

# Programação Orientada a Objetos

Prof. Paulo Henrique Pisani

<http://professor.ufabc.edu.br/~paulo.pisani/>

# Tópicos

- Generics
- Herança com Generics
- Parâmetros de tipo limitados
- Métodos com Generics
- Type Erasure
- Algumas restrições
- Mais Generics

# Generics

# Generics

- Permite que **tipos de dados** (e.g. classes e interfaces) sejam parâmetros na definição de:
  - Classes
  - Interfaces
  - Métodos
- Dessa forma, podemos reaproveitar código, parametrizando o **tipo de dados**;
- Generics também é muito útil para evitar erros de codificação (veremos exemplos durante a aula).

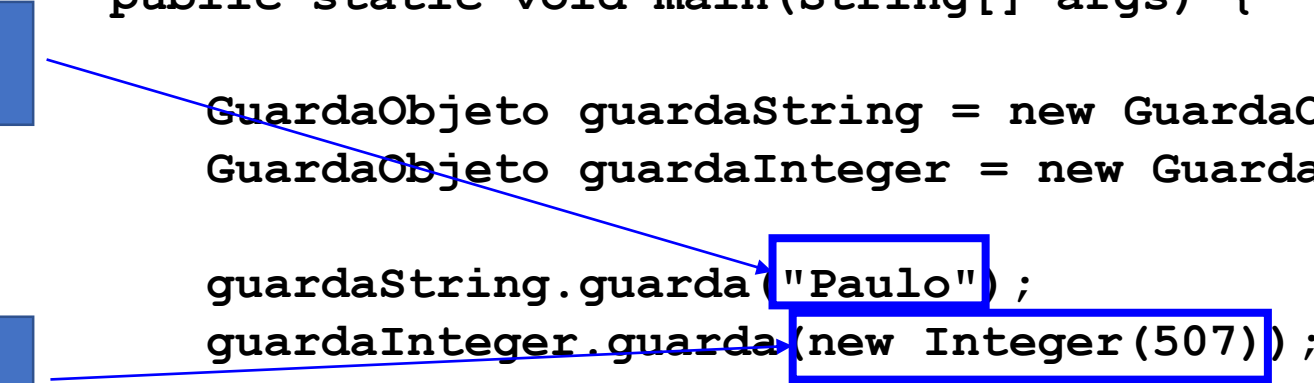
# Primeiro exemplo...

- Vamos criar uma classe que guarda um **objeto** qualquer:

```
public class GuardaObjeto {  
  
    private Object objeto;  
  
    public void guarda(Object objeto) {  
        this.objeto = objeto;  
    }  
  
    public Object get() {  
        return this.objeto;  
    }  
  
}
```

# Primeiro exemplo...

```
public class Principal {  
  
    public static void main(String[] args) {  
  
        GuardaObjeto guardaString = new GuardaObjeto();  
        GuardaObjeto guardaInteger = new GuardaObjeto();  
  
        guardaString.guarda("Paulo");  
        guardaInteger.guarda(new Integer(507));  
  
        System.out.println(guardaString.get());  
        System.out.println(guardaInteger.get());  
  
    }  
}
```



# Mudando o exemplo...

- Queremos que apenas instâncias de Strings sejam guardadas em guardaString; também queremos que apenas instâncias de Integer sejam guardadas em guardaInteger.
- Entretanto, nada impede que uma String seja guardada em guardaInteger!
  - Veja a seguir...

# Mudando o exemplo...

```
public class Principal {  
  
    public static void main(String[] args) {  
  
        GuardaObjeto guardaString = new GuardaObjeto();  
        GuardaObjeto guardaInteger = new GuardaObjeto();  
  
        guardaString.guarda("Paulo");  
        guardaInteger.guarda(new Integer(507));  
  
        System.out.println(guardaString.get());  
        System.out.println(guardaInteger.get());  
  
        guardaInteger.guarda("Professor");  
    }  
}
```



O que podemos fazer para forçar o tipo?

O que podemos fazer para forçar o tipo?

Ah, vamos fazer duas classes! Uma que trabalha com String e outra com Integer.



# O que podemos fazer para forçar o tipo?

Ah, vamos fazer duas classes! Uma que trabalha com String e outra com Integer.

Mas assim teríamos que copiar código! Podemos reutilizar código usando generics!



