Programação Orientada por Objectos

Introdução à Coleções e Tipos Genéricos

Prof. Rui César das Neves, Prof. José Cordeiro

Departamento de Sistemas e Informática

Escola Superior de Tecnologia de Setúbal – Instituto Politécnico de Setúbal

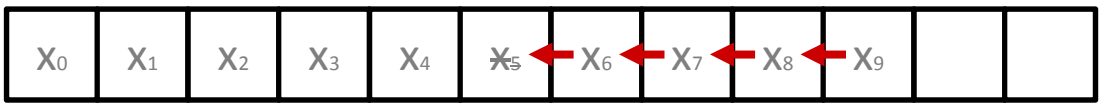
2014/2015

Sumário

- Criação de estruturas que permitem armazenar um número variável de elementos
- Tentativa de generalizar o mecanismo através do uso de **`Object`**
- Tipos e Métodos Genéricos
  - Como definir
  - Convenções de nomes
  - Sua utilização
  - Múltiplos Tipos Parametrizados
  - Restrição de tipos
  - Uso de *wildcard* (**?**)
  - Limitações do uso de genéricos

Criação de Lista de Nomes

- **Problema:**
  - Criar uma estrutura que permita guardar nomes de pessoas (**`String`**) sem ser necessário que tenha um tamanho fixo (como acontece nos **`arrays`**). Pretende-se que a estrutura "cresça" à medida que se vão introduzindo mais valores.
- **Solução:**
  - Criar uma classe que contenha um atributo com os diversos nomes introduzidos (será um **`array`**) e que disponibilize métodos para introduzir, remover e alterar os elementos. Estes métodos serão responsáveis por fazer a gestão do espaço ocupado pelo **`array`** (existe também um atributo que indica o espaço já ocupado):
    - Ao introduzir um valor, se não houver espaço, é criado um **`array`** novo (que será "armazenado" no atributo da classe, posteriormente), maior que o existente e todos os valores já guardados são copiados para ele, acrescentando, no final, o novo valor.
    - Ao retirar um elemento, é feita a cópia dos elementos nas posições seguintes para a posição anterior, "preenchendo-se", assim, o lugar anteriormente ocupado:



Lista de Nomes – Atributos e Construtor

```
public class ListaNomes {

  private static final int TAMANHO_OMISSAO = 5;
  private String[] valores;
  private int quantos;

  public ListaNomes() {
    valores = new String[TAMANHO_OMISSAO];
    quantos = 0;
  }
```

Lista de Nomes – Adicionar Elemento

```
//Nome em inglês por compatibilidade com a implementação
//standard disponibilizada no Java
public void add(String elemento) {
  if (quantos == valores.length) {
    String[] novos = new String[valores.length * 2];
    for (int i = 0; i < valores.length; i++) {
      novos[i] = valores[i];
    }
    valores = novos;
  }
  valores[quantos++] = elemento;
}
```

Lista de Nomes – Remover Elemento

```
public boolean remove(int posicao) {
  if ((posicao >= 0) && (posicao < quantos)) {
    for (int i = posicao; i < quantos - 1; i++) {
      valores[i] = valores[i + 1];
    }
    quantos--;
    return true;
  } else {
    return false;
  }
}
```

Lista de Nomes – Obter e alterar por posição

```
public String get(int posicao) {
  if ((posicao >= 0) && (posicao < quantos)) {
    return valores[posicao];
  } else {
    return null;
  }
}

public boolean set(int posicao, String elemento) {
  if ((posicao >= 0) && (posicao < quantos)) {
    valores[posicao] = elemento;
    return true;
  } else {
    return false;
  }
}
```

Lista de Nomes – Obter informações

```
public int size() {
  return quantos;
}

//Desnecessário para a interação com a classe.
//Serve apenas para compreendermos o que se está a
//passar "internamente"
public int capacity() {
  return valores.length;
}
```

