

# Tipos Genéricos

---

## Orientação a Objetos - DCC025

Prof. Edmar Welington Oliveira  
[oliveira.edmar@ufjf.edu.br](mailto:oliveira.edmar@ufjf.edu.br)

Universidade Federal de Juiz de Fora - UFJF  
Departamento de Ciência da Computação - DCC

# Tipos Genéricos

- Genéricos (Tipos Parametrizados)
  - Recurso introduzido no Java 5.0.
  - Sintaxe que permite restringir tipos de dados aceitos por referências genéricas.
  - Permite verificação de tipo em tempo de compilação.
  - Exemplo: coleções
    - Permitem agrupar objetos de diferentes tipos
    - Devido à sua interface genérica (baseada em Object)

# Tipos Genéricos

- Problema

- Antigamente as coleções permitiam que adicionássemos qualquer tipo de objeto entre seus elementos. Obviamente um programador mais descuidado poderia adicionar um objeto de tipo errado entre os elementos dessa coleção.

# Código

```
public class Pessoa {  
    private String nome;  
  
    public String getNome() {  
        return nome;  
    }  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
}
```

```
import java.util.*;
```

```
public class TesteColecaoValoresAleatorios {  
    public static void main(String[] args) {  
        Collection lista = new ArrayList();  
        lista.add("Valor String");  
        lista.add(new Pessoa());  
        lista.add(10);  
        lista.add(new Double(100.9));  
    }  
}
```

# Código

```
public class TesteColecaoSemGenericos {  
    public static void main(String[] args) {  
        Pessoa p1 = new Pessoa();  
        p1.setNome("Marco");  
        Pessoa p2 = new Pessoa();  
        p2.setNome("Diego");  
        Pessoa p3 = new Pessoa();  
        p3.setNome("Marcelo");  
        Pessoa p4 = new Pessoa();  
        p4.setNome("Wilson");  
        Pessoa p5 = new Pessoa();  
        p5.setNome("Paulo");  
        Collection lista = new ArrayList();  
        lista.add(p1);  
        lista.add(p2);  
        lista.add(p3);  
        lista.add(p4);  
        lista.add(p5);  
        Iterator ite = lista.iterator();  
        while (ite.hasNext()) {  
            Pessoa pessoa = (Pessoa) ite.next();  
        }  
    }  
}
```

Apesar de nossa coleção ter apenas um tipo de classe, para recuperarmos os objetos precisamos fazer um cast. Funciona perfeitamente, mas ainda pode ser melhorado.

# Código

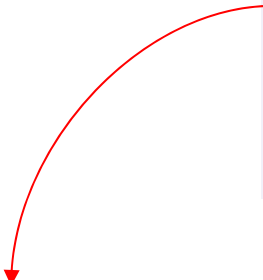
```
public class TesteColecaoComGenericos {  
    public static void main(String[] args) {  
        Pessoa p1 = new Pessoa();  
        p1.setNome("Marco");  
        Pessoa p2 = new Pessoa();  
        p2.setNome("Diego");  
1. Collection<Pessoa> lista = new ArrayList<Pessoa>();  
        lista.add(p1);  
        lista.add(p2);  
2. Iterator<Pessoa> ite = lista.iterator();  
3. while (ite.hasNext()) {  
4.     Pessoa pessoa = ite.next();  
        }  
    }  
}
```

**Collection<Pessoa> lista = new ArrayList<Pessoa>();**

Com essa sintaxe, não precisamos mais fazer cast.

# Código

```
public class TesteColecaoComGenericos {  
    public static void main(String[] args) {  
        Pessoa p1 = new Pessoa();  
        p1.setNome("Marco");  
        Pessoa p2 = new Pessoa();  
        p2.setNome("Diego");  
        Collection<Pessoa> lista = new ArrayList<Pessoa>();  
        lista.add(p1);  
        lista.add(p2);  
        1. for(Pessoa p : lista){  
        2.     System.out.println(p.getNome());  
        }  
    }  
}
```



Melhoramento do for tradicional. Não é mais necessário utilizar o **iterator**. Para cada Pessoa dentro da lista, imprime-se o nome.