# Podemos forçar o tipo com Generics!

- Criação de classes com Generics:

```
public class NomeClasse<T1, T2, ..., Tn> {  
    ...  
}
```



Parâmetros de tipo

# Podemos forçar o tipo com Generics!

Parâmetro de tipo

```
public class GuardaObjeto<T> {  
    private T objeto;  
  
    public void guarda(T objeto) {  
        this.objeto = objeto;  
    }  
  
    public T get() {  
        return this.objeto;  
    }  
}
```

# Agora podemos especificar o parâmetro de tipo

```
public class Principal {  
  
    public static void main(String[] args) {  
  
        GuardaObjeto<String> guardaString = new GuardaObjeto<String>();  
        GuardaObjeto<Integer> guardaInteger = new GuardaObjeto<Integer>();  
  
        guardaString.guarda("Paulo");  
        guardaInteger.guarda(new Integer(507));  
  
        System.out.println(guardaString.get());  
        System.out.println(guardaInteger.get());  
  
        guardaInteger.guarda("Professor");  
    }  
}
```

Erro de compilação!

error: incompatible types: String cannot be converted to Integer  
guardaInteger.guarda("Professor");

# Tipos com generics

- Diversas classes e interfaces são implementados com Generics:
  - ArrayList<E>
  - LinkedList<E>
  - Iterable<E>
  - List<E>
  - Deque<E>
- Vamos usar o ArrayList na aula de hoje...

# java.util.ArrayList

- É uma classe que gerencia um array;
- Alguns métodos úteis:
  - add
  - remove
  - get
  - size

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

```
public class Principal {
```

```
    public static void main(String[] args) {
```

Esse ArrayList  
permite a inserção  
de qualquer objeto

```
        ArrayList lista = new ArrayList();  
        lista.add("Paulo");  
        lista.add("507-2");  
        lista.add(new Integer(123));  
        lista.add(new Scanner(System.in));
```

```
        for (int i = 0; i < lista.size(); i++)  
            System.out.println(lista.get(i));
```

```
    }
```

```
}
```



# java.util.ArrayList

Note: Principal.java uses unchecked or unsafe operations.  
Note: Recompile with -Xlint:unchecked for details.

A mensagem acima é exibida na compilação, pois não especificamos o tipo

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

public class Principal {

    public static void main(String[] args) {

        ArrayList lista = new ArrayList();
        lista.add("Paulo");
        lista.add("507-2");
        lista.add(new Integer(123));
        lista.add(new Scanner(System.in));

        for (int i = 0; i < lista.size(); i++)
            System.out.println(lista.get(i));

    }
}
```

# Generics no ArrayList

- Podemos especificar um tipo no ArrayList:

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

public class Principal {
```

```
    public static void main(String[] args) {
```

```
        ArrayList<Object> lista = new ArrayList<Object>();
```

```
        lista.add("Paulo");
```

```
        lista.add("507-2");
```

```
        lista.add(new Integer(123));
```

```
        lista.add(new Scanner(System.in));
```

```
        for (int i = 0; i < lista.size(); i++)
```

```
            System.out.println(lista.get(i));
```

```
    }
```

```
}
```

Este tipo pode até mesmo ser Object

# Generics no ArrayList

- Podemos especificar um tipo no ArrayList:

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

```
public class Principal {
```

```
    public static void main(String[] args) {
```

```
        ArrayList<String> lista = new ArrayList<String>();
```

```
        lista.add("Paulo");
```

```
        lista.add("507-2");
```

```
        lista.add(new Integer(123));
```

```
        lista.add(new Scanner(System.in));
```

```
        for (int i = 0; i < lista.size(); i++)  
            System.out.println(lista.get(i));
```

```
    }
```

```
}
```

String

Agora estas linhas  
geram um erro de  
compilação

# Herança com Generics

# Vamos criar uma fila com Generics

```
import java.util.ArrayList;

public class Fila<T> {

    private ArrayList<T> fila = new ArrayList<T>();

    public void enqueue(T item) {
        fila.add(item);
    }

    public T dequeue() {
        if (fila.size() == 0)
            return null;
        T item = fila.get(0);
        fila.remove(0);
        return item;
    }
}
```

Observe que essa implementação de fila não é a ideal, pois faz uso de um ArrayList. Utilizaremos apenas como exemplo de aplicação do Generics.

# Usando a fila

```
public class Principal {  
  
    public static void main(String[] args) {  
  
        Fila f1 = new Fila();  
        f1.enqueue("ABC");  
        f1.enqueue("DEF");  
        f1.enqueue(123);  
        System.out.println(f1.dequeue());  
        System.out.println(f1.dequeue());  
        System.out.println(f1.dequeue());  
  
    }  
}
```

Veja que não  
especificamos o tipo

# Usando a fila


```
public class Principal {  
  
    public static void main(String[] args) {  
  
        Fila<String> f1 = new Fila<String>();  
        f1.enqueue("ABC");  
        f1.enqueue("DEF");  
        f1.enqueue(123);  
        System.out.println(f1.dequeue());  
        System.out.println(f1.dequeue());  
        System.out.println(f1.dequeue());  
  
    }  
}
```

Agora esta linha gera  
um erro de compilação

Especificamos o  
tipo String

# Vamos estender a classe Fila!

Veja que não especificamos  
o tipo em Fila



```
public class FilaDuvidas extends Fila {  
  
    private String professor;  
  
    public FilaDuvidas(String professor) {  
        this.professor = professor;  
    }  
  
    public String getProfessor() {  
        return this.professor;  
    }  
}
```