Lista de Nomes – Visualização

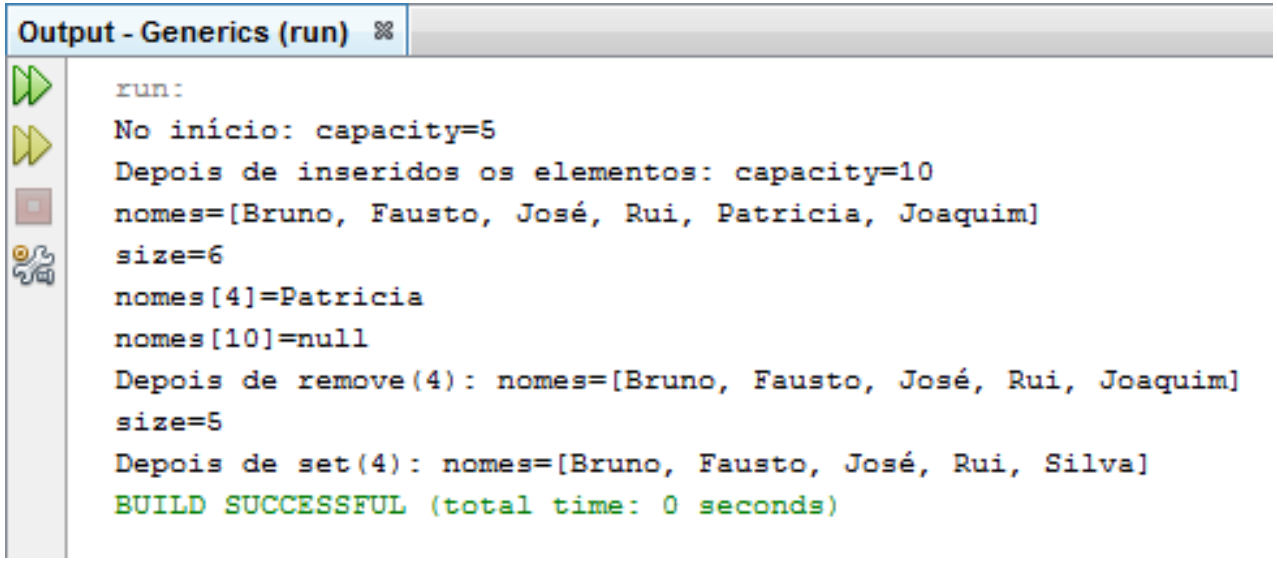
```
@Override
public String toString() {
  String resultado = "[";
  boolean primeiro = true;
  for (int i = 0; i < quantos; i++) {
    if (primeiro) {
      primeiro = false;
    } else {
      resultado += ", ";
    }
    resultado += valores[i];
  }
  resultado += "]";
  return resultado;
}
}
```

Lista de Nomes – Utilização

```
public static void main(String[] args) {
  ListaNomes nomes = new ListaNomes();
  System.out.format("No início: capacity=%d\n", nomes.capacity());
  nomes.add("Bruno");
  nomes.add("Fausto");
  nomes.add("José");
  nomes.add("Rui");
  nomes.add("Patricia");
  nomes.add("Joaquim");
  System.out.format("Depois de inseridos os elementos: capacity=%d\n",
                                                        nomes.capacity());
  System.out.format("nomes=%s\n", nomes);
  System.out.format("size=%d\n", nomes.size());
  System.out.format("nomes[%d]=%s\n", 4, nomes.get(4));
  System.out.format("nomes[%d]=%s\n", 10, nomes.get(10));
```

Lista de Nomes – Utilização

```
        nomes.remove(4);
        System.out.format("Depois de remove(4): nomes=%s\n", nomes);
        System.out.format("size=%d\n", nomes.size());
        nomes.set(4, "Silva");
        System.out.format("Depois de set(4): nomes=%s\n", nomes);
    }
```



Equilíbrio Flexibilidade – Performance

- Através desta abordagem é possível utilizar um mecanismo de armazenamento de informação mais flexível do que um simples **`array`**.
- Tal é conseguido através de um custo de *performance* em espaço e tempo:
  - Há espaço desperdiçado sempre que existam posições vazias no **`array`** (agravado pelo facto do **`remove`** não reduzir o espaço ocupado).
  - Sempre que não há espaço suficiente no **`array`** que armazena os valores é criado um novo **`array`** (com mais espaço): nesse instante existe uma duplicação (na realidade mais do que duplicação) do espaço ocupado.
  - A cópia da informação do **`array`** original para o novo **`array`** pode demorar "algum tempo", tanto mais quanto a sua dimensão.

Equilíbrio Flexibilidade – Performance

- Quando se consegue saber à *priori* o número correto de elementos que pretendemos armazenar é aconselhável utilizar um **`array`**.
- Quando não temos indicação do espaço necessário devemos recorrer a este tipo de estruturas: as "coleções" (**`collections`**). As coleções são classes que o Java fornece e que permitem guardar vários elementos (como na classe **`ListaNomes`**).
- Nas implementações *standard* das coleções, fornecidas pelo Java, é possível indicar (no construtor) a dimensão prevista (poderá crescer) para o **`array`** interno, reduzindo-se o número de vezes que é necessário criar um novo **`array`**.