## Exemplo: Caixa de Objetos

```
public class CaixaDeObjeto {  
    private Object objeto;  
  
    public void adiciona(Object objetoAdicionado) {  
        objeto = objetoAdicionado;  
    }  
  
    public Object recuperaObjeto() {  
        return objeto;  
    }  
}
```



# Exemplo: Caixa de Objetos

```
public class TesteCaixaDeObjeto {  
    public static void main(String[] args) {  
        CaixaDeObjeto caixa = new CaixaDeObjeto();  
        caixa.adiciona(new Integer(10));  
        Integer valorRecuperado = (Integer) caixa.recuperaObjeto();  
        System.out.println("Valor integer: " + valorRecuperado);  
    }  
}
```

Sem a utilização de genéricos precisamos sempre fazer conversões.

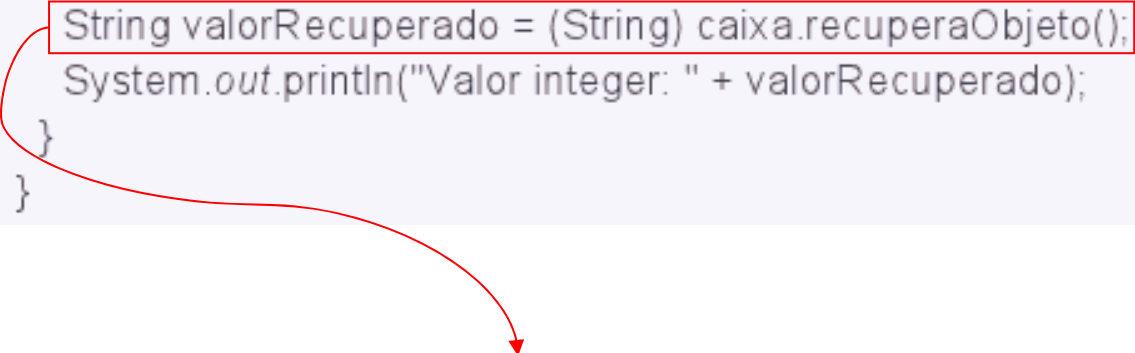
**Integer valorRecuperado = (Integer) caixa.recuperaObjeto();**

Não existe validação nenhuma em tempo de compilação já que a conversão é forçada.

# Exemplo: Caixa de Objetos

Nada impede o programador de converter para um tipo errado.  
Infelizmente esse problema só será descoberto em **tempo de execução**.

```
public class TesteCaixaDeObjeto {  
    public static void main(String[] args) {  
        CaixaDeObjeto caixa = new CaixaDeObjeto();  
        caixa.adiciona(new Integer(10));  
        String valorRecuperado = (String) caixa.recuperaObjeto();  
        System.out.println("Valor integer: " + valorRecuperado);  
    }  
}
```



Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String

# Exemplo: Caixa de Objetos

```
1. public class CaixaDeObjeto<T> {  
2.   private T objeto;  
  
3.   public void adiciona(T objetoAdicionado) {  
      objeto = objetoAdicionado;  
    }  
  
4.   public T recuperaObjeto() {  
      return objeto;  
    }  
}
```

1. Declaração do tipo parametrizado.
2. A variável objeto agora é do tipo parametrizado.
3. O método adiciona só permite que você utilize o tipo T.
4. Recupera o objeto convertido para o tipo T.

# Exemplo: Caixa de Objetos

```
public class TesteCaixaDeObjeto {  
    public static void main(String[] args) {  
        1. CaixaDeObjeto<Integer> caixa = new CaixaDeObjeto<Integer>();  
        caixa.adiciona(new Integer(10));  
        System.out.println("Valor integer: " + caixa.recuperaObjeto());  
    }  
}
```

Quando criamos um objeto do tipo CaixaDeObjeto devemos informar o tipo, conforme a sintaxe. A partir desse momento, não será mais necessário fazer cast.