# Vamos estender a classe Fila!

Mas podemos especificar também



```
public class FilaDuvidas extends Fila<String> {  
  
    private String professor;  
  
    public FilaDuvidas(String professor) {  
        this.professor = professor;  
    }  
  
    public String getProfessor() {  
        return this.professor;  
    }  
}
```

# Usando a FilaDuvidas

A classe FilaDuvidas não possui o parâmetro de tipo (Generics)

```
public class SistemaDuvidas {  
  
    public static void main(String[] args) {  
        FilaDuvidas fila = new FilaDuvidas("Paulo");  
  
        fila.enqueue("Quando sera a P2?");  
        fila.enqueue("O que eh uma exception unchecked?");  
        fila.enqueue(123);  
    }  
}
```

Esta linha gera um erro de compilação!  
FilaDuvidas estende Fila<String>

# Usando a FilaDuvidas

Erro de compilação! A classe FilaDuvidas não possui o parâmetro de tipo (Generics)

```
public class SistemaDuvidas {  
  
    public static void main(String[] args) {  
        FilaDuvidas<String> fila = new FilaDuvidas<String>("Paulo");  
  
        fila.enqueue("Quando sera a P2?");  
        fila.enqueue("O que eh uma exception unchecked?");  
        fila.enqueue(123);  
    }  
}
```

Esta linha gera um erro de compilação!  
FilaDuvidas estende Fila<String>

# Especificando tipo em FilaDuvidas

Agora criamos uma FilaDuvidas genérica, o tipo é repassado para Fila



```
public class FilaDuvidas<T> extends Fila<T> {
```

```
    String professor;
```

```
    public FilaDuvidas(String professor) {  
        this.professor = professor;  
    }
```

```
    public String getProfessor() {  
        return this.professor;  
    }
```

```
}
```

# Usando a FilaDuvidas

Agora a classe FilaDuvidas possui o parâmetro de tipo (Generics)

```
public class SistemaDuvidas {  
  
    public static void main(String[] args) {  
        FilaDuvidas<String> fila = new FilaDuvidas<String>("Paulo");  
  
        fila.enqueue("Quando sera a P2?");  
        fila.enqueue("O que eh uma exception unchecked?");  
    }  
}
```

# Podemos adicionar mais parâmetros de tipo!

```
public class FilaDuvidas<T, S> extends Fila<T> {  
  
    S professor;  
  
    public FilaDuvidas(S professor) {  
        this.professor = professor;  
    }  
  
    public S getProfessor() {  
        return this.professor;  
    }  
  
}
```

# Usando a FilaDuvidas

Agora a classe FilaDuvidas possui dois parâmetros de tipo (Generics)

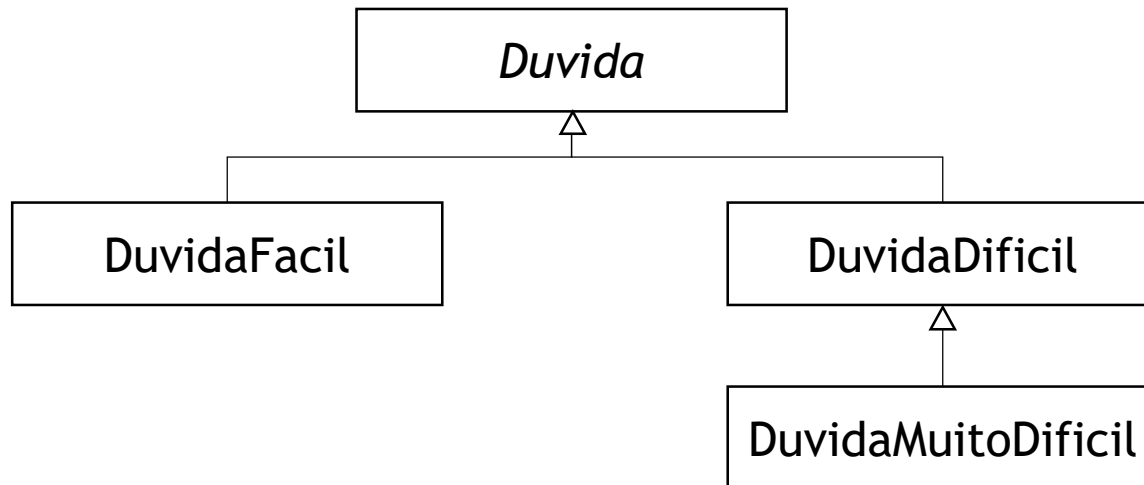
```
public class SistemaDuvidas {  
  
    public static void main(String[] args) {  
        FilaDuvidas<String, String> fila = new FilaDuvidas<String, String>("Paulo");  
  
        fila.enqueue("Quando sera a P2?");  
        fila.enqueue("O que eh uma exception unchecked?");  
  
    }  
  
}
```

# Parâmetros de tipo limitados



# Parâmetros de tipo limitados

- Podemos limitar os tipos aceitos usando **extends** e **super**:
  - `<T extends Duvida>`
- Para isso, vamos considerar a seguinte hierarquia de classes:



# Classe Duvida

```
public abstract class Duvida {  
  
    private String texto;  
  
    public Duvida(String texto) {  
        this.texto = texto;  
    }  
  
    @Override  
    public String toString() {  
        return "Duvida=" + this.texto;  
    }  
  
}
```

```
public class DuvidaFacil extends Duvida {

    public DuvidaFacil(String texto) {
        super(texto);
    }

}

public class DuvidaDifícil extends Duvida {

    public DuvidaDifícil(String texto) {
        super(texto);
    }

}

public class DuvidaMuitoDifícil extends DuvidaDifícil {

    public DuvidaMuitoDifícil(String texto) {
        super(texto);
    }

}
```

# Voltaremos a usar a seguinte versão da FilaDuvidas

```
public class FilaDuvidas<T> extends Fila<T> {  
  
    String professor;  
  
    public FilaDuvidas(String professor) {  
        this.professor = professor;  
    }  
  
    public String getProfessor() {  
        return this.professor;  
    }  
  
}
```

# Agora a lista de dúvidas aceita apenas instâncias de Duvida

```
public class SistemaDuvidas {  
  
    public static void main(String[] args) {  
        FilaDuvidas<Duvida> fila = new FilaDuvidas<Duvida>("Paulo");  
  
        fila.enqueue(new DuvidaFacil("Quando sera a P2?"));  
        fila.enqueue(new DuvidaDificil("O println eh um metodo de instancia ou de classe?"));  
        fila.enqueue(new DuvidaMuitoDificil("Pra que serve o synchronized?"));  
  
        Duvida d = (Duvida) fila.dequeue();  
        while (d != null) {  
            System.out.println(d);  
            d = (Duvida) fila.dequeue();  
        }  
    }  
}
```

# Agora a lista de dúvidas aceita apenas instâncias de Duvida

```
public class SistemaDuvidas {  
  
    public static void main(String[] args) {  
        FilaDuvidas<Duvida> fila = new FilaDuvidas<Duvida>("Paulo");  
  
        fila.enqueue(new DuvidaFacil("Quando sera a P2?"));  
        fila.enqueue(new DuvidaDificil("O println eh um metodo de instancia ou de classe?"));  
        fila.enqueue(new DuvidaMuitoDificil("Pra que serve o synchronized?"));  
  
        Duvida d = (Duvida) fila.desenfileira();  
        while (d != null) {  
            System.out.println(d);  
            d = (Duvida) fila.desenfileira();  
        }  
    }  
}
```

Não precisamos mais do cast  
em Duvida quando  
chamamos desenfileira

# Agora a lista de dúvidas aceita apenas instâncias de Duvida

```
public class SistemaDuvidas {  
  
    public static void main(String[] args) {  
        FilaDuvidas<Duvida> fila = new FilaDuvidas<Duvida>("Paulo");  
  
        fila.enqueue(new DuvidaFacil("Quando sera a P2?"));  
        fila.enqueue(new DuvidaDificil("O println eh um metodo de instancia ou de classe?"));  
        fila.enqueue(new DuvidaMuitoDificil("Pra que serve o synchronized?"));  
  
        Duvida d = fila.dequeue();  
        while (d != null) {  
            System.out.println(d);  
            d = fila.dequeue();  
        }  
    }  
}
```

Não precisamos mais do cast  
em Duvida quando  
chamamos dequeue

# Podemos limitar os tipos aceitos!

```
public class FilaDuvidas<T extends Duvida> extends Fila<T> {  
  
    String professor;  
  
    public FilaDuvidas(String professor) {  
        this.professor = professor;  
    }  
  
    public String getProfessor() {  
        return this.professor;  
    }  
  
}
```

Assim permitimos apenas que seja usada a classe Duvida ou subclasses de Duvida

A palavra-chave extends aqui pode significar tanto extends quanto implements (se o tipo for uma interface)



# Agora a lista de dúvidas aceita apenas instâncias de Duvida

```
public class SistemaDuvidas {  
  
    public static void main(String[] args) {  
        FilaDuvidas<Duvida> fila = new FilaDuvidas<Duvida>("Paulo");  
  
        fila.enqueue(new DuvidaFacil("Quando sera a P2?"));  
        fila.enqueue(new DuvidaDificil("O println eh um metodo de instancia ou de classe?"));  
        fila.enqueue(new DuvidaMuitoDificil("Pra que serve o synchronized?"));  
  
        Duvida d = fila.dequeue();  
        while (d != null) {  
            System.out.println(d);  
            d = fila.dequeue();  
        }  
    }  
}
```

Ok!

