

Computação Orientada a Objetos

Genéricos

Slides baseados em:

- Deitel, H.M.; Deitel P.J. **Java: Como Programar**, Pearson Prentice Hall, 6a Edição, 2005. **Capítulo 18**
- Slides Profa. Patrícia R. Oliveira

Profa. Karina Valdivia Delgado
EACH-USP

Introdução

O que fazer para escrever um único método de ordenação **ordena** para elementos em um array de **Integer**, em um array de **String** ou em um array de qualquer tipo que suporte ordenamento?

Introdução

O que fazer para escrever uma única classe **Stack** que seria utilizada como uma pilha de inteiros, uma pilha de números de ponto flutuante, uma pilha de **String** ou uma pilha de qualquer outro tipo?

Introdução

- **Genéricos:** recurso que fornece um meio de criar os objetos gerais citados nas Questões 1 e 2.
- **Classes genéricas:** permite que o programador defina, com uma única declaração de classe, um conjunto de tipos relacionados.
- **Métodos genéricos:** permitem que o programador defina, com uma única declaração de método, um conjunto de métodos relacionados.

Introdução

- Os genéricos também fornecem **segurança de tipo em tempo de compilação**, permitindo a detecção de tipos inválidos em tempo de compilação.

Introdução

O que fazer para escrever um único método de ordenação **ordena** para elementos em um array de **Integer**, em um array de **String** ou em um array de qualquer tipo que suporte ordenamento?

- Escrever um **método genérico** para ordenar um objeto array e então invocar esse método para ordenar arrays de **Integer**, arrays de **Double** e arrays de **String**.

Introdução

O que fazer para escrever uma única classe **Stack** que seria utilizada como uma pilha de inteiros, uma pilha de números de ponto flutuante, uma pilha de **String** ou uma pilha de qualquer outro tipo?

- Escrever uma única **classe Stack genérica** que manipulasse uma pilha de objetos e instanciasse objetos **Stack** em uma pilha de **Integer**, uma pilha de **Double**, uma pilha de **String**, etc.

Exemplos de aplicações com Genéricos

- **Métodos sobrecarregados** são bastante utilizados para realizar operações semelhantes em tipos diferentes de dados.
- Ex: Utilização de três métodos **printArray** sobrecarregados para imprimir um array de tipos diferentes.

Ex: métodos de impressão para arrays de tipos diferentes

- Método `printArray` para imprimir um array de `Integer`

```
public static void printArray( Integer[] inputArray )
{
    // exibe elementos do array
    for ( Integer element : inputArray )
        System.out.printf( "%s ", element );

    System.out.println();
} // fim do método printArray
```

Ex: métodos de impressão para arrays de tipos diferentes

- Método **printArray** para imprimir um array de **Double**

```
public static void printArray( Double[] inputArray )
{
    // exibe elementos do array
    for ( Double element : inputArray )
        System.out.printf( "%s ", element );

    System.out.println();
} // fim do método printArray
```

Ex: métodos de impressão para arrays de tipos diferentes

- Método **printArray** para imprimir um array de **Character**

```
public static void printArray(Character[] inputArray )
{
    // exibe elementos do array
    for ( Character element : inputArray )
        System.out.printf( "%s ", element );

    System.out.println();
} // fim do método printArray
```

Ex: métodos de impressão para arrays de tipos diferentes

- Cada chamada a **printArray** corresponde a uma das declarações desse método

```
public static void main( String args[] ) {  
    // cria arrays de Integer, Double e Character  
    Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };  
    Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };  
    Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' };  
  
    System.out.println( "Array integerArray contem:" );  
    printArray( integerArray ); // passa um array de Integers  
    System.out.println( "\nArray doubleArray contem:" );  
    printArray( doubleArray ); // passa um array Doubles  
    System.out.println( "\nArray characterArray contem:" );  
    printArray( characterArray ); // passa um array de Characters  
} // fim de main
```