A caixa de objetos só aceitará objetos do tipo Integer

## Exemplo: Caixa de Objetos

```
public class TesteCaixaDeObjeto {  
    public static void main(String[] args) {  
        CaixaDeObjeto<Integer> caixa = new CaixaDeObjeto<Integer>();  
        caixa.adiciona(new Integer(10));  
        System.out.println("Valor integer: " + caixa.recuperaObjeto());  
        CaixaDeObjeto<Pessoa> caixaPessoa = new CaixaDeObjeto<Pessoa>();  
        Pessoa pessoa = new Pessoa();  
        pessoa.setNome("Marco");  
        caixaPessoa.adiciona(pessoa);  
        System.out.println("Nome: " + caixaPessoa.recuperaObjeto().getNome());  
    }  
}
```

# Tipos Genéricos

```
interface List {  
    public void add(Object item);  
    public Object get(int index);  
    public Iterator iterator();  
    ...  
}
```

Vantagem: aceita qualquer tipo

```
interface Iterator {  
    public Object next();  
    ...  
}
```

Desvantagem: retorna tipo genérico  
e requer downcasting que só é  
verificado em tempo de execução

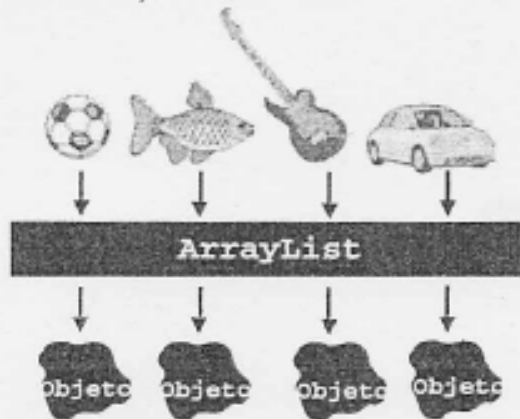
# Motivação

- Generics
  - Antes dos genéricos, o compilador não se importava com o que era inserido em um conjunto, porque todas as implementações de conjuntos eram declaradas para conter o tipo Object. Era possível inserir **qualquer coisa** em qualquer ArrayList, por exemplo. Era como se todas as ArrayLists fossem declaradas como **ArrayList<Object>**

# Motivação

## SEM os tipos genéricos

Os elementos ENTRAM como uma referência aos objetos SoccerBall, Fish, Guitar e Car



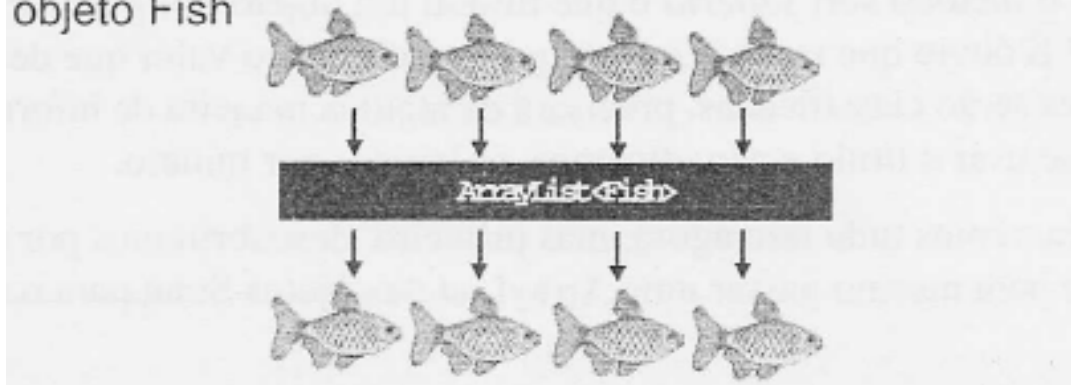
Antes de Generics, não havia como se declarar o tipo de um ArrayList, por exemplo. A referência utilizada era a classe Object. Métodos `get()` deviam usar esta referência