# Agora a lista de dúvidas aceita apenas instâncias de Duvida

```
public class SistemaDuvidas {  
  
    public static void main(String[] args) {  
        FilaDuvidas<Object> fila = new FilaDuvidas<Object>("Paulo");  
  
        fila.enqueue(new DuvidaFacil("Quando sera a P2?"));  
        fila.enqueue(new DuvidaDificil("O println eh um metodo de instancia ou de classe?"));  
        fila.enqueue(new DuvidaMuitoDificil("Pra que ser o synchronized?"));  
  
        Duvida d = (Duvida) fila.dequeue();  
        while (d != null) {  
            System.out.println(d);  
            d = (Duvida) fila.dequeue();  
        }  
    }  
}
```

Erro de compilação!  
Object não é  
subclasse de Duvida!

error: type argument Object is not within bounds of type-variable T  
FilaDuvidas<Object> fila = new FilaDuvidas<Duvida>("Paulo");

# Métodos com Generics

# Métodos com Generics

- Podemos ter métodos com parâmetros de tipo também.

```
acesso <T1, T2, ..., Tn> retorno nomeMetodo(T1 p1, ..., Tn pn) {  
    ...  
}
```

# Método de impressão genérico

```
public class Principal {  
  
    public static <T> void imprimeObjeto(T obj) {  
        System.out.println(obj);  
    }  
  
    public static void main(String[] args) {  
  
        Principal.<String>imprimeObjeto("Sala 507-2");  
  
    }  
  
}
```

# Método de impressão genérico

```
public class Principal {  
  
    public static <T> void imprimeObjeto(T obj) {  
        System.out.println(obj);  
    }  
  
    public static void main(String[] args) {  
  
        imprimeObjeto("Sala 507-2");  
    }  
}
```

Podemos omitir o parâmetro  
de tipo na chamada

# Vamos limitar a impressão para subclasses de Duvida

```
public class Principal {  
  
    public static <T extends Duvida> void imprimeObjeto(T obj) {  
        System.out.println(obj);  
    }  
  
    public static void main(String[] args) {  
  
        imprimeObjeto(new DuvidaFacil("O que eh uma classe abstrata?"));  
  
    }  
}
```

Ok! DuvidaFacil é subclasse de Duvida.

# Vamos limitar a impressão para subclasses de Duvida

```
public class Principal {  
  
    public static <T extends Duvida> void imprimeObjeto(T obj) {  
        System.out.println(obj);  
    }  
  
    public static void main(String[] args) {  
  
        imprimeObjeto(new DuvidaFacil("O que eh uma classe abstrata?"));  
        imprimeObjeto("Sala 507-2");  
  
    }  
}
```

```
error: method imprimeObjeto in class Principal cannot be applied to given types;  
      imprimeObjeto("Sala 507-2");  
      ^  
required: T  
found: String  
reason: inferred type does not conform to upper bound(s)  
    inferred: String  
    upper bound(s): Duvida  
where T is a type-variable:  
    T extends Duvida declared in method <T>imprimeObjeto(T)  
1 error
```



# Vantagens de limitar o tipo...

- Limitando o tipo, podemos usar métodos específicos do tipo especificado;
- Vamos adicionar um método na classe Duvida para testar:

```
public abstract class Duvida {  
  
    private String texto;  
  
    public Duvida(String texto) {  
        this.texto = texto;  
    }  
  
    ...  
  
    public String getTextoDuvida() {  
        return this.texto;  
    }  
  
}
```

# Método com tipo limitado

Observe que agora conseguimos chamar um método da classe Duvida

```
public class Principal {  
  
    public static <T extends Duvida> void imprimeObjeto(T obj) {  
        System.out.println(obj.getTextoDuvida());  
    }  
  
    public static void main(String[] args) {  
  
        imprimeObjeto(new DuvidaFacil("O que eh uma classe abstrata?"));  
  
    }  
  
}
```

# Type Erasure

# Type Erasure

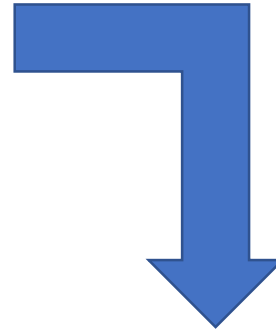
- Como vimos na aula, Generics é usado para tornar o tipo de dados um parâmetro;
- Dessa forma, adicionamos verificações de tipo mais fortes **no momento da compilação**;
- O compilador Java “apaga” informações informações de tipos genéricos! (*type erasure*)
- Portanto, **em tempo de execução**, não temos informações sobre os tipos genéricos usados.

?!

# Type Erasure

- Como vimos na aula, Generics é usado para tornar o tipo de dados um parâmetro;
- Dessa forma, adicionamos verificações de tipo mais fortes **no momento da compilação**;
- O compilador Java “apaga” informações informações de tipos genéricos! (*type erasure*)
- Portanto, **em tempo de execução**, não temos informações sobre os tipos genéricos usados.

```
public class GuardaObjeto<T> {  
  
    private T objeto;  
  
    public void guarda(T objeto) {  
        this.objeto = objeto;  
    }  
  
    public T get() {  
        return this.objeto;  
    }  
  
}
```