Ex: métodos de impressão para arrays de tipos diferentes

- Quando o compilador encontra uma chamada de método, ele sempre tenta localizar uma declaração de método com o **mesmo nome de método e parâmetros** que correspondam aos tipos de argumento da chamada.

```
...  
printArray( integerArray );  
...
```

O compilador determina o tipo do argumento de **integerArray** e tenta localizar um método chamado **printArray** que especifica um único parâmetro **Integer[]**

Ex: métodos de impressão para arrays de tipos diferentes

- Nesse exemplo, os tipos de elementos dos arrays aparecem:
 - Nos cabeçalhos dos métodos

```
public static void printArray( Integer[] inputArray )  
public static void printArray( Double[] inputArray )  
public static void printArray( Character[] inputArray )
```

– Nas instruções **for-each**

```
for ( Integer element : inputArray )  
for ( Double element : inputArray )  
for ( Character element : inputArray )
```

Ex: métodos de impressão para arrays de tipos diferentes

- Utilizando um tipo genérico, é possível declarar um método `printArray` que pode exibir as representações string dos elementos de qualquer array de objetos.

```
public static <E> void printArray( E[] inputArray )
{
    // exibe elementos do array
    for ( E element : inputArray )
        System.out.printf( "%s ", element );

    System.out.println();
} // fim do método printArray
```

Ex: métodos de impressão para arrays de tipos diferentes

- Utilizando um tipo genérico, é possível declarar um método `printArray` que pode exibir as representações string dos elementos de qualquer array de objetos.

```
public static <E> void printArray( E[] inputArray )
{
    // exibe e
    for ( E el
        System.out.println( "%s", element );

    System.out.println();
} // fim do método printArray
```

Seção de parâmetros de tipos.

Essa seção precede o tipo de retorno do método.

Ex: métodos de impressão para arrays de tipos diferentes

- Utilizando um tipo genérico é possível declarar um método para imprimir a representação de qualquer array de qualquer tipo.

O especificador de formato %s pode ser utilizado para gerar saída de qualquer objeto com representação de string – o método **toString** é chamado implicitamente

```
public static <E> void printArray(E[] inputArray)
{
    // exibe elementos do array
    for ( E element : inputArray )
        System.out.printf( "%s ", element );

    System.out.println();
} // fim do método printArray
```

Métodos genéricos

- Se as operações realizadas por **métodos sobrecarregados** forem idênticas para cada tipo de argumento, esses métodos podem ser codificados por **métodos genéricos**.
 - Representação mais compacta e conveniente.
- Pode-se escrever **uma única declaração de método genérico** que pode ser chamada com argumentos de tipos diferentes.

Declaração de métodos genéricos

- Todas as declarações de métodos genéricos têm uma **seção de parâmetros de tipos**, delimitada por colchetes angulares (ex, **<E>**)
- A seção de parâmetros de tipos contém um ou mais parâmetros de tipos, separados por vírgulas.
- Um **parâmetro de tipo** (também conhecido como variável de tipo) é um identificador que especifica um **nome genérico do tipo**.

Declaração de métodos genéricos

- Os **parâmetros de tipo** na declaração de um método genérico podem ser utilizados para especificar:
 - o tipo de retorno
 - tipos de parâmetros
 - tipos de variáveis locais
- Parâmetros de tipo atuam também como **marcadores de lugar** para os tipos dos argumentos passados ao método genérico, conhecidos como **argumentos de tipos reais**

Declaração de métodos genéricos

- O corpo de um método genérico é declarado como o de qualquer outro método.
- Os **parâmetros de tipo** podem representar **somente tipos por referência** – não tipos primitivos, como **int**, **double** e **char**
- É recomendável que parâmetros de tipo sejam especificados como **letras maiúsculas individuais**.
 - Ex: usamos **E** para representar o tipo de um **elemento** em um array (ou em outra estrutura de dados)

Declaração de métodos genéricos

- Um parâmetro de tipo pode ser **declarado somente uma vez** na seção de parâmetro de tipo, mas pode **aparecer mais de uma vez** na lista de parâmetros do método.
- Ex:

```
public static <E> void printTwoArrays(E[] array1, E[] array2)
```

- Os nomes de parâmetros de tipo não precisam ser únicos entre diferentes métodos genéricos.

