Java – Aula 16
Tipos Genéricos

Cristiano Amaral Maffort
cristiano@cefetmg.br
maffort@gmail.com

Técnico em Informática
Departamento de Computação
CEFET-MG – Belo Horizonte

Introdução

- Muitos algoritmos são logicamente iguais
  - Não importando o ***tipo de dado*** ao qual eles estão sendo aplicados
    - Ex: algoritmos de ordenação, estruturas de dados (pilha, lista, fila) etc
- Tipos genéricos são também chamados de tipos parametrizados
- Os tipos genéricos ***permitem criar classes, interfaces e métodos*** em que
  - ***o tipo de dado com o qual operam é especificado como parâmetro***.
- Tipos genéricos permitem generalizar algoritmos
  - De modo que eles sejam definidos uma única vez
    - Independentemente do tipo de dado
  - E ***aplicá-los a uma ampla variedade de tipos de dados***

Pilha Básica

```java
public class ArrayStack {

    int itens[], topo;

    public ArrayStack(int n) {
        itens = new int[n];
        topo = 0;
    }

    public void empilhar(int item) {
        itens[topo++] = item;
    }

    public int desempilhar() {
        return itens[--topo];
    }

    public int tamanho() {
        return topo;
    }
}
```

Pilha de inteiros.

E se forem necessárias pilhas de ***outros tipos de dados?***

Pilha de Object

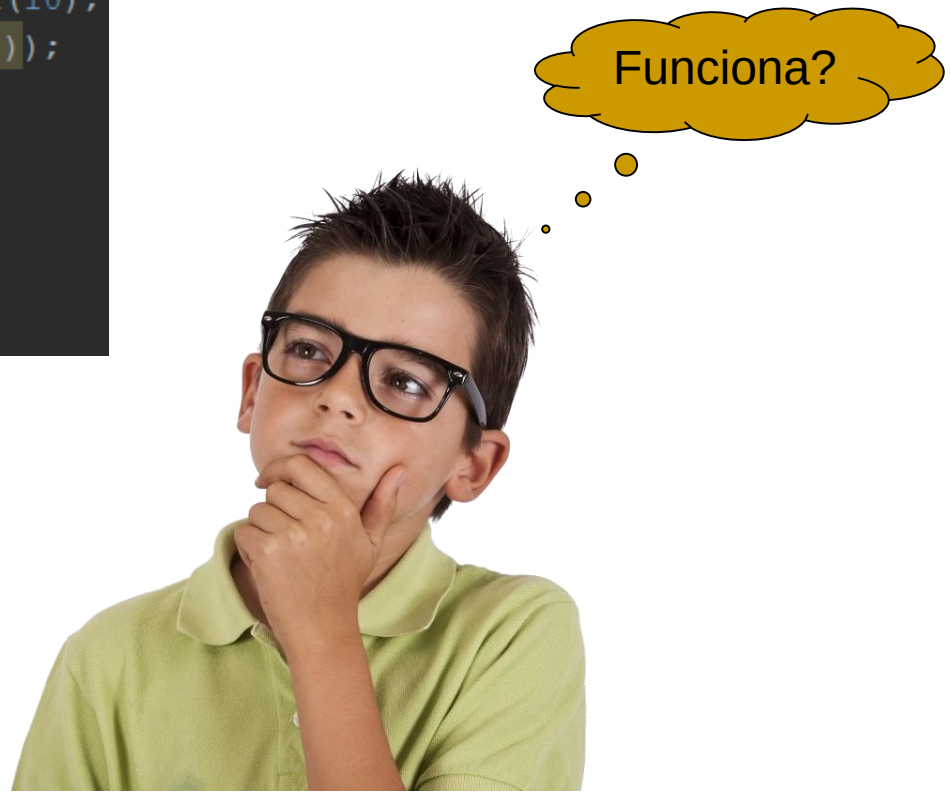
```java
public class ArrayStackGeneral {

    Object itens[];
    int topo;

    public ArrayStackGeneral(int n) {
        itens = new Object[n];
        topo = 0;
    }

    public void empilhar(Object item) {
        itens[topo++] = item;
    }

    public Object desempilhar() {
        return itens[--topo];
    }

    public int tamanho() {
        return topo;
    }
}
```

Pilha do tipo mais geral de Java (Object).

