Lição 9

Trabalhando com bibliotecas de classes

Objetivos

Ao final da lição, o estudante deverá estar apto a:

- Explicar o que é programação orientada a objetos e alguns de seus conceitos
- Diferenciar entre classes e objetos
- Diferenciar atributo e métodos de objeto de atributos e métodos de classe
- Explicar o que são métodos, como invocá-los e como enviar argumentos
- Identificar o escopo de um atributo
- Realizar conversões de tipos de dados primitivos e de objetos
- Comparar objetos e determinar as classes dos objetos

Introdução à POO

- POO ou Programação Orientada a Objeto
- Diz respeito a **objetos** como o elemento básico dos seus programas
- **Objetos** são caracterizados pelas suas propriedades e comportamentos

Introdução à POO

Objeto	*Propriedades*	*Comportamentos*
Carro	tipo de câmbio fabricante cor	virar frear acelerar
Leão	peso cor apetite temperamento	rugir dormir caçar

Encapsulamento

- Método de ocultar os elementos de uma classe de sua implementação
- Colocar uma proteção ao redor das propriedades e métodos dos objetos

Classes

- Pode ser pensada como um gabarito, um protótipo ou uma planta de um objeto
- É a estrutura fundamental em programação orientada a objetos
- Membros de classes:
  - propriedades (atributos)
  - Métodos

Objetos

- Composto por um conjunto de atributos (propriedades) que são as variáveis que descrevem as características essenciais do objeto e, consiste num conjunto de métodos (comportamentos) que descrevem como um objeto se comporta

- É a instância de uma classe

Classes e Objetos

Classe Carro		Objeto Carro A	Objeto Carro B
Atributos de Objeto	Número da placa	ABC 111	XYZ 123
	Cor	Azul	Vermelha
	Fabricante	Mitsubishi	Toyota
	Velocidade	50 km/h	100 km/h
Métodos de Objeto	Método Acelerar		
	Método Girar		
	Método Frear		

Classes e Objetos

- Classes fornecem o benefício da **reusabilidade**
- Programadores podem utilizar a mesma classe diversas vezes para criar os objetos

Conteúdo da Classe

- Atributos de Objeto
- Métodos de Objeto

- Atributos de Classe (atributos estáticos)
- Métodos de Classe (métodos estáticas)

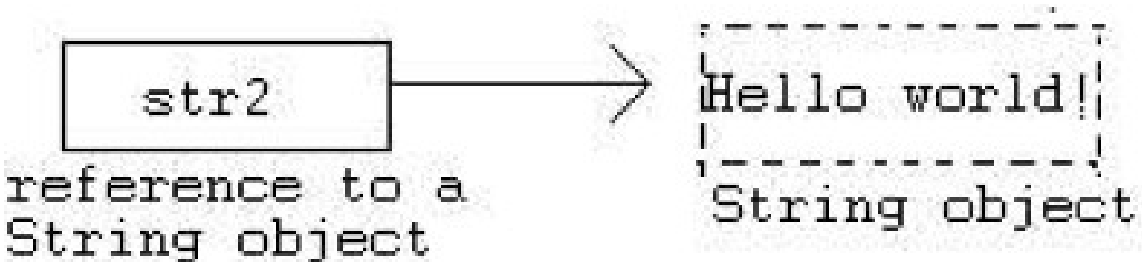
Instância de Classe

- Para criar um objeto, ou uma instância de uma classe, utilizamos o operador **new**
- Para criar um objeto da classe String, escreve-se o seguinte código:

```
String str2 = new String("Hello world!");
```

ou:

```
String str2 = "Hello world!";
```



Instância de Classe

- Operador **new**
  - Aloca memória para um objeto e retorna uma referência
  - Ao criar um objeto, invoca-se o construtor da classe
- Método construtor
  - Método onde são colocadas todas as inicializações
  - Possui o mesmo nome da classe

Métodos

- É uma parte separada do código que pode ser chamada pelo programa principal
- Pode ou não retornar valor
- Pode aceitar tantos argumentos quantos forem necessários
- Após o fim da execução de um método, o fluxo de controle é retornado a quem o chamou

Porque usar Métodos?