Métodos genéricos –

Tradução em tempo de compilação

- Quando o compilador encontra a chamada **printArray(integerArray)**:
 - Primeiro determina o tipo do argumento **integerArray** e tenta encontrar um método **printArray** com um único parâmetro desse tipo. Não há tal método nesse exemplo!
 - Em seguida, verifica que há um método genérico **printArray** que especifica um parâmetro de array individual, e utiliza o parâmetro de tipo para representar o tipo de elemento do array.

Métodos genéricos –

Tradução em tempo de compilação

- O compilador também determina se as operações no **corpo do método genérico** podem ser aplicadas a elementos do tipo armazenado no argumento do array.
- Quando o compilador traduz o método genérico em *bytecode* Java, ele **remove a seção de parâmetros de tipo** e substitui os parâmetros de tipo por tipos reais.
- Esse processo é chamado de **erasure**

Ex: método genérico de impressão para arrays de tipos diferentes

Erasure: o compilador remove a seção de parâmetros de tipo e substitui os parâmetros de tipo por tipos reais.

```
public static <E> void printArray( E[] inputArray )
{
    // exibe elementos do array
    for ( E element : inputArray )
        System.out.printf( "%s ", element );

    System.out.println();
} // fim do método printArray
```

Ex: método genérico de impressão para arrays de tipos diferentes

Erasure: o compilador remove a seção de parâmetros de tipo e substitui os parâmetros de tipo por tipos reais.

```
public static <E> void printArray( E[] inputArray )  
{  
    // exib  
    for ( E  
        Syst  
  
        System.out.println();  
} // fim do método printArray
```

Por padrão, todos os tipos genéricos são substituídos pelo tipo **Object**. Conhecido também como **limite superior do parâmetro de tipo**

Ex: método genérico depois de a ***erasure*** ser realizada pelo compilador

Erasure: o compilador remove a seção de parâmetros de tipo e substitui os parâmetros de tipo por tipos reais.

```
public static void printArray( Object [] inputArray )
{
    // exibe elementos do array
    for ( Object element : inputArray )
        System.out.printf( "%s ", element );

    System.out.println();
} // fim do método printArray
```

Ex: método genérico depois de a ***erasure*** ser realizada pelo compilador

Há somente uma cópia desse código utilizada para todas as chamadas a `printArray`

```
public static void printArray( Object [] inputArray )
{
    // exibe elementos do array
    for ( Object element : inputArray )
        System.out.printf( "%s ", element );

    System.out.println();
} // fim do método printArray
```

Mais um exemplo

- Considere a tarefa de implementar um método genérico **maximo** para retornar o maior elemento dentre três elementos do mesmo tipo.
- **Problema:** O operador relacional $>$ (maior que) não pode ser usado com objetos.

Mais um exemplo

- Considere a tarefa de implementar um método genérico **maximo** para retornar o maior elemento dentre três elementos do mesmo tipo.
- **Problema:** O operador relacional `>` (maior que) não pode ser usado com objetos.
- **Solução:** é possível comparar dois objetos da mesma classe se essa classe implementar a interface genérica **Comparable** `<T>` (pacote **java.lang**).

Revisando

- A expressão **`object1.compareTo(object2)`** deve retornar:
 - **`0`** se os objetos forem iguais
 - **`<0`** se **`object1`** for menor que **`object2`**
 - **`>0`** se **`object1`** for maior que **`object2`**

Ex: método genérico para encontrar o máximo

```
//determina o maior dos três objetos comparable
public static <T extends Comparable<T>> T maximo(T x, T y, T z){
    T max = x; //supõe que x é inicialmente maior

    if(y.compareTo(max) > 0)
        max = y // y é o maior até agora

    if(z.compareTo(max) > 0)
        max = z // z é o maior

    return max; //retorna o maior objeto
}
```