Replicação do Funcionamento

- Replicar este mecanismo para outros tipos de informação: criar lista de pessoas. Primeiro é necessário definir a classe **Pessoa**:

```
public class Pessoa {
  private int idade;
  private String nome;
  public Pessoa(String nome,
                int idade) {
    this.nome = nome;
    this.idade = idade;
  }
  public int getIdade() {
    return idade;
  }
  public void setIdade(int idade){
    this.idade = idade;
  }
  public String getNome() {
    return nome;
  }
  public void setNome(String nome){
    this.nome = nome;
  }
  @Override
  public String toString() {
    return String.format("%s (%d
anos)", nome, idade);
 }
}
```

ListaPessoas

- Classe **ListaPessoas** (idêntica a **ListaNomes**, diferenças a vermelho, mas com **String** substituído por **Pessoa**, atenção ao **toString**):

```
public class ListaPessoas {
  private static final int TAMANHO_OMISSAO = 5;
  private Pessoa[] valores;
  private int quantos;

  public ListaPessoas() {
    valores = new Pessoa[TAMANHO_OMISSAO];
    quantos = 0;
  }

  public void add(Pessoa elemento) {
    if (quantos == valores.length) {
      Pessoa[] novos = new Pessoa[valores.length * 2];
      for (int i = 0; i < valores.length; i++) {
        novos[i] = valores[i];
      }
      valores = novos;
    }
    valores[quantos++] = elemento;
  }
```

ListaPessoas

```
public boolean remove(int posicao) {
  if ((posicao >= 0) && (posicao < quantos)) {
    for (int i = posicao; i < quantos - 1; i++) {
      valores[i] = valores[i + 1];
    }
    quantos--;
    return true;
  } else {
    return false;
  }
}

public Pessoa get(int posicao) {
  if ((posicao >= 0) && (posicao < quantos)) {
    return valores[posicao];
  } else {
    return null;
  }
}
```

ListaPessoas

```
  public boolean set(int posicao, Pessoa elemento) {
    if ((posicao >= 0) && (posicao < quantos)) {
      valores[posicao] = elemento;
      return true;
    } else {
      return false;
    }
  }
  public int size() {
    return quantos;
  }
  public int capacity() {
    return valores.length;
  }
  @Override
  public String toString() {
    String resultado = "[";
    ...
    resultado += "]";
    return resultado;
  }
}
```

Verificar o Funcionamento com Herança

- A **ListaPessoas** pode receber (pelo principio de substituição), objetos de classes filhas. Criando a classe **Funcionario** é possível colocá-los na lista:

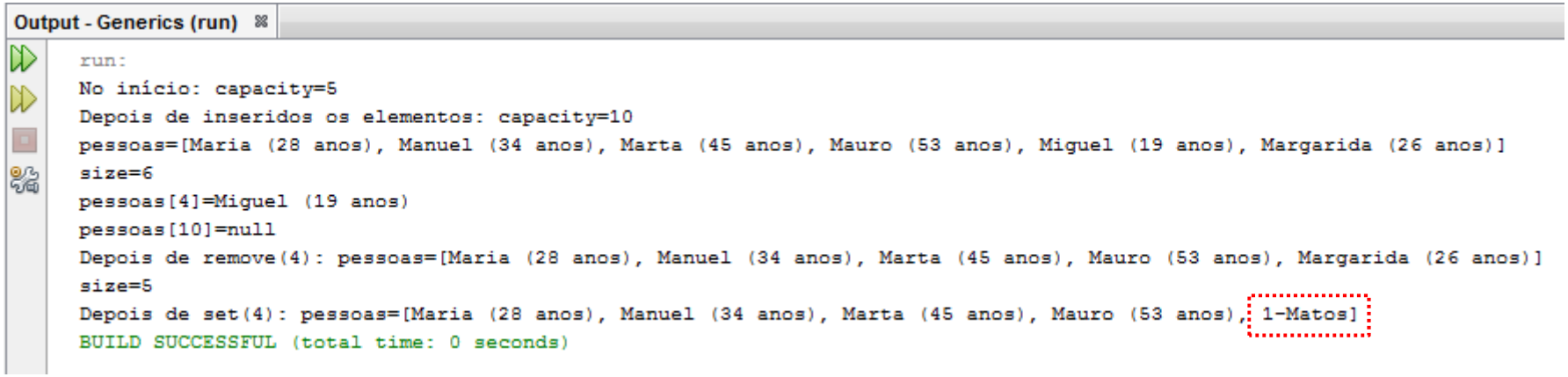
```
public class Funcionario extends Pessoa {
    private static int quantos = 0;
    private int codigo;

    public Funcionario(String nome, int idade) {
        super(nome, idade);
        codigo = ++quantos;
    }

    public int getCodigo() {
        return codigo;
    }

    @Override
    public String toString() {
        return codigo + "-" + getNome();
    }
}
```