Toda classe Java herda Object. Então permite reuso do algoritmo. ***Mas é seguro?***

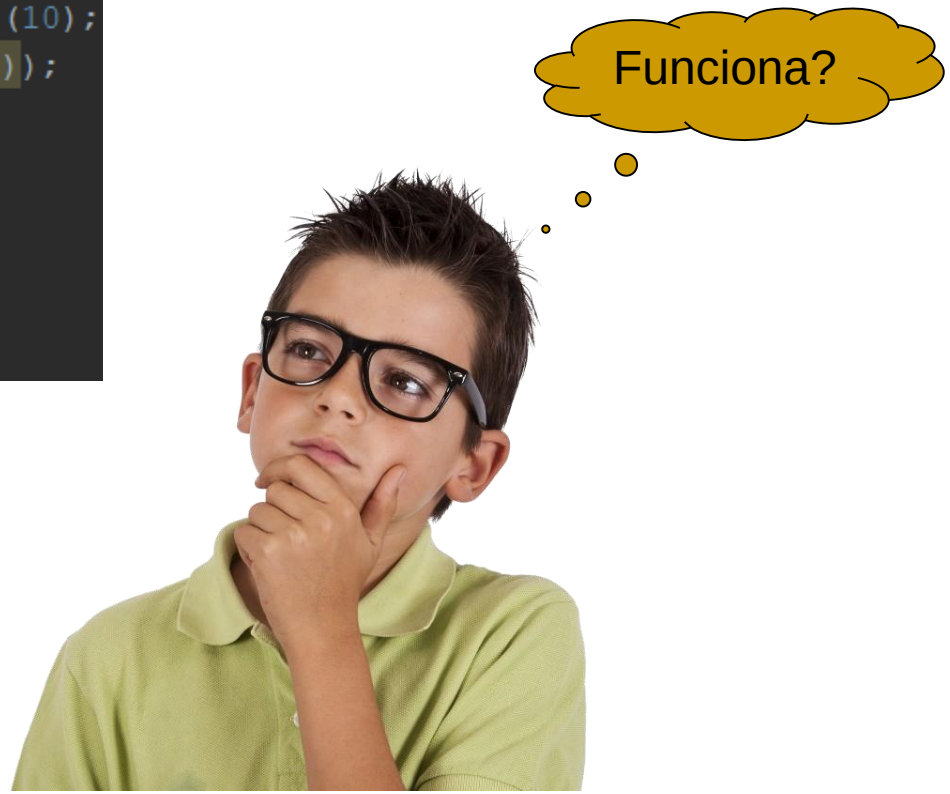
Pilha de Object

```
public static void main(String[] args) {
    ArrayStackGeneral pilha = new ArrayStackGeneral(10);
    pilha.empilhar(new String("José da Silva Sauro"));
    pilha.empilhar(Integer.valueOf(500));
    pilha.empilhar(Double.valueOf(3.1415));

    for (int i = 0; pilha.tamanho() != 0; i++)
        System.out.println(pilha.desempilhar());
}
```



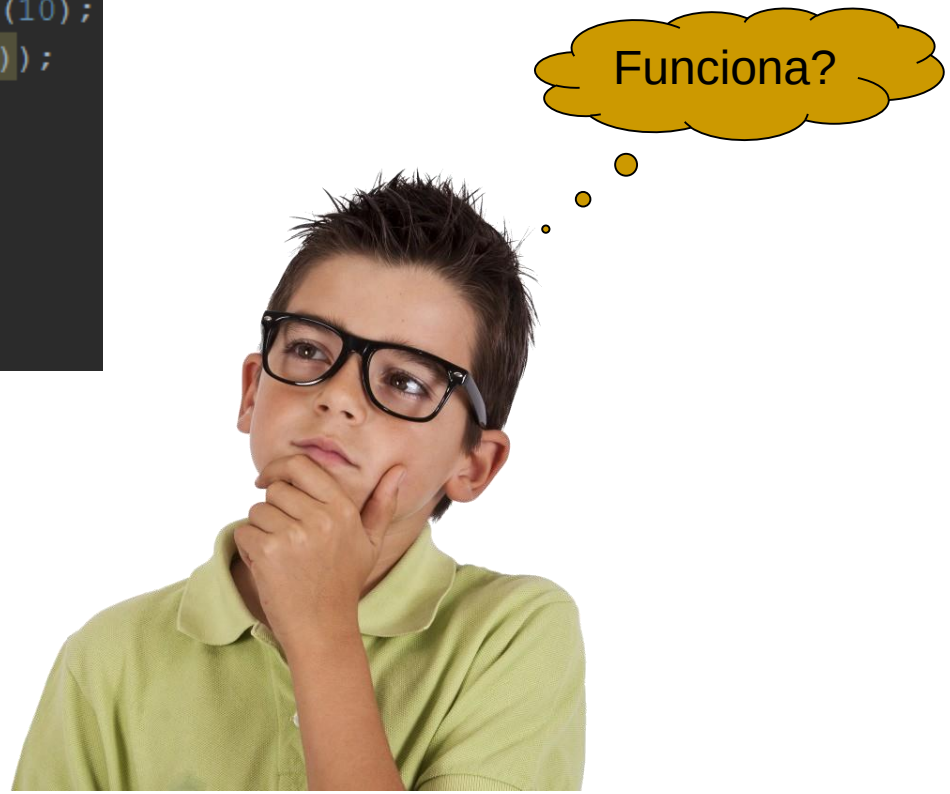
Pilha de Object

```
public static void main(String[] args) {
    ArrayStackGeneral pilha = new ArrayStackGeneral(10);
    pilha.empilhar(new String("José da Silva Sauro"));
    pilha.empilhar(Integer.valueOf(500));
    pilha.empilhar(Double.valueOf(3.1415));

    Double d = (Double) pilha.desempilhar();
    System.out.println(d);
}
```