# Motivação

## COM os tipos genéricos

Os elementos ENTRAM somente como referência do objeto Fish



Agora, com o Generics, é possível inserir em coleções somente objetos de um tipo específico. Eles saem da coleção não mais como referência de `Object`, mas com o mesmo tipo quando entraram

# Motivação

- Generics

- Classes genéricas significam maior compatibilidade de tipos

- Praticamente todos os códigos que programadores escrevem envolvendo tipos genéricos serão códigos relacionados a conjuntos (coleções). Embora tipos genéricos possam ser utilizados de outras formas, sua finalidade principal é permitir a criação de conjuntos com **compatibilidade de tipo** - ou seja, código que faça o compilador impedir de se inserir em uma lista de Peixes um objeto do Tipo Cão.

# Antes de Generics

- Problema
  - Considere as duas classes abaixo

```
class Fazenda {  
    ...  
    private List galinheiro =  
        new ArrayList();  
    public List getGalinheiro() {  
        return galinheiro;  
    }  
}
```

```
class Galinha {  
    public void entrarNoGalinheiro(List galinheiro) {  
        galinheiro.add(this);  
    }  
}
```

# Antes de Generics

## ■ Problema

- Era preciso fazer coerção (cast) para usar os dados recuperados de coleções

```
Fazenda fazenda = Fazenda.getInstance();

Galinha g1 = new Galinha("Chocagilda");
Galinha g2 = new Galinha("Cocotalva");
// galinhas entram no galinheiro
g1.entrarNoGalinheiro(fazenda.getGalinheiro());
g2.entrarNoGalinheiro(fazenda.getGalinheiro());

Fazendeiro ze = new Fazendeiro(fazenda);
Iterator galinhas = ze.pegarGalinhas();
while(galinhas.hasNext()) {
    Galinha galinha = (Galinha) galinhas.next();
}
```

# Problema

```
g1.entrarNoGalinheiro(fazenda.getGalinheiro());  
g2.entrarNoGalinheiro(fazenda.getGalinheiro());  
...  
// entra uma raposa!!!  
Raposa galius = new Raposa("Galius");  
galius.assaltar(fazenda.getGalinheiro());  
  
Fazendeiro ze = new Fazendeiro(fazenda);  
Iterator galinhas = ze.pegarGalinhas();  
while(galinhas.hasNext()) {  
    Galinha galinha =  
        (Galinha) galinhas.next();  
}
```

```
class Raposa {  
    public void assaltar(List lugar) {  
        lugar.add(this);  
    }  
}
```

**FALHA!** Há uma raposa no meio das galinhas

[java] Exception in thread "main" java.lang.ClassCastException: Raposa

Correto! List tem add(Object) e Raposa é um Object

# Solução

## ■ Solução com Generics

### ■ Genéricos permitem associar um tipo à referência

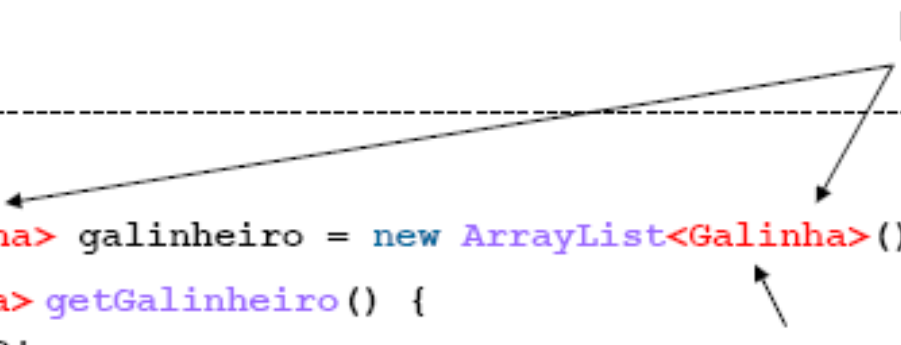
- Restringe tipos permitidos ao passado como parâmetro
- Permite verificação em tempo de compilação