Após a compilação, **T** é  
definido como **Object**

```
public class GuardaObjeto {  
  
    private Object objeto;  
  
    public void guarda(Object objeto) {  
        this.objeto = objeto;  
    }  
  
    public Object get() {  
        return this.objeto;  
    }  
  
}
```

```

public class Principal {

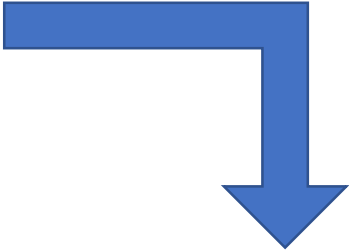
    public static void main(String[] args) {

        GuardaObjeto<String> gs = new GuardaObjeto<String>();
        GuardaObjeto<Integer> gi = new GuardaObjeto<Integer>();

        gs.guarda("Paulo");
        gi.guarda(new Integer(507));

        Integer tmp = gi.get();
    }
}

```



Após a compilação, os  
parâmetros de tipo são  
“apagados”; Casts são  
adicionados também.

```

public class Principal {

    public static void main(String[] args) {

        GuardaObjeto gs = new GuardaObjeto();
        GuardaObjeto gi = new GuardaObjeto();

        gs.guarda("Paulo");
        gi.guarda(new Integer(507));

        Integer tmp = (Integer) gi.get();
    }
}

```

```

public class Principal {

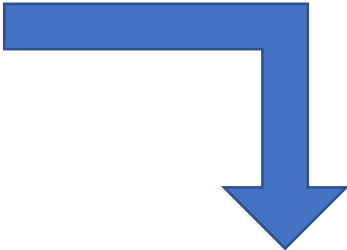
    public static void main(String[] args) {

        GuardaObjeto<String> gs = new GuardaObjeto<String>();
        GuardaObjeto<Integer> gi = new GuardaObjeto<Integer>();

        gs.guarda("Paulo");
        gi.guarda(new Integer(507));

        Integer tmp = gi.get();
    }
}

```



Após a compilação, os  
parâmetros de tipo são  
“apagados”; Casts são  
adicionados também.

Esse processo é  
conhecido como  
Type Erasure!

```

public class Principal {

    public static void main(String[] args) {

        GuardaObjeto gs = new GuardaObjeto();
        GuardaObjeto gi = new GuardaObjeto();

        gs.guarda("Paulo");
        gi.guarda(new Integer(507));

        Integer tmp = (Integer) gi.get();
    }
}

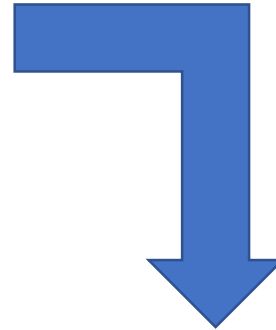
```



# Type Erasure

- Type erasure troca os parâmetros por **Object** quando o parâmetro não é limitado (era o caso do exemplo anterior);
- Caso o **parâmetro seja limitado** (com **extends**), o parâmetro é substituído por seu limite; **(exemplo a seguir)**
- Type erasure também pode gerar **métodos ponte** para preservar polimorfismo em algumas situações.

```
public class GuardaObjetoLimitado<T extends Duvida> {  
  
    private T objeto;  
  
    public void guarda(T objeto) {  
        this.objeto = objeto;  
    }  
  
    public T get() {  
        return this.objeto;  
    }  
  
}
```



Após a compilação, **T** é definido como **Duvida**

```
public class GuardaObjetoLimitado {  
  
    private Duvida objeto;  
  
    public void guarda(Duvida objeto) {  
        this.objeto = objeto;  
    }  
  
    public Duvida get() {  
        return this.objeto;  
    }  
  
}
```

```

public class Principall {

    public static void main(String[] args) {

        GuardaObjetoLimitado<DuvidaFacil> gs
            = new GuardaObjetoLimitado<DuvidaFacil>();

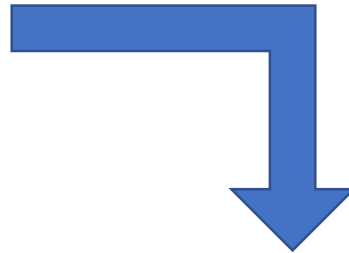
        gs.guarda(new DuvidaFacil("O que eh um metodo de classe?"));

        DuvidaFacil d = gs.get();

    }

}

```



Após a compilação, os  
parâmetros de tipo são  
“apagados”; Casts são  
adicionados também.

```

public class Principall {

    public static void main(String[] args) {

        GuardaObjetoLimitado gs = new GuardaObjetoLimitado();

        gs.guarda(new DuvidaFacil("O que eh um metodo de classe?"));

        DuvidaFacil d = (DuvidaFacil) gs.get();

    }

}

```

# Vamos examinar o bytecode

- Para isso, utilizaremos o **javap**: o Disassembler de Bytecode Java
- Usaremos ele da seguinte forma:

```
javap -c NomeDaClasse.class
```