Ex: método genérico para encontrar o máximo

parâmetros de tipo são utilizados no tipo de retorno

```
//determina o maior dos três objetos Comparable  
public static <T extends Comparable<T>> T maximo(T x, T y, T z){  
    T max = x; //supõe que x é inicialmente maior  
  
    if(y.compareTo(max) > 0)  
        max = y // y é o maior até agora  
  
    if(z.compareTo(max) > 0)  
        max = z // z é o maior  
  
    return max; //retorna o maior objeto  
}
```

Ex: método genérico para encontrar o máximo

parâmetros de tipo são utilizados na lista de parâmetros

```
//determina o maior dos três objetos comparable
public static <T extends Comparable<T>> T maximo(T x, T y, T z){
    T max = x; //supõe que x é inicialmente maior

    if(y.compareTo(max) > 0)
        max = y // y é o maior até agora

    if(z.compareTo(max) > 0)
        max = z // z é o maior

    return max; //retorna o maior objeto
}
```

Ex: método genérico para encontrar o máximo

e parâmetros de tipo são utilizados no tipo da variável local max

```
//determina o maior dos três objetos comparable
public static <T extends Comparable<T>> T maximo(T x, T y, T z){
    T max = x; //supõe que x é inicialmente maior

    if(y.compareTo(max) > 0)
        max = y // y é o maior até agora

    if(z.compareTo(max) > 0)
        max = z // z é o maior

    return max; //retorna o maior objeto
}
```

Ex: método genérico para encontrar o máximo

a seção de parâmetros de tipos especifica que T estende Comparable <T>

```
//determina o maior dos três objetos comparable
public static < T extends Comparable <T> > T maximo(T x, T y, T z){
    T max = x; //supõe que x é inicialmente maior

    if(y.compareTo(max) > 0)
        max = y // y é o maior até agora

    if(z.compareTo(max) > 0)
        max = z // z é o maior

    return max; //retorna o maior objeto
}
```

Limite superior do parâmetro de tipo

- **Object** é o limite superior padrão do parâmetro de tipo
- O limite superior do parâmetro de tipo pode ser especificado na seção de parâmetro de tipo
 - coloca-se, depois do nome do parâmetro de tipo, a palavra-chave **extends** e o **nome da classe ou interface que representa o limite superior**

Ex: método genérico para encontrar o máximo

nesse exemplo o limite superior do parâmetro de tipo foi especificado como o tipo **Comparable<T>**

```
//determina o maior dos três objetos comparable
public static < T extends Comparable <T> > T maximo(T x, T y, T z)
{
    T max = x; //supõe que x é inicialmente maior

    if(y.compareTo(max) > 0)
        max = y // y é o maior até agora

    if(z.compareTo(max) > 0)
        max = z // z é o maior

    return max; //retorna o maior objeto
}
```

Ex: método genérico para encontrar o máximo

```
//determina o maior dos três objetos comparable  
public static < T extends Comparable <T> > T maximo(T x, T y, T z)  
{  
    T max = x; //supõe que x é inicialmente maior  
  
    if(y.compareTo(max) > 0)  
        max = y;  
  
    if(z.compareTo(max) > 0)  
        max = z;  
  
    return max;  
}
```

Somente objetos **Comparable<T>** podem ser passados como argumentos para o método.

Qualquer objeto diferente desse causará um erro de compilação

Ex: método genérico para encontrar o máximo

Erasure: durante a tradução em tempo de compilação, o compilador remove a seção de parâmetros de tipo e substitui os parâmetros de tipo pelo **limite superior do parâmetro de tipo**

```
//determina o maior dos três objetos comparable
public static < T extends Comparable <T> > T maximo(T x, T y, T z)
{
    T max = x; //supõe que x é inicialmente maior

    if(y.compareTo(max) > 0)
        max = y // y é o maior até agora

    if(z.compareTo(max) > 0)
        max = z // z é o maior

    return max; //retorna o maior objeto
}
```