- Decomposição – É a chave para a solução de problemas
- A criação de métodos resolve uma parte específica do problema
- Separar o problema em partes menores e manuseáveis

Chamando Atributos

- Para ilustrar como chamar métodos, vamos usar a classe String como exemplo
- Pode-se usar a documentação das APIs Java para conhecer os métodos disponíveis na classe String

```
nomeDoObjeto.nomeDoMetodo(argumentos);
```

Chamando Atributos

Declaração do método	Definição
public char charAt(int indice)	Retornar o caractere especificado no índice. Um índice vai de 0 até length() - 1. O primeiro caractere da seqüência está no índice 0, o seguinte, no índice 1, e assim sucessivamente por todo o array.
public boolean equalsIgnoreCase (String outraString)	Comparar o conteúdo de duas Strings, ignorando maiúsculas e minúsculas. Duas strings são consideradas iguais quando elas têm o mesmo tamanho e os caracteres das duas strings são iguais, sem considerar caixa alta ou baixa.

Chamando Atributos

```
String str1 = "Hello";

char x = str1.charAt(0);

String str2 = "hello";

boolean result = str1.equalsIgnoreCase(str2);
```

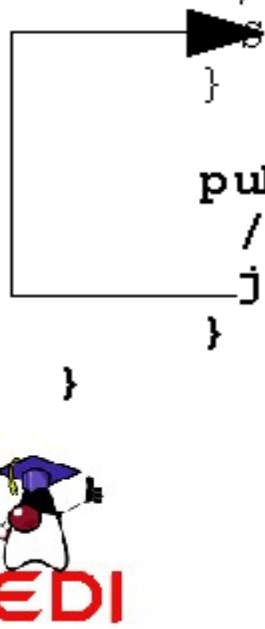
Envio de Argumentos

- **Envio por valor** – o método faz uma cópia do valor do argumento passado para o método
- **Envio por referência** – o método faz uma cópia da referência do argumento passado

Envio por valor

```
public class TestPassByValue
{
     public static void main( String[] args ){
      int i = 10;
      //print the value of i
      System.out.println( i );

      //call method test
      //and pass i to method test
      test( i );

      //print the value of i. i not changed
      System.out.println( i );
     }

     public static void test( int j ){
      //change value of parameter j
      j = 33;
     }
}
```