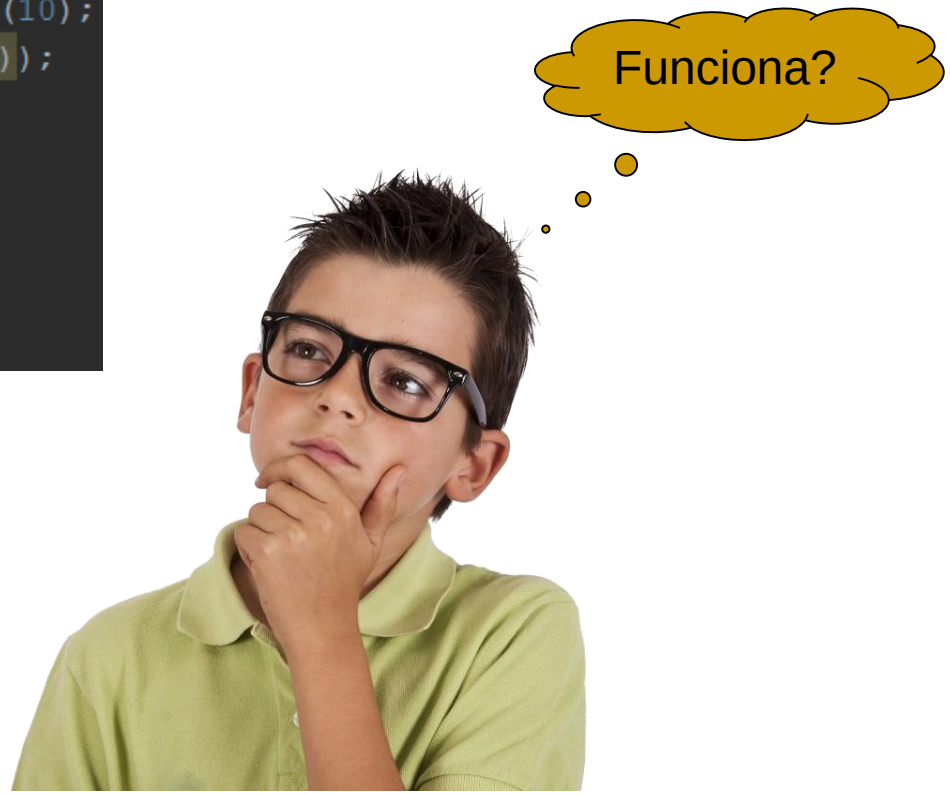
Pilha de Object

```
public static void main(String[] args) {
    ArrayStackGeneral pilha = new ArrayStackGeneral(10);
    pilha.empilhar(new String("José da Silva Sauro"));
    pilha.empilhar(Integer.valueOf(500));
    pilha.empilhar(Double.valueOf(3.1415));

    Integer i = (Integer) pilha.desempilhar();
    System.out.println(i);
}
```



Pilha de Object

```
public static void main(String[] args) {
    ArrayStackGeneral pilha = new ArrayStackGeneral(10);
    pilha.empilhar(new String("José da Silva Sauro"));
    pilha.empilhar(Integer.valueOf(500));
    pilha.empilhar(Double.valueOf(3.1415));

    Integer i = (Integer) pilha.desempilhar();
    System.out.println(i);
}
```




```
Exception in thread "main" java.lang.ClassCastException:
class java.lang.Double cannot be cast to class java.lang.Integer
```