ListaPessoas - Utilização

```
public static void main(String[] args) {
  ListaPessoas pessoas = new ListaPessoas();
  System.out.format("No início: capacity=%d\n", pessoas.capacity());
  pessoas.add(new Pessoa("Maria", 28));
  pessoas.add(new Pessoa("Manuel", 34));
  pessoas.add(new Pessoa("Marta", 45));
  pessoas.add(new Pessoa("Mauro", 53));
  pessoas.add(new Pessoa("Miguel", 19));
  pessoas.add(new Pessoa("Margarida", 26));
  System.out.format("Depois de inseridos os elementos: capacity=%d\n",
                                                      pessoas.capacity());
  System.out.format("pessoas=%s\n", pessoas);
  System.out.format("size=%d\n", pessoas.size());
  System.out.format("pessoas[%d]=%s\n", 4, pessoas.get(4));
  System.out.format("pessoas[%d]=%s\n", 10, pessoas.get(10));
```

ListaPessoas - Utilização

```
    pessoas.remove(4);
    System.out.format("Depois de remove(4): pessoas=%s\n", pessoas);
    System.out.format("size=%d\n", pessoas.size());
    pessoas.set(4, new Funcionario("Matos", 47));
    System.out.format("Depois de set(4): pessoas=%s\n", pessoas);
}
```



Duplicação de Código

- Consegue-se replicar o comportamento (estrutura com tamanho variável) mas à custa de uma enorme duplicação de código: há métodos (**`remove`**, **`size`**, **`capacity`**, **`toString`**) integralmente repetidos.
- Estamos a violar um dos princípios da boa programação e objetivo da Programação Orientada a Objetos: não duplicar código.
- Tentar aplicar algum dos mecanismos disponibilizados pela POO para resolver o problema: utilizar a herança, por generalização

Duplicação de Código

- Através de generalização poderíamos tentar implementar uma super classe, comum a **`ListaNomes`** e **`ListaPessoas`** que contivesse os métodos comuns e apenas definir nas classes filhas as partes distintas.
- O ganho seria pouco pois o **`array`**, por ter elementos de tipos distintos, teria que ficar nas classes filhas. Além disso, a maioria do corpo dos métodos das classes filhas continuaria a ser repetido.
- A solução passa por encontrarmos um tipo comum, a todos os possíveis, para representar o tipo do elemento: **`Object`**

Classe com comportamento comum – ListaObject

```
public class ListaObject {

  private static final int TAMANHO_OMISSAO = 5;
  private Object[] valores;
  private int quantos;

  public ListaObject() {
    valores = new Object[TAMANHO_OMISSAO];
    quantos = 0;
  }

  public void add(Object elemento) {
    if (quantos == valores.length) {
      Object[] novos = new Object[valores.length * 2];
      for (int i = 0; i < valores.length; i++) {
        novos[i] = valores[i];
      }
      valores = novos;
    }
    valores[quantos++] = elemento;
  }
```

ListaObject

```
public boolean remove(int posicao) {
  if ((posicao >= 0) && (posicao < quantos)) {
    for (int i = posicao; i < quantos - 1; i++) {
      valores[i] = valores[i + 1];
    }
    quantos--;
    return true;
  } else {
    return false;
  }
}
public Object get(int posicao) {
  if ((posicao >= 0) && (posicao < quantos)) {
    return valores[posicao];
  } else {
    return null;
  }
}
```

ListaObject

```
  public boolean set(int posicao, Object elemento) {
    if ((posicao >= 0) && (posicao < quantos)) {
      valores[posicao] = elemento;
      return true;
    } else {
      return false;
    }
  }
  public int size() {
    return quantos;
  }
  public int capacity() {
    return valores.length;
  }
  @Override
  public String toString() {
    String resultado = "[";
    ...
    resultado += "]";
    return resultado;
  }
}
```

