção lançada atualmente, não a classe Exception ou Throwable

- Examinar a excepção para descobrir o problema concreto que está a acontecer para o tratamento ser o mais “limpo” possível

- Não é necessário apanhar todas as excepções:
  - Um erro de programação não é para ser tratado. É para ser corrigido!!
  - Perguntar a si próprio: “Esta excepção representa o comportamento que eu quero que o programa recupere?”

```
public static void main (String args[]) {
  try {
     createFile("c:/testFile.txt");
  }
  catch (Exception e) {

     System.out.println("Problem creating the file!");
     ...< other actions >...
  }
}

public static void createFile(String fileName) throws
   IOException {
     File f = new File(fileName);
     f.createNewFile();

     int[] intArray = new int[5];
     intArray[5] = 27;
  }
```

Apanhar a superclasse?

Sem processamento do objecto Exception?

MÁS PRÁTICAS

```
public static void main (String args[]) {
  try {
     createFile("c:/testFile.txt");
  }
  catch (Exception e) {
       System.out.println(e);
     ...< other actions >...
  }
}

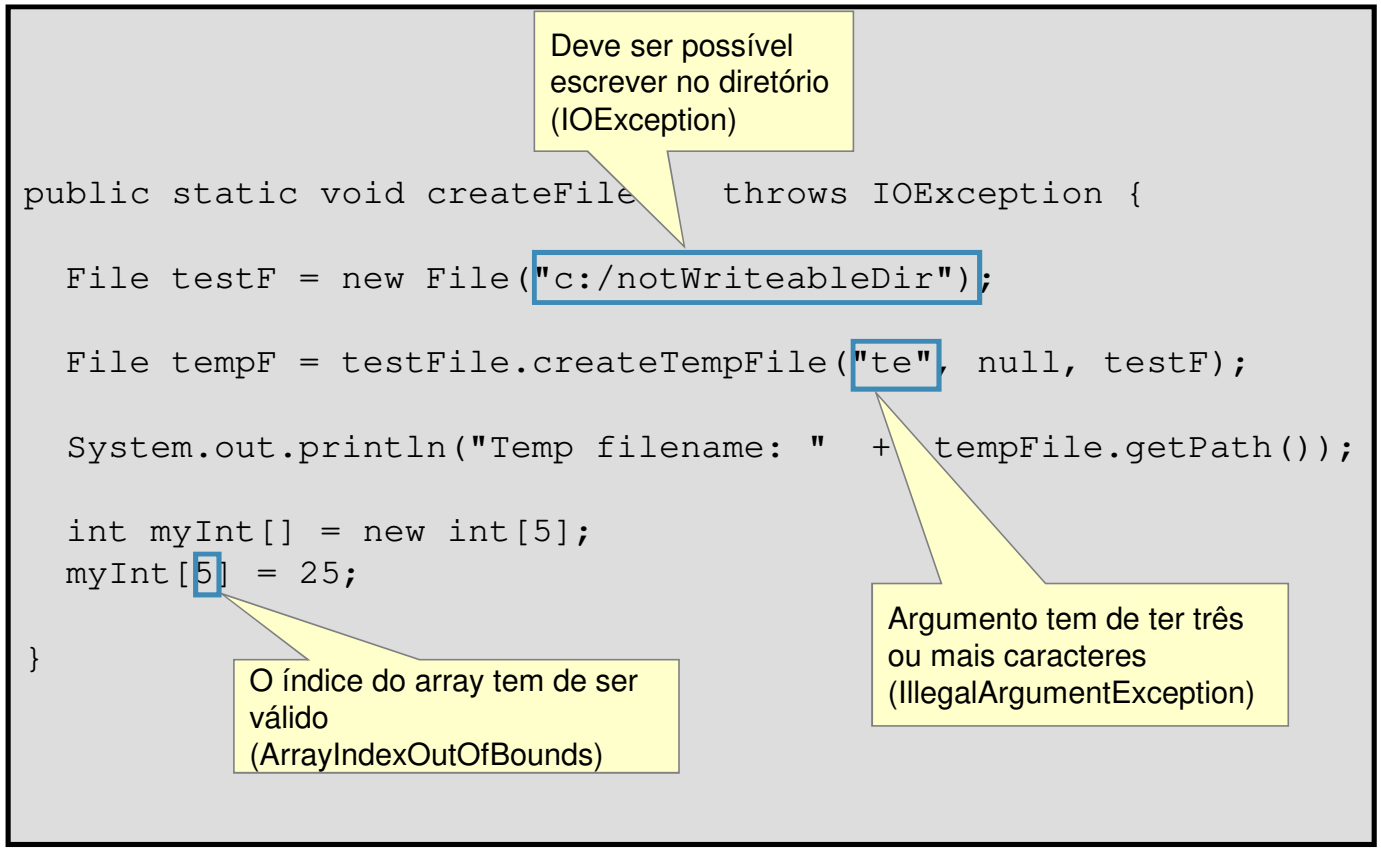
public static void createFile(String fileName) throws
   IOException {
     File f = new File(fileName);
     System.out.println(fileName + " exists? " + f.exists());
     f.createNewFile();
     System.out.println(fileName + " exists? " + f.exists());
     int[] intArray = new int[5];
     intArray[5] = 27;
  }
```

Qual é o tipo de objeto?

`toString()` é invocado neste objeto

MÁS PRÁTICAS

VÁRIAS EXCEPÇÕES



```
public static void main (String args[]) {
   try {
      createFile();
   }
   catch (IOException ioe) {
      System.out.println(ioe);
   }
}

public static void createFile()  throws IOException {

  File testF = new File("c:/notWriteableDir");
  File tempF = testFile.createTempFile("te", null, testF);
  System.out.println("Temp filename is " + tempFile.getPath());
  int myInt[] = new int[5];
  myInt[5] = 25;
}
```

APANHAR IOEXCEPTION

```
public static void main (String args[]) {
   try {
      createFile();
   }
   catch (IOException ioe) {
      System.out.println(ioe);
   } catch (IllegalArgumentException iae) {
      System.out.println(iae);
   }
}

public static void createFile()  throws IOException {

  File testF = new File("c:/writeableDir");
  File tempF = testFile.createTempFile("te", null, testF);
  System.out.println("Temp filename is " + tempFile.getPath());
  int myInt[] = new int[5];
  myInt[5] = 25;
}
```

APANHAR ILLEGALARGUMENTEXCEPTION

```
public static void main (String args[]) {
   try {
      createFile();
   }
   catch (IOException ioe) {
      System.out.println(ioe);
   } catch (IllegalArgumentException iae) {
      System.out.println(iae);
   } catch (Exception e) {
      System.out.println(e);
   }
}
public static void createFile()  throws IOException {
  File testF = new File("c:/writeableDir");
  File tempF = testFile.createTempFile("te", null, testF);
  System.out.println("Temp filename is " + tempFile.getPath());
  int myInt[] = new int[5];
  myInt[5] = 25;
}
```

APANHAR AS RESTANTES EXCEPÇÕES