Pilha de Object

Nesse caso, para efetivamente utilizar o dado/objeto, é necessário realizar coerção de tipo.

```
public static void main(String[] args) {
    ArrayStackGeneral pilha = new ArrayStackGen
    pilha.empilhar(new String("José da Si
    pilha.empilhar(Integer.valueOf(
    pilha.empilhar(Double.val
    Double d = (Double) pilha.desempilhar();
    System.out.println(d);
}
```

Dilema

- O tipo Object permite criar algoritmos reutilizáveis
  - Mas requer que o usuário do algoritmo realize coerções explicitas quando precisar recuperar dados e acessar a interface do objeto.
  - Coerções incompatíveis somente são identificadas em tempo de execução.
- Tipos específicos/especializados comprometem o reuso do algoritmo
  - Dispensa o usuário do algoritmo de realizar coerções de tipos.
  - São mais seguros, porque o compilador será capaz de verificar a compatibilidade dos tipos em tempo de compilação.
- São mais eficientes?
- Como resolver?

Pilha Genérica

```java
public class ArrayStackGeneric<T> {

    T itens[];
    int topo;

    public ArrayStackGeneric(int n) {
        itens = (T[]) new Object[n];
        topo = 0;
    }

    public void empilhar(T item) {
        itens[topo++] = item;
    }

    public T desempilhar() {
        return itens[--topo];
    }

    public int tamanho() {
        return topo;
    }
}
```

T é o parâmetro de tipo genérico.

Java não reconhece construtor genérico. Por isso foi necessário a coerção aqui.

Pilha Genérica

```
public static void main(String[] args) {
    ArrayStackGeneric<Integer> pilha = new ArrayStackGeneric<>(10);
    pilha.empilhar(new String("José da Silva Sauro"));
    pilha.empilhar(Integer.valueOf(500));
    pilha.empilhar(Double.valueOf(3.1415));

    Integer i = pilha.desempilhar();
    System.out.println(i);
}
```

Incompatibilidades detectadas em tempo de compilação.

String não é compatível com Integer

Double não é compatível com Integer

Não é necessário fazer coerção de tipos.

Tipos genéricos

- Forma geral:
  - `class nome-classe <lista-parâmetros-tipo> {...}`
- Tipos genéricos somente funcionam com objetos
- Tipos genéricos diferem de acordo com seus argumentos de tipo
  - Ex:
    - `Stack<Integer> pilhaI = new Stack<>();`
    - `Stack<Double> pilhaD = new Stack<>();`
    - `pilhaI = pilhaD; // erro!`
- Tipos genéricos admitem declarar mais de um parâmetro de tipo
  - Ex:
    - `class TwoGenParam<T, V> { ... }`

Limitação nos tipos

- Dado o algoritmo genérico, algumas vezes é necessário limitar o tipo do dado passado para o tipo genérico criado
  - Ex:
    - `class NumericFns<T extends Number> { ... }`
    - `class TwoGenComp<T, C extends Comparable> { ... }`
      - Obs: Comparable é uma interface
- Algumas vezes é necessário realizar limitações nos métodos
  - `void test(Gen<? extends  A> obj) { ... }`
  - O método `test` aceitará qualquer objeto cujo tipo genérico seja `A` ou subtipo de `A`.

Erasure

- Tipos genéricos foram criados apenas na versão 1.5 de Java.
- No projeto dos tipos genéricos procuraram compatibilizar com versões anteriores da linguagem.
  - Ou seja, o código genérico tinha que ser compatível com códigos não genéricos preexistentes.
- *Erasure* foi a maneira como Java implementou essa compatibilização.
  - Para isso, o compilador Java remove todas as informações de tipos genéricos (*erased*), substituindo os parâmetros de tipo pelos por Object, ou pelo tipo especificado nas limitações.
    - Assim, o compilador impõe a compatibilidade de tipos.
- A JVM não processa/executa tipos genéricos
  - *Generics* não existem em tempo de execução

Bibliografia Obrigatória

- HORSTMANN, Cay S.; CORNELL, Gary. ***Core Java: Fundamentos***. 8. ed. São Paulo: Pearson Prentice Hall, 2020, p. 297 – 309.
- SCHILDT, Herbert; SKRIEN, Dale. ***Programação com Java: uma introdução abrangente***. Porto Alegre: AMGH, 2013, p. 519-557.