Ex: método generico depois de a erasure ser realizada pelo compilador

Erasure: durante a tradução em tempo de compilação, o compilador remove a seção de parâmetros de tipo e substitui os parâmetros de tipo pelo **limite superior do parâmetro de tipo**

```
//determina o maior dos três objetos comparable
public static Comparable maximo(Comparable x, Comparable y, Comparable z){
    Comparable max = x; //supõe que x é inicialmente maior

    if(y.compareTo(max) > 0)
        max = y // y é o maior até agora

    if(z.compareTo(max) > 0)
        max = z // z é o maior

    return max; //retorna o maior objeto
}
```

Classes genéricas

- O conceito de uma estrutura de dados, como uma pilha, pode ser entendido independentemente do tipo que ela manipula.
- Classes genéricas fornecem um meio de descrever o conceito de uma pilha (ou de qualquer outra classe) de uma **maneira independente do tipo**.

Classes genéricas

- Uma classe **Stack** genérica poderia ser a base para criar várias classes **Stack**, por exemplo:
 - **Stack de Double**
 - **Stack de Integer**
 - **Stack de Character**
- São conhecidas como **classes parametrizadas** porque aceitam um ou mais parâmetros.

Declaração de classes genéricas

- A declaração de uma classe genérica se parece com a declaração de uma não-genérica, exceto que o nome da classe é seguido por uma **seção de parâmetros de tipo**.

```
public class Stack< E >

    private final int size; // número de elementos na pilha
    private int top; // localização do elemento superior
    private E[] elements; // array que armazena elementos na pilha
    ...
}
```

Declaração de classes genéricas

- A declaração de uma classe genérica se parece com a declaração de uma não-genérica, exceto que o nome do **de parâmetro**

O parâmetro de tipo **E** representa o tipo de elemento que a **Stack** manipulará.

```
public class Stack< E >
```

```
    private final int size; // número de elementos na pilha
```

```
    private int top; // localização do elemento superior
```

```
    private E[] elements; // array que armazena elementos na pilha
```

```
    ...
```

Declaração de classes genéricas

- A declaração de uma classe genérica se parece com a declaração de uma não-genérica, exceto que o tipo de elemento é declarado entre colchetes e precedido por **E**.

de p

- A classe **Stack** declara a variável **elements** como um array do tipo **E**
- Esse array armazenará os elementos da **Stack**

```
public class
```

```
private final int top; // número de elementos na pilha
private int top; // localização do elemento superior
private E[] elements; // array que armazena elementos da pilha
...
```

Declaração de classes genéricas

- A declaração de uma classe genérica se parece com a declaração de uma não-genérica, exceto que o tipo de dados é substituído por uma classe genérica.

como criar o vetor?

```
public class
```

```
private final int size; // número de elementos na pilha  
private int top; // posição do elemento superior  
private E[] elements; // array que armazena elementos da pilha  
...
```

Exemplo classe Stack genérica

- O compilador Java não permite usar parâmetros de tipo em expressões de criação de arrays.
- **Solução:** criar um array do tipo **Object** e fazer uma coerção na referência retornada por new para o tipo **E[]**

```
elements = ( E[ ] ) new Object[ size ]; //cria o array
```


Exemplo classe Stack genérica

- Construtor

```
public Stack (int s)
{
    size = s > 0 ? s: 10; // configura o tamanho de Stack
    top=-1; //Stack inicialmente vazia
    elements = ( E[] ) new Object[ size ]; //cria o array
} // fim do construtor
```

Exemplo classe Stack genérica

- Método **push.Stack**

```
// insere o elemento na pilha; se bem-sucedido retorna true;  
// caso contrário, lança uma FullStackException  
public void push( E pushValue )  
{  
    if ( top == size - 1 ) // se a pilha estiver cheia  
        throw new FullStackException("Stack is full");  
  
    elements[ ++top ] = pushValue; // insere pushValue na Stack  
} // fim do método push
```