Pass i as parameter which is copied to j

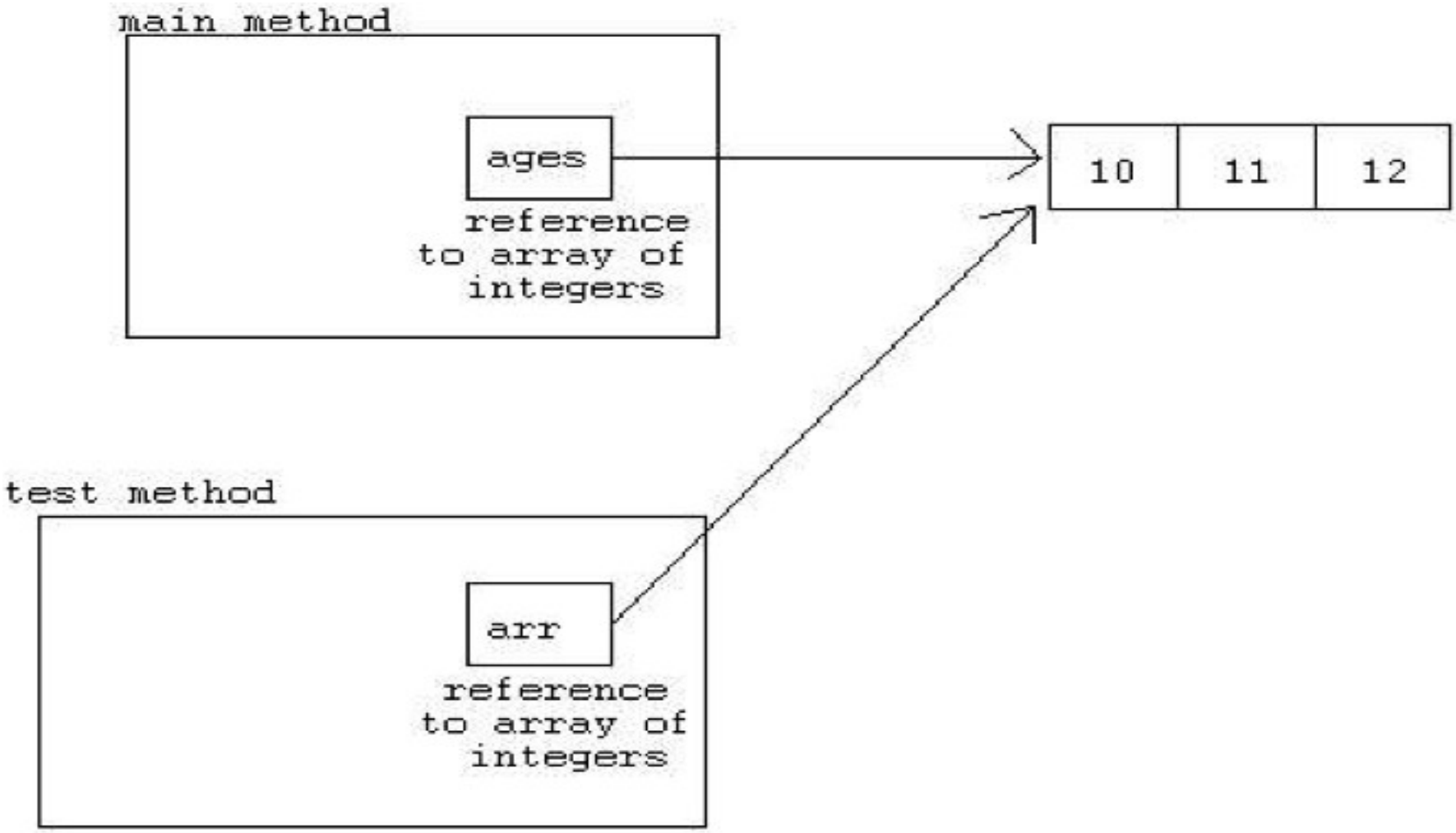


Envio por Referência

```
class TestPassByReference
{
      public static void main( String[] args ){
            //create an array of integers
            int []ages  = {10, 11, 12};

            //print array values
            for( int i=0; i<ages.length; i++ ){
                  System.out.println( ages[i] );
            }

            //call test and pass reference to array
            test( ages );

            //print array values again
            for( int i=0; i<ages.length; i++ ){
                  System.out.println( ages[i] );
            }
      }

      public static void test( int[] arr ){
            //change values of array
            for( int i=0; i<arr.length; i++ ){
                  arr[i] = i + 50;
            }
      }
}
```

Pass ages as parameter which is copied to variable arr

Envio por Referência



Chamando Métodos Estáticos

- Métodos que podem ser invocados sem que um objeto tenha sido instanciado pela classe (sem invocar a palavra chave new)
- Pertencem à classe como um todo e não a uma instância (ou objeto) específica da classe
- São diferenciados dos métodos de instância pela declaração da palavra chave **static** na definição do método

```
NomeClasse.nomeMetodoEstatico(argumentos);
```