### ■ Em outras palavras

- Não é necessário Type Casting para extrair os objetos das coleções. O compilador não permite colocar na coleção elementos incompatíveis com os tipos declarados

# Solução

```
class Fazenda {  
    ...  
    private List<Galinha> galinheiro = new ArrayList<Galinha>();  
    public List<Galinha> getGalinheiro() {  
        return galinheiro;  
    }  
}
```



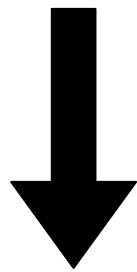
Parâmetro de tipo  
(type parameter)

Isto não é um ArrayList de Object  
mas um ArrayList de Galinha

```
class Galinha {  
    public void entrarNoGalinheiro(List<Galinha> galinheiro) {  
        galinheiro.add(this);  
    }  
}
```

# Solução

```
Fazendeiro ze = new Fazendeiro(fazenda);  
Iterator galinhas = ze.pegarGalinhas();  
while(galinhas.hasNext()) {  
    Galinha galinha =  
        (Galinha) galinhas.next();  
}
```



Agora não precisamos mais do cast

```
Fazendeiro ze = new Fazendeiro(fazenda);  
Iterator<Galinha> galinhas = ze.pegarGalinhas();  
while(galinhas.hasNext()) {  
    Galinha galinha = galinhas.next(); // sem cast!!!  
}
```



# Generics

O "E" é um espaço reservado para o tipo que você realmente usará quando declarar e criar uma ArrayList.

ArrayList é uma subclasse de AbstractList, portanto, independentemente do tipo que você especificar para a ArrayList, ele será automaticamente usado como o tipo da AbstractList.

```
public class ArrayList<E> extends AbstractList<E> implements List<E> ... {
```

O tipo (o valor de <E>) também se tornará o tipo da interface List.

```
public boolean add(E o)
```

```
    // mais código  
}
```

Aqui está a parte importante! Qualquer que seja "E" ele determinará que tipo de coisas você poderá adicionar à ArrayList.

# Generics

ESSE código:

```
ArrayList<String> thisList = new ArrayList<String>
```

Significa que a ArrayList:

```
public class ArrayList<E> extends AbstractList<E> ... {
```

```
    public boolean add(E o)
```

```
        // mais código
```

```
    }
```

Será tratada pelo compilador como:

```
public class ArrayList<String> extends AbstractList<String>... {
```

```
    public boolean add(String o)
```

```
        // mais código
```

```
    }
```

# Programação Genérica

- Generics (ou Programação Genérica)
  - Determinar para o compilador qual tipo de classe deve ser interpretada.

# Exemplo

<T> Identifica Tipo Genérico.  
T será substituída por qualquer outra classe

```
03. public class Aparelho <T> {  
04.     T objeto;  
05.  
06.     public Aparelho(T objeto) {  
07.         this.objeto = objeto;  
08.     }  
09.  
10.     public T getObjeto() {  
11.         return objeto;  
12.     }  
13.  
14.     public void setObjeto(T objeto) {  
15.         this.objeto = objeto;  
16.     }  
17. }
```

# Exemplo

```
03. public class TV {
04.     int tamanho;
05.     int canal;
06.     int volume;
07.     boolean ligada;
08.
09.     public TV(int tamanho, int canal, int volume, boolean ligada) {
10.         this.tamanho = tamanho;
11.         this.canal = canal;
12.         this.volume = volume;
13.         this.ligada = ligada;
14.     }
15.
16.     // métodos get e set
17.
18. }
```

```
03. public class Radio {
04.     public static int AM = 1;
05.     public static int FM = 2;
06.     private float frequencia;
07.     private int volume;
08.     private int banda;
09.
10.     public Radio(float frequencia, int volume, int banda) {
11.         this.frequencia = frequencia;
12.         this.volume = volume;
13.         this.banda = banda;
14.     }
15.
16.     // métodos get e set
17.
18. }
```