Exemplo classe Stack genérica

- Método **push**.Stack

Lança uma `FullStackException`,
uma nova exceção que estende
`RuntimeException` (ver pag.
655 livro Deitel)

```
// insere o elemento na pilha; bem sucedido retorna true;  
// caso contrário, lança uma FullStackException  
public void push( E pushValue )  
{  
    if ( top == size - 1 ) // se a pilha estiver cheia  
        throw new FullStackException("Stack is full");  
  
    elements[ ++top ] = pushValue; // insere pushValue na Stack  
} // fim do método push
```

Exemplo classe Stack genérica

- Método **pop.Stack**

```
// retorna o elemento superior se não estiver vazia;  
// caso contrário lança uma EmptyStackException  
public E pop()  
{  
    if ( top == -1 ) // se pilha estiver vazia  
        throw new EmptyStackException( "Stack is empty, cannot pop" );  
  
    return elements[ top-- ]; // remove e retorna o elemento superior  
} // fim do método pop
```

Exemplo classe Stack genérica

- Método **pop**.Stack

Lança uma `EmptyStackException`, uma nova exceção que estende `RuntimeException` (ver pag. 655 livro Deitel)

```
// retorna o elemento superior se não estiver vazia;  
// caso contrário lança uma EmptyStackException  
public E pop()  
{  
    if ( top == -1 ) // se pilha estiver vazia  
        throw new EmptyStackException( "Stack is empty, cannot pop" );  
  
    return elements[ top-- ]; // remove e retorna o elemento superior  
} // fim do método pop
```

Exemplo classe Stack genérica

```
public static void main(String args[]){  
    Stack<Double> doubleStack=new Stack<Double> (3);  
    Stack<Integer> integerStack=new Stack<Integer>(10);  
    doubleStack.push(1.1);  
    doubleStack.push(1.5);  
    doubleStack.push(1.3);  
    doubleStack.push(1.7);  
    Double popValue=doubleStack.pop();  
    integerStack.push(1);  
    integerStack.push(5);  
    Integer i1=integerStack.pop();  
    int i2=integerStack.pop();  
    Integer i3=integerStack.pop();  
}
```

Exemplo classe Stack genérica

```
public static void main(String args[]){  
    Stack<Double> doubleStack=new Stack<Double> (3);  
    Stack<Integer> integerStack=new Stack<Integer>(10);  
    doubleStack.push(1.1);  
    doubleStack.push(1.5);  
    doubleStack.push(1.3);  
    doubleStack.push(1.2);  
    Double popValue=doubleStack.pop();  
    integerStack.push(1);  
    integerStack.push(5);  
    Integer i1=integerStack.pop();  
    int i2=integerStack.pop();  
    Integer i3=integerStack.pop();  
}
```

Double é o argumento de tipo da Stack

Exemplo classe Stack genérica

```
public static void main(String args[]){  
    Stack<Double> doubleStack=new Stack<Double> (3);  
    Stack<Integer> integerStack=new Stack<Integer>(10);  
    doubleStack.push(1);  
    doubleStack.push(1.5);  
    doubleStack.push(1.3);  
    doubleStack.push(1);  
    Double popValue=doubleStack.pop();  
    integerStack.push(1);  
    integerStack.push(5);  
    Integer i1=integerStack.pop();  
    int i2=integerStack.pop();  
    Integer i3=integerStack.pop();  
}
```

Integer é o argumento de tipo da Stack

Exemplo classe Stack genérica

```
public static void main(String args[]){  
    Stack<Double> doubleStack=new Stack<Double> (3);  
    Stack<Integer> integerStack=new Stack<Integer>(10);  
    doubleStack.push(1.1);  
    doubleStack.push(1.5);  
    doubleStack.push(1.3);  
    doubleStack.push(1.7);  
    Double popValue = doubleStack.pop();  
    integerStack.push(1);  
    integerStack.push(5);  
    Integer i1=integerStack.pop();  
    int i2=integerStack.pop();  
    Integer i3=integerStack.pop();  
}
```