Chamando Métodos Estáticos

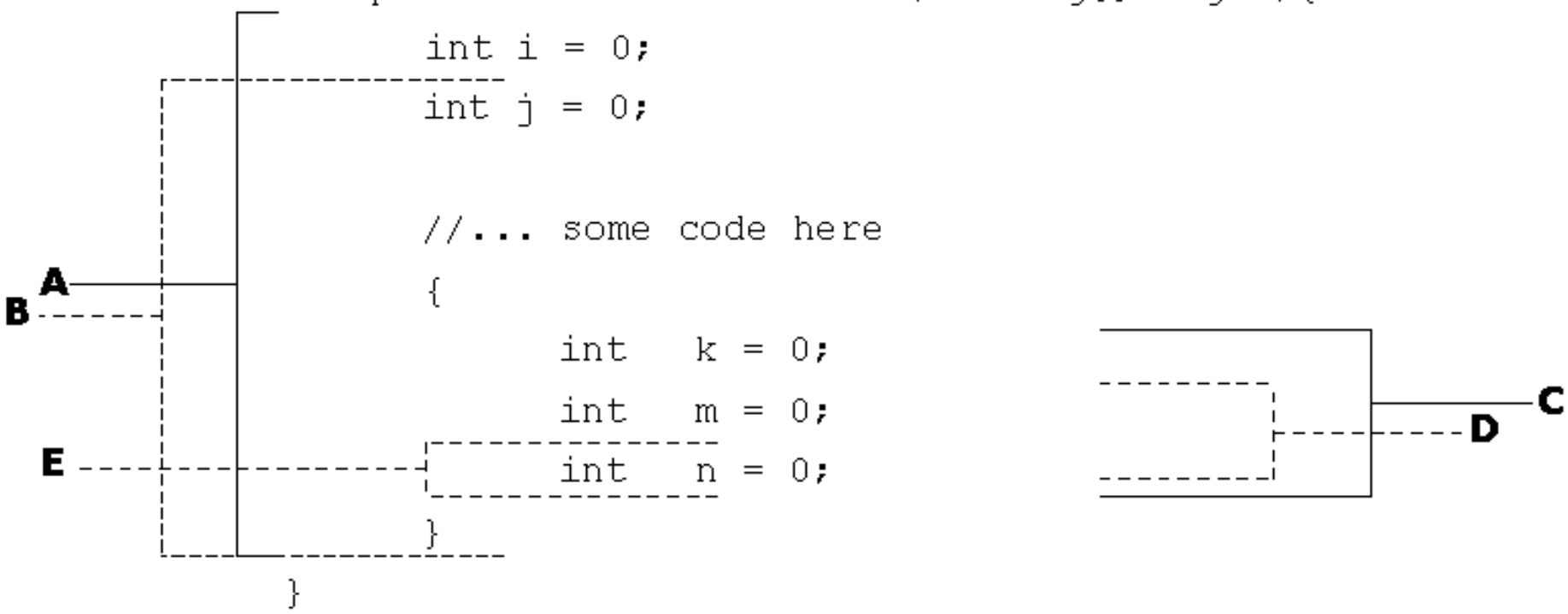
```
int i = Integer.parseInt("10");

String hexEquivalent = Integer.toHexString(10);
```

Escopo de um Atributo

- Determina onde o atributo é acessível na classe
- Determina o tempo de vida de um atributo ou quanto tempo o atributo pode existir na memória
- É determinado pelo local onde o atributo é declarado na classe
- Para simplificar, pensemos no escopo como sendo algo existente entre as chaves {...}. A chave à direita é chamada de chave de saída do bloco (**outer**) e a chave à esquerda é chamada chave de entrada do bloco (**inner**)
- Quando declarado dentro de um bloco, seu escopo inicia com a sua declaração e vai até a chave de saída do bloco

Exemplo 1

```
public class ScopeExample
{
    public static void main( String[] args ){
        int i = 0;
        int j = 0;

        //... some code here
        {
            int   k = 0;
            int   m = 0;
            int   n = 0;
        }
    }
```