# Exemplo

```
package tiexpert;

public class MinhaClasse {
    public static void main(String[] args) {
        Aparelho<TV> aparelho1 = new Aparelho<TV>(new TV(29, 0, 0, false));
        Aparelho<Radio> aparelho2 = new Aparelho<Radio>(new Radio(88.1f, 0, Radio.FM));
        System.out.println(aparelho1.getObjeto().getClass());
        System.out.println(aparelho2.getObjeto().getClass());
    }
}
```

Sem o uso de generics, o compilador não conseguiria saber qual a diferença entre aparelho1 e aparelho2, pois ele os trataria da mesma forma. Nesse caso, seria obrigatório o uso de typecast para determinar ao compilador que dado está sendo processado.

Se pensarmos em uma aplicação muito grande, além de ficar cansativo ter que usar typecast toda vez que fossemos usar um aparelho, seria muito provável que erraríamos em algum momento.

# Coringa

```
12 static void printShapeCollection( Collection<Shape> collection ){
13     for ( Shape shape : collection )
14         System.out.println( shape );
15 }

26 Collection<Shape> shapes = new BasicCollection<Shape>();
27 shapes.add( new Circle( 5.0 ) );
28 shapes.add( new Rectangle( 4.5, 21.2 ) );
29 shapes.add( new Cube( ) );
30 System.out.printf( "From printShapeCollection( shapes )\n" );
31 printShapeCollection( shapes );

33 Collection<Circle> circles = new BasicCollection<Circle>();
34 circles.add( new Circle( 5.0 ) );
35 circles.add( new Circle( 15.0 ) );
36 circles.add( new Circle( 25.0 ) );
37 printShapeCollection( circles );
```

# Coringa

```
12 static void printShapeCollection( Collection<Shape> collection ){
13     for ( Shape shape : collection )
14         System.out.println( shape );
15 }

26 Collection<Shape> shapes = new BasicCollection<Shape>();
27 shapes.add( new Circle( 5.0 ) );
28 shapes.add( new Rectangle( 4.5, 21.2 ) );
29 shapes.add( new Cube( ) );
30 System.out.printf( "From printShapeCollection( shapes )\n" );
31 printShapeCollection( shapes );

33 Collection<Circle> circles = new BasicCollection<Circle>();
34 circles.add( new Circle( 5.0 ) );
35 circles.add( new Circle( 15.0 ) );
36 circles.add( new Circle( 25.0 ) );
37 //printShapeCollection( circles );
```

**Collection<gray.adts.shapes.*Shape*>** in  
UnboundedWildcardEx **cannot be applied to**  
(java.util.**Collection<gray.adts.shapes.*Circle*>**)



# Coringa

## ■ Problema

### ■ Problema do exemplo anterior

- Ainda que Circle “é um tipo de” Shape ...
- ... um Collection<Circle> NÃO “é um tipo de” Collection<Shape>

## ■ Solução

### ■ Precisamos de mais flexibilidade no tipo do parâmetro.

- Ao invés de especificar o tipo exato da coleção, queremos aceitar uma coleção que armazena uma família de tipos. A solução Java é o tipo coringa “?”, que aceita qualquer tipo. Sendo mais formal, “?” é tipo ser um “tipo desconhecido” - então Collection<?> é uma “coleção de tipo desconhecido.”

# Coringa

```
20 static void printAnyCollection( Collection<?> collection ){  
21     for ( Object element : collection )  
22         System.out.println( element );  
23 }
```

método de impressão mais geral. Pode imprimir coleções de qual tipo de dado. Note o tipo da variável no loop for.

# Coringas Limitados

## ■ Fato

- Em algumas situações podemos não querer que quaisquer tipos de dados possam ser utilizados e seria interessante colocar um limite (restrição) na família de tipos aceitos. Os coringas limitados servem para isso.
- Exemplo: O método `addAll()` de `Collection`.
- Abaixo está uma classe abstrata `AbstractCollection`

```
public abstract class AbstractCollection<E> implements Collection<E>{
```

Significa que podemos armazenar elementos do tipo `E` ou qualquer uma das subclasses de `E` na coleção.