Classe genérica

- O **`ListaObject`** poderá ser utilizado para qualquer tipo de situação, eliminando-se a necessidade de implementar classes para tipos específicos:
  - `ListaObject nomes = new ListaObject();`
  - `ListaObject pessoas = new ListaObject();`

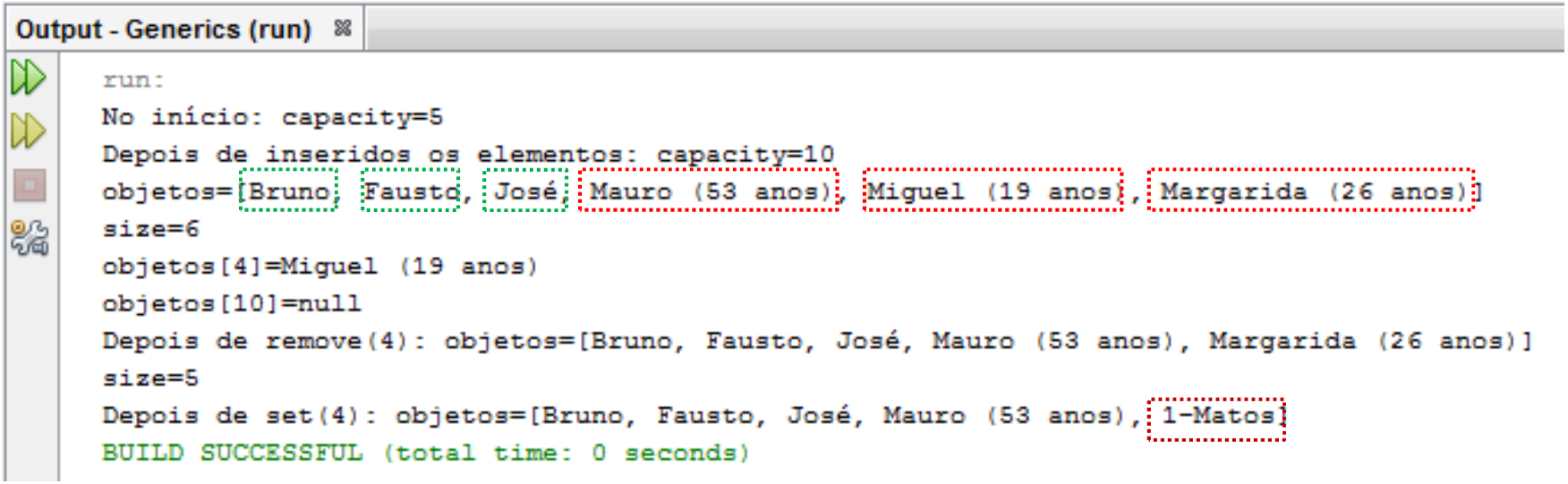
- Infelizmente surge um problema: não temos controlo sobre o que é introduzido na lista – podemos misturar objetos de tipos distintos

ListaObject - Utilização

```
public static void main(String[] args) {
  ListaObject objetos = new ListaObject();
  System.out.format("No início: capacity=%d\n", objetos.capacity());
  objetos.add("Bruno");
  objetos.add("Fausto");
  objetos.add("José");
  objetos.add(new Pessoa("Mauro", 53));
  objetos.add(new Pessoa("Miguel", 19));
  objetos.add(new Pessoa("Margarida", 26));
  System.out.format("Depois de inseridos os elementos: capacity=%d\n",
                                                    objetos.capacity());
  System.out.format("nomes=%s\n", objetos);
  System.out.format("size=%d\n", objetos.size());
  System.out.format("nomes[%d]=%s\n", 4, objetos.get(4));
  System.out.format("nomes[%d]=%s\n", 10, objetos.get(10));
```

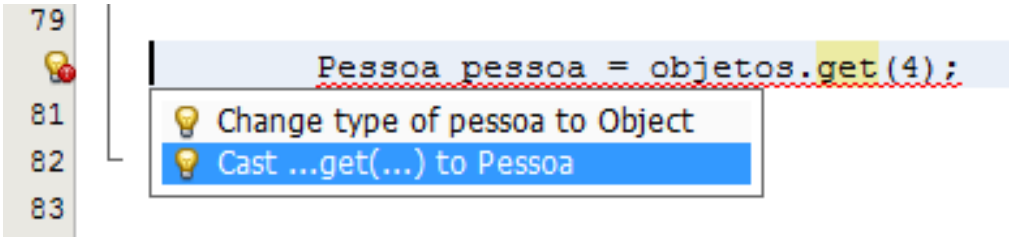
ListaObject - Utilização

```
  objetos.remove(4);
  System.out.format("Depois de remove(4): nomes=%s\n", objetos);
  System.out.format("size=%d\n", objetos.size());
  objetos.set(4, new Funcionario("Matos", 47));
  System.out.format("Depois de set(4): nomes=%s\n", objetos);
}
```