# Classe GuardaObjeto

```
javap -c GuardaObjeto.class
```

Compiled from "GuardaObjeto.java"

```
public class GuardaObjeto<T> {
```

```
    public GuardaObjeto();
```

```
        Code:
```

```
            0: aload_0
```

```
            1: invokespecial #1               // Method java/lang/Object."<init>":()V
```

```
            4: return
```

```
    public void guarda(T);
```

```
        Code:
```

```
            0: aload_0
```

```
            1: aload 1
```

```
            2: putfield    #2               // Field objeto:Ljava/lang/Object;
```

```
            5: return
```

```
    public T get();
```

```
        Code:
```

```
            0: aload 0
```

```
            1: getfield    #2               // Field objeto:Ljava/lang/Object;
```

```
            4: areturn
```

```
}
```

Compilou  
para Object,  
conforme  
vimos nesta  
aula

# Classe GuardaObjetoLimitado

```
javap -c GuardaObjetoLimitado.class
```

Compiled from "GuardaObjetoLimitado.java"

```
public class GuardaObjetoLimitado<T extends Duvida> {
```

```
    public GuardaObjetoLimitado();
```

Code:

```
    0: aload_0
```

```
    1: invokespecial #1          // Method java/lang/Object."<init>":()V
```

```
    4: return
```

```
    public void guarda(T);
```

Code:

```
    0: aload_0
```

```
    1: aload 1
```

```
    2: putfield      #2          // Field objeto:LDuvida;
```

```
    5: return
```

```
    public T get();
```

Code:

```
    0: aload 0
```

```
    1: getfield      #2          // Field objeto:LDuvida;
```

```
    4: areturn
```

```
}
```

Compilou  
para Duvida,  
conforme  
vimos nesta  
aula

# Classe PrincipalL (onde usamos o GuardaObjetoLimitado)

```
javap -c PrincipalL.class
```

Compiled from "PrincipalL.java"

```
public class PrincipalL {  
    public PrincipalL();
```

Code:

```
    0: aload_0
```

```
    1: invokespecial #1
```

```
        // Method java/lang/Object."<init>":()V
```

```
    4: return
```

```
public static void main(java.lang.String[]);
```

Code:

```
    0: new          #2
```

```
        // class GuardaObjetoLimitado
```

```
    3: dup
```

```
    4: invokespecial #3
```

```
        // Method GuardaObjetoLimitado."<init>":()V
```

```
    7: astore_1
```

```
    8: aload_1
```

```
    9: new          #4
```

```
        // class DuvidaFacil
```

```
   12: dup
```

```
   13: ldc          #5
```

```
        // String O que eh um metodo de classe?
```

```
   15: invokespecial #6
```

```
        // Method DuvidaFacil."<init>":(Ljava/lang/String;)V
```

```
   18: invokevirtual #7
```

```
        // Method GuardaObjetoLimitado.guarda:(LDuvida;)V
```

```
   21: aload_1
```

```
   22: invokevirtual #8
```

```
        // Method GuardaObjetoLimitado.get:()LDuvida;
```

```
   25: checkcast    #4
```

```
        // class DuvidaFacil
```

```
   28: astore_2
```

```
   29: return
```

```
}
```

Adicionou o  
cast para  
DuvidaFacil,  
conforme  
vimos na aula  
também!

# Algumas restrições



# Algumas restrições

- Tipos generic **não podem ser** inicializados com tipo primitivos (int, double, etc);



```
FilaDuvidas<int> fila = new FilaDuvidas<int>("Paulo");
```

# Algumas restrições

- Não é permitido instanciar objetos com o parâmetro de tipo (na verdade há uma forma usando Reflection);

```
public class GuardaObjetoLimitado<T> {
```

```
    private T objeto;
```

```
    public void guarda(T objeto) {  
        this.objeto = new T();  
    }
```

```
    public T get() {  
        return this.objeto;  
    }
```

```
}
```

# Algumas restrições

- **Atributos de classe (static)** não podem usar o parâmetro de tipo.

```
public class GuardaObjetoLimitado<T> {
```



```
    private static T objeto;
```

```
    public void guarda(T objeto) {  
        this.objeto = objeto;  
    }
```

```
    public T get() {  
        return this.objeto;  
    }
```

```
}
```

