



Genéricos





Tipos Genéricos

- ⇒ Permite abstrair sobre os tipos de dados.
- ⇒ Um tipo genérico é um tipo que não é definido em tempo de compilação, mas que é especificado em tempo de execução.
- ⇒ Permite que uma classe seja definida em termos de um conjunto de parâmetros de tipo.





Tipos Genéricos

➡ Um tipo genérico é definido usando uma ou mais variáveis de tipo e um ou mais métodos que usam variáveis de tipo

Exemplo: o tipo `java.util.List<E>` é um tipo genérico

- uma lista que armazena elementos de algum tipo representado pelo placeholder `E`.
- tem um método `add()` que possui um argumento do tipo `E`
- tem um método `get()` que retorna um valor do tipo `E`

➡ Para usar um tipo genérico, especificamos o tipo atual, produzindo um tipo parametrizado. Exemplo: `List<String>`





Tipos Genéricos

⇒ Em coleções: permite definir que uma coleção é restrita a conter somente um tipo abstrato de dado particular.

```
Lista l = new ListaArray();  
l.insere (new Integer(10));  
Integer x = (Integer) l.retornaIterator().next();
```

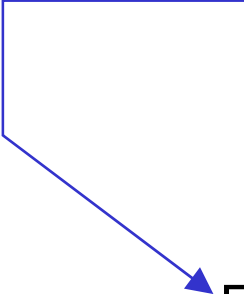


```
Lista<Integer> l = new ListaArray<Integer>();  
l.insere (new Integer(10));  
Integer x = l.retornaIterator().next();
```





Tipos Genéricos



```
Lista<Integer> l = new ListaArray<Integer>();  
l.insere (new Integer(10));  
Integer x = l.retornalterator().next();
```

Especifica uma lista de Integer e não uma lista qualquer.

- Lista é uma interface genérica que recebe como parâmetro um tipo, ex. Integer.
- O parâmetro com o tipo também é especificado durante a criação da lista de objetos.



Tipos Genéricos

➡ Não é obrigatório especificar os parâmetros de tipo para usar as classes de coleções do Java.

```
List l = new ArrayList();  
l.add("gato");  
l.add(new Integer(123));  
Object o = l.get(0);
```

- Este código funciona perfeitamente no Java 1.4.
- No Java 5.0, a compilação não gera nenhum erro, mas o compilador avisa que operações não verificadas ou não seguras estão sendo usadas.





Tipos Genéricos

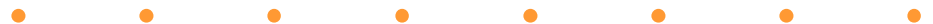
➡ O código a seguir compila no Java 5.0 sem nenhuma mensagem e permite adicionar objetos de vários tipos na lista.

```
List<Object> l = new ArrayList<Object>();
```

```
l.add("macaco");
```

```
l.add(123);
```

```
Object o = l.get(0);
```





Wildcard (Coringa)

➡ Suponha que precisemos escrever um método para mostrar os elementos de um List.

- No Java 1.4, teríamos o seguinte código:

```
public static void printList(PrintWriter out, List list) {  
    for(int i=0, n=list.size(); i < n; i++) {  
        if (i > 0) out.print(", ");  
        out.print(list.get(i).toString());  
    }  
}
```

No Java 5.0, List é um tipo genérico e a compilação iria gerar alguns avisos.



Wildcard (Coringa)

- Se usarmos o Object como tipo de parâmetro

```
public static void printList(PrintWriter out, List<Object> list) {  
    for(int i=0, n=list.size(); i < n; i++) {  
        if (i > 0) out.print(", ");  
        out.print(list.get(i).toString());  
    }  
}
```

Este código compila sem avisos mas somente listas declaradas como `List<Object>` podem ser passadas como parâmetros. `List<String>` and `List<Integer>` não podem ser atribuídas a um `List<Object>`.

Wildcard (Coringa)

- Usamos um wildcard (coringa) como o parâmetro de tipo

```
public static void printList(PrintWriter out, List<?> list) {  
    for(int i=0, n=list.size(); i < n; i++) {  
        if (i > 0) out.print(", ");  
        Object o = list.get(i);  
        out.print(o.toString());  
    }  
}
```

- ➡ O coringa ? representa um tipo desconhecido e o tipo List<?> é lido como um "List de desconhecido."
- ➡ Regra geral: se um tipo é genérico e não sabemos o valor da variável tipo, usamos o coringa ? ao invés de usar um tipo básico.



Wildcard (Coringa)

➡ Suponha que precisemos escrever um método `somaList()` para calcular a soma de uma lista de objetos do tipo `Number`.

- Se usarmos um `List` não tipado não obteremos verificação de tipo e temos que lidar com os avisos do compilador.
- Se usarmos um `List<Number>` não poderemos chamar o método para um `List<Integer>` ou `List<Double>`.



Wildcard (Coringa)

- Se usarmos um coringa, não teremos a verificação de tipo desejada, pois teremos que confiar que o método será chamado dentro de um List cujo parâmetro é Number ou uma Subclasse e nunca, por exemplo, um String.

```
public static double somaList (List<?> lista) {  
    double total = 0.0;  
    for(Object o : lista) {  
        Number n = (Number) o; // precisa do cast e pode falhar  
        total += n.doubleValue();  
    }  
    return total;  
}
```



Wildcard (Coringa)

- Podemos usar um coringa restrito (bounded wildcard) que define que o parâmetro tipo da List é um tipo desconhecido que é Number ou uma subclasse de Number.

```
public static double somaList (List<? extends Number> lista) {  
    double total = 0.0;  
    for(Number n : list)  
        total += n.doubleValue();  
    return total;  
}
```

➡ O tipo List<? extends Number> pode ser lido como “Lista de descendente desconhecido de Number”. E Number é considerado um descendente dele próprio.





Vantagens dos Tipos Genéricos

O uso de tipos genéricos permite que:

➡ o compilador proporcione tipagem forte aumentando a segurança dos programas.

Exemplo: evita adicionar um `String[]` a um `List` que deve conter somente objetos `String`.

➡ o compilador realize alguns “castings” automaticamente.

Exemplo: o compilador sabe que o método `get()` de um `List<String>` retorna um objeto `String`. Assim, não é necessário fazer o casting do valor retornado de `Object` para `String`.





Declaração das Interfaces

```
public interface Lista <E>{  
    public void insere (E elemento, int posicao);  
    public void insere (E elemento);  
    public boolean contem (E elemento);  
    ...  
    public Iterator<E> retornaliterator();  
}
```





Declaração das Classes

```
public class ListaArray<E> implements Lista <E>{
```

```
    private E[] elementos;
```

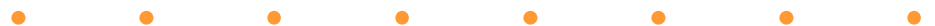
```
    private int numElementos;
```

```
    public ListaArray (){
```

```
        this.elementos = (E[]) new Object[10];
```

```
    }
```

➡ Java permite que um array seja definido com um tipo parametrizado, mas não permite que um tipo parametrizado seja usado para criar um array novo.





Definição dos Métodos das Classes

```
public void insere (E elemento){  
    if (this.numElementos == this.elementos.length) {  
        E[] novoArray = (E[]) new Object[2*this.elementos.length];  
        System.arraycopy(this.elementos,0,novoArray,0,this.elementos.length);  
        this.elementos = novoArray;  
    }  
    this.elementos[this.numElementos] = elemento;  
    this.numElementos++;  
}
```