# Coringas Limitados

```
public abstract class AbstractCollection<E> implements Collection<E>{
```

Qual deveria ser o parâmetro de `addAll()`?

1ª tentativa

```
public Boolean addAll(Collection<E> c)
```

Muito restritivo. Dessa forma, iremos impedir a adição de um `Collection<Circle>` a um `Collection<Shape>` ou um `Collection<Integer>` a um `Collection<Number>`.

2ª tentativa

```
public Boolean addAll(Collection<?> c)
```

Não restritivo o bastante. Dessa forma, permitiríamos *tentar* adicionar um `Collection<Shape>` a um `Collection<Number>`.

# Coringas Limitados

- Solução

- O que precisamos é de um mecanismo flexível que nós permita especificar uma família de tipos restritos por algum “limite superior” da família de tipos. Os coringas limitados servem para isto.

# Coringas Limitados

- Pode-se impor limites aos curingas
  - Limites superiores: `<? extends T>`
    - Aceita T e todos os seus descendentes
    - Ex: se T for Collection, aceita List, Set, ArrayList, etc.
  - Limites inferiores: `<? super T>`
    - Aceita T e todos os seus ascendentes
    - Ex: se T for ArrayList, aceita List, Collection, Object

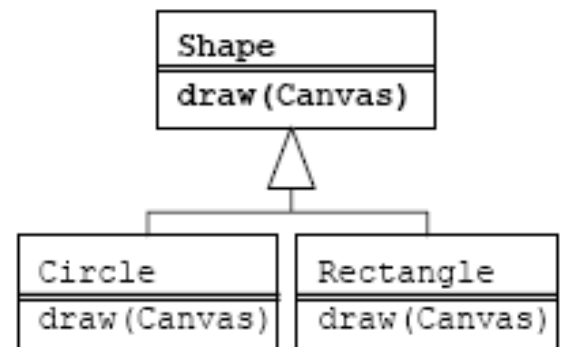
# Coringas Limitados

```
1 // método de AbstractCollection<E>
2 public boolean addAll(Collection<? extends E> c) {
3     boolean modified = false;
4     Iterator<? extends E> e = c.iterator();
5
6     while ( e.hasNext() ) {
7         if ( add( e.next() ) )
8             modified = true;
9     }
10    return modified;
11 }
```

O coringa limitado no tipo do parâmetro significa que o tipo do Collection é desconhecido, mas é limitado pelo tipo E. Ou seja, o tipo do elemento de c precisa ser E ou uma das suas subclasses.

# Coringas Limitados

```
class Canvas {  
    public void drawAll(List<? extends Shape> shapes) {  
        for(Shape s: shapes) {  
            s.draw(this);  
        }  
    }  
}
```





# Métodos Genéricos

- Classe Genérica

- Significa que a declaração da classe inclui um parâmetro de tipo

- Método Genérico

- Significa que a declaração do método usa um parâmetro de tipo em sua assinatura. É possível usar parâmetros de tipo em um método de diferentes maneiras:
    - Usando o parâmetro de tipo definido na declaração da classe
    - Usando um parâmetro de tipo que não foi definido na declaração da classe

# Métodos Genéricos

```
public class ArrayList<E> extends AbstractList<E> ... {  
  
    public boolean add(E o)
```

Quando você declarar um parâmetro de tipo para a classe, poderá simplesmente usar esse tipo em qualquer local em que usaria um tipo de classe

```
public <T extends Animal> void takeThing(ArrayList<T> ...
```

```
    public <U> void verifica(U u) {  
        System.out.println("U: " + u.getClass().getName());  
    }
```

# Tipos Genéricos

---

## Orientação a Objetos - DCC025

Prof. Edmar Welington Oliveira  
[oliveira.edmar@ufjf.edu.br](mailto:oliveira.edmar@ufjf.edu.br)

Universidade Federal de Juiz de Fora - UFJF  
Departamento de Ciência da Computação - DCC