Exemplo 2

```
class TestPassByReference
{
      public static void main( String[] args ){
          //create an array of integers
          int []ages          = {10, 11, 12};

          //print array values
          for( int i=0; i<ages.length; i++ ){
                    System.out.println( ages[i] );
          }

          //call test and pass reference to array
          test( ages );

          //print array values again
          for( int i=0; i<ages.length; i++ ){
                    System.out.println( ages[i] );
          }
      }

      public static void test( int[] arr ){
          //change values of array
          for( int i=0; i<arr.length; i++ ){
                    arr[i] = i + 50;
          }
      }
}
```

A

B

C

D

E

Escopo de um atributo

- O identificador deve ser único no escopo

- Isto significa que a seguinte declaração:

```
{
        int test = 10;
        int test = 20;
}
```

O compilador irá gerar um **erro** pois você deverá ter um nome único para o atributo dentro do bloco

Escopo de um atributo

- Pode-se ter atributos definidos com o mesmo nome, desde que não sejam declarados no mesmo bloco

```
public class TestBlock {
    int test = 10;
    public void test() {
        System.out.print(test);
        int test = 20;
        System.out.print(test);
    }
    public static void main(String[] args) {
        TestBlock testBlock = new TestBlock();
        testBlock.test();
    }
}
```

Casting de Tipos

- Casting de Tipos
- Casting de objetos

Casting de Tipos Primitivos

- Casting permite a conversão de um tipo para outro
- É usual entre os tipos numéricos
- Tipos lógicos não aceitam **casting**
- Casting pode ser **implícito** ou **explícito**

Casting Implícito

- Armazenar um atributo de tipo **menor** em um de tipo **maior:**

```
int numInt = 10;
double numDouble = numInt; // cast implícito
```

- Pela execução de fórmulas:

```
byte numInt1 = 1;
short numInt2 = 2;
int numDouble = numInt1 + numInt2; // resulta em int
```

Casting Explícito

- Converter um atributo cujo tipo tem um comprimento maior para um de comprimento menor, é necessário um **cast explícito**

```
(tipoDado)valor
```

Exemplos de Casting Explícito

Para realizar um **casting** de um valor do tipo **int** para **char**

```
char valChar = 'A';
System.out.print((int)valChar); // explícito
```

Exemplos de Casting Explícito

```
double   valDouble = 10.12;
int       valInt = (int)valDouble;

//converte valDouble para o tipo int
double x = 10.2;
int y = 2;

int result = (int)(x/y); //faz o cast do resultado da operação
                         //para int
```

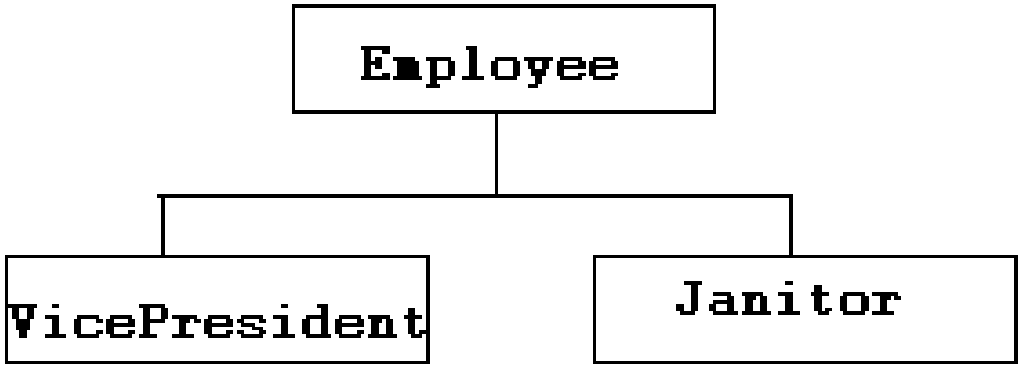
Casting de Objetos

- Em instâncias de classes a operação de casting também pode ser usada para fazer a conversão para instâncias de outras classes: **A classe de origem e a classe de destino devem ser da mesma família, relacionadas por herança; uma classe deve ser subclasse da outra**