# Mais Generics

# Mais Generics

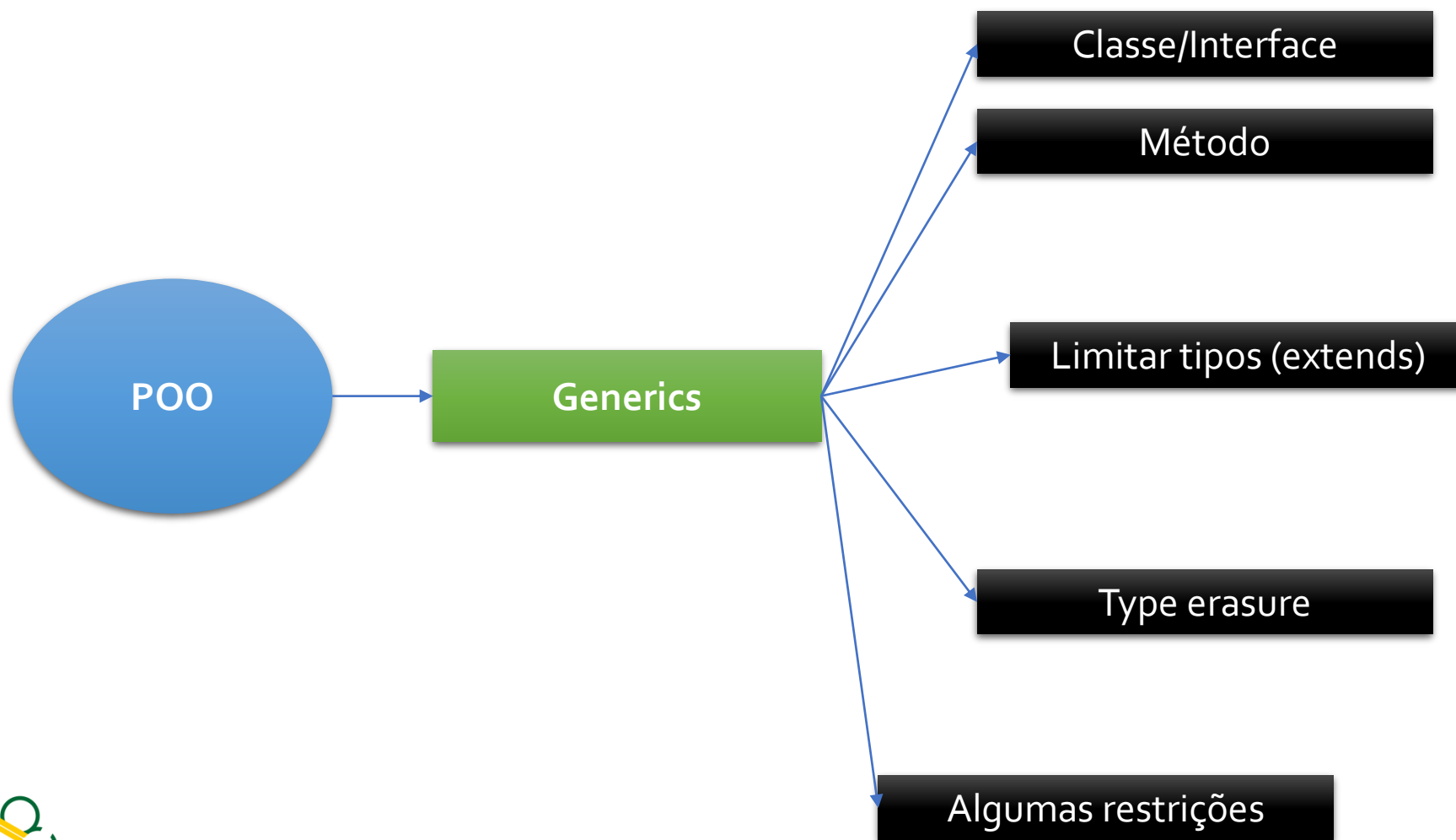
- Outros tópicos em Generics:
  - Mais restrições (vimos apenas algumas nesta aula)
  - Wildcards
  - Hierarquia
  - ...

# Convenções de nomenclatura

- E - Elemento
- K - Chave
- N - Número
- T - Tipo
- V - Valor
- S, U, V etc. - mais tipos

<https://docs.oracle.com/javase/tutorial/java/generics/types.html>

# Resumo da aula



# Referências

- Documentação Java:
  - <https://docs.oracle.com/javase/8/docs/>
  - <https://docs.oracle.com/javase/tutorial/java/generics/index.html>



# Referências (projeto pedagógico)

- BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. UML: guia do usuário. Rio de Janeiro, RJ: Campus, 2005.
- GUEDES, G. T. A. UML 2: uma abordagem prática. São Paulo, SP: Novatec, 2009.
- DEITEL, H. M.; DEITEL, P. J. Java: como programar. 6a edição. Porto Alegre, RS: Bookman, 2005.
- BARNES, D. J.; KOLLING, M. Programação orientada a objetos com Java. 4ª edição. São Paulo, SP: Editora Pearson Prentice Hall, 2009.

# Referências (projeto pedagógico)

- FLANAGAN, D. Java: o guia essencial. 5ª edição. Porto Alegre, RS: Bookman, 2006.
- BRUEGGE, B.; DUTOIT, A. H. Object-oriented software engineering: using UML, patterns, and Java. 2ª edição. Upper Saddle River, NJ: Prentice Hall, 2003.
- LARMAN, C. Utilizando UML e padrões: uma introdução à análise e ao projeto orientados a objetos e ao desenvolvimento iterativo. 3ª edição. Porto Alegre, RS: Bookman, 2007.
- FOWLER, M. UML essencial: um breve guia para a linguagem-padrão de modelagem de objetos. 3ª edição. Porto Alegre, RS: Bookman, 2005.