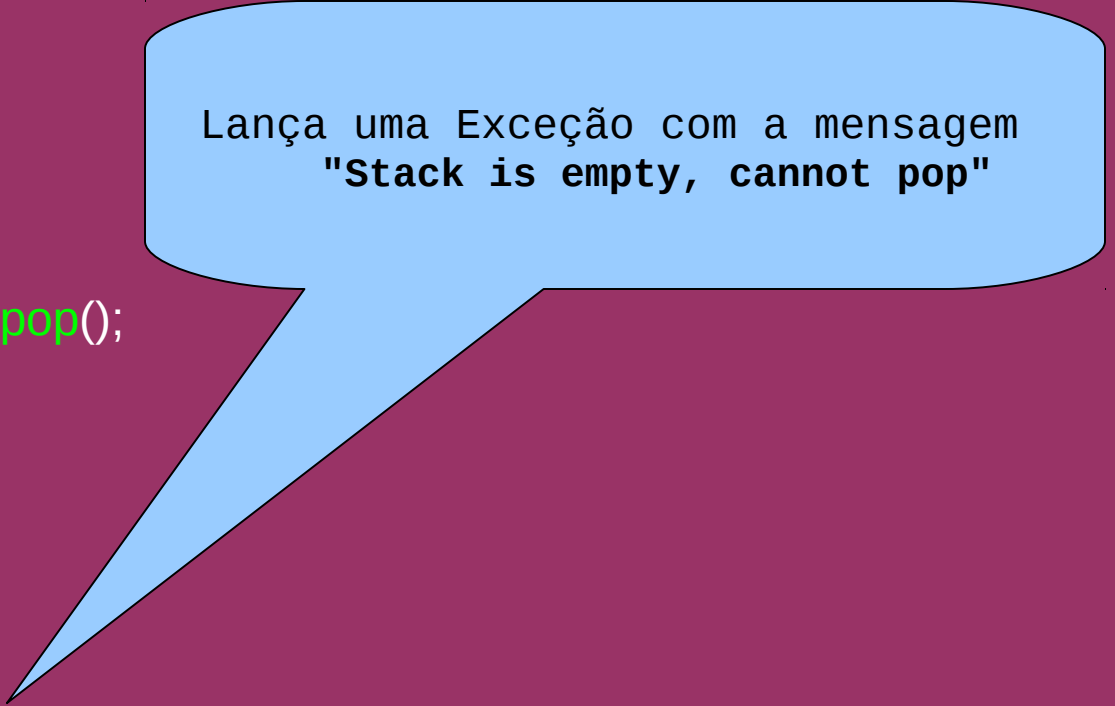
Lança uma Exceção com a mensagem **"Stack is full"**

Exemplo classe Stack genérica

```
public static void main(String args[]){  
    Stack<Double> doubleStack=new Stack<Double> (3);  
    Stack<Integer> integerStack=new Stack<Integer>(10);  
    doubleStack.push(1.1);  
    doubleStack.push(1.5);  
    doubleStack.push(1.3);  
    doubleStack.push(1.7);  
    Double popValue=doubleStack.pop();  
    integerStack.push(1);  
    integerStack.push(5);  
    Integer i1=integerStack.pop();  
    int i2=integerStack.pop();  
    Integer i3=integerStack.pop();  
}
```

Exemplo classe Stack genérica

```
public static void main(String args[]){  
    Stack<Double> doubleStack=new Stack<Double> (3);  
    Stack<Integer> integerStack=new Stack<Integer>(10);  
    doubleStack.push(1.1);  
    doubleStack.push(1.5);  
    doubleStack.push(1.3);  
    doubleStack.push(1.7);  
    Double popValue=doubleStack.pop();  
    integerStack.push(1);  
    integerStack.push(5);  
    Integer i1=integerStack.pop();  
    int i2=integerStack.pop();  
    Integer i3=integerStack.pop();  
}
```



Lança uma Exceção com a mensagem
"Stack is empty, cannot pop"

Exercício

- Defina um método genérico chamado de *getMidPointValue* que retorne o elemento localizado no ponto médio de um determinado array. Lembre-se de utilizar apropriadamente os parâmetros para tipos genéricos.