- Analogamente à conversão de valores primitivos para um tipo maior, alguns objetos não podem ser convertidos implicitamente

```
(NomeClasse)objeto
```

Exemplo de Casting de Objetos




```
Employee emp = new Employee();
VicePresident veep = new VicePresident();
emp = veep;  // cast não é necessário de baixo para cima
veep = (VicePresident)emp; //necessita do cast explícito
```

Convertendo Tipos Primitivos para Objetos e vice-versa

- Não se pode fazer, sob qualquer circunstância, **casting** de um objeto para um tipo de dado primitivo ou vice-versa
- Como alternativa, o pacote **java.lang** inclui classes que fazem a correspondência para cada tipo primitivo: Float, Boolean, Byte, dentre outros
- Chamamos estas classes de **Wrapper Class**

Wrapper Class

- Possuem o mesmo nome dos tipos primitivos, exceto pelo fato que o nome dessas classe começam com letra maiúscula (Short ao invés de short, Double ao invés de double, etc)
- Duas possuem nomes que diferem do correspondente tipo de dados: **Character** é usado para variáveis **char** e **Integer** para variáveis **int**
- Usando as classes que correspondem a cada um dos tipos primitivos, é possível criar objetos que armazenam o mesmo tipo de valor

Convertendo Tipos Primitivos para Objetos e vice-versa

- Criar uma instância de uma classe Integer com o valor 7801 (primitivo para Objeto)

```
Integer dataCount = new Integer(7801);
```

- Converter um objeto Integer para um tipo de dado primitivo int. O resultado é o valor 7801

```
int newCount = dataCount.intValue();
```

- Converter String para um tipo numérico, digamos int (objeto para primitivo)

```
String pennsylvania = "65000";

int penn = Integer.parseInt(pennsylvania);
```

Comparando Objetos

- Operadores de comparação de valores trabalham com tipos primitivos e não com objetos
- A exceção são os operadores: == (igualdade) e != (diferente)
- Quando aplicados a objetos, estes operadores verificam se dois objetos possuem ou não a mesma referência

Comparando Objetos

```
class EqualsTest {
    public static void main(String[] arguments) {
      String str1, str2;
      str1 = "Free the bound periodicals.";
      str2 = str1;
      System.out.println("String1: " + str1);
      System.out.println("String2: " + str2);
      System.out.println("Same object? " + (str1 == str2));
      str2 = new String(str1);
      System.out.println("String1: " + str1);
      System.out.println("String2: " + str2);
      System.out.println("Same object? " + (str1 == str2));
      System.out.println("Same value? " + str1.equals(str2));
    }
}
```

Determinando a classe de um objeto

- É possível descobrir a qual classe pertence um objeto
- Suponhamos o seguinte objeto:

```
QualquerClasse obj = new QualquerClasse();
```

Método getClass()

- **getClass()** retorna a classe do objeto (onde Class é a classe em si) que possui um método chamado **getName()**

- **getName()** retorna o nome da classe

```
String name = key.getClass().getName();
```

Operador instanceof

- O instanceof tem dois operadores: a referência para o objeto à esquerda e o nome da classe à direita
- A expressão retorna um lógico cujo valor depende se o objeto é uma instância da classe declarada ou qualquer uma de suas subclasses

```
String ex1 = "Texas";
System.out.println(ex1 instanceof String);
String ex2;
System.out.println(ex2 instanceof String);
```

Sumário

- Classes e objetos
  - Atributos de objeto
  - Atributos de classe
- Instância de classe
- Métodos
  - Métodos de objeto
  - Enviando atributos aos métodos (por valor e por referência)
  - Métodos estáticos
  - Escopo de um atributo
- Casting (tipo primitivo e objeto)
- Convertendo tipos primitivos para objetos e vice-versa
- Comparando objetos
- Determinando a classe de um objeto

Parceiros

- **Os seguintes parceiros tornaram JEDI possível em Língua Portuguesa:**