Problemas

- Perdemos a ajuda do compilador no controle de tipo de dados
- Tudo se complica, ainda mais, quando tentamos atribuir (ou utilizar) os valores da lista a variáveis do tipo específico:




```
Pessoa pessoa = (Pessoa)objetos.get(4);
```

Pretendia-se uma solução mais versátil

- O que se pretende é ter a possibilidade de definir código onde o tipo dos atributos, parâmetros e/ou variáveis pudesse ser modificado no momento da sua utilização. Na definição apenas seria indicada uma "espécie de variável" que permitisse referir/sinalizar o tipo de dados. No momento da utilização indicava-se o tipo concreto pretendido.
- Uma abordagem em tudo semelhante ao uso dos parâmetros nos métodos que permitem generalizar o algoritmo implementado. No momento da execução do método indica-se, através dos argumentos, os valores a serem utilizados "em substituição" dos parâmetros colocados na definição.

Solução através de Tipo Genérico

- Na versão J2SE 5.0 foi introduzido o conceito de Tipo Genérico (**`Generic`**) ou Tipo Parametrizado (**`type parameters`**).
- Na definição de uma classe, ou de um método, é possível indicar (entre **< >**) um parâmetro que representa um tipo de dados. Este parâmetro será utilizado nos locais onde se colocaria o tipo de dados (indicação do tipo dos atributos, na lista de parâmetros dos métodos, na declaração de variáveis):

```
public class Lista<E> {
...
}
public static <T> void informacao(T t) {
  System.out.println("T: " + t.getClass().getName());
}
```

Convenção de nomes

- Por convenção, os **`type parameter`** são representados normalmente apenas por uma letra, que indica o que o tipo representa:
  - E - Element (utilizado regularmente no Java Collections Framework)
  - K - Key
  - N - Number
  - T - Type
  - V - Value
  - S,U,V etc. – 2º, 3º, 4º tipos

Classe Lista<E>

```
public class Lista<E> {
  private static final int TAMANHO_OMISSAO = 5;
  private E[] valores;
  private int quantos;

  @SuppressWarnings("unchecked") //necessário devido ao cast da tabela
  public Lista() {
    //valores = new E[TAMANHO_OMISSAO]; Não é permitido pelo compilador de Java
    valores = (E[]) new Object[TAMANHO_OMISSAO];
    quantos = 0;
  }

  @SuppressWarnings("unchecked") //necessário devido ao cast da tabela
  public void add(E elemento) {
    if (quantos == valores.length) {
      Object[] novos = new Object[valores.length * 2];
      for (int i = 0; i < valores.length; i++) {
        novos[i] = valores[i];
      }
      valores = (E[]) novos;
    }
    valores[quantos++] = elemento;
  }
```

Classe Lista<E>

```
public boolean remove(int posicao) {
  if ((posicao >= 0) && (posicao < quantos)) {
    for (int i = posicao; i < quantos - 1; i++) {
      valores[i] = valores[i + 1];
    }
    quantos--;
    return true;
  } else {
    return false;
  }
}

public E get(int posicao) {
  if ((posicao >= 0) && (posicao < quantos)) {
    return valores[posicao];
  } else {
    return null;
  }
}
```

Classe Lista<E>

```
  public boolean set(int posicao, E elemento) {
    if ((posicao >= 0) && (posicao < quantos)) {
      valores[posicao] = elemento;
      return true;
    } else {
      return false;
    }
  }
  public int size() {
    return quantos;
  }
  public int capacity() {
    return valores.length;
  }
  @Override
  public String toString() {
    String resultado = "[";
    ...
    resultado += "]";
    return resultado;
  }
}
```

Classe Lista<E> - Utilização

- No momento da utilização da classe indica-se o tipo pretendido:
  - `Lista<String> nomes = new Lista<String>();`
  - `Lista<Pessoa> pessoas = new Lista<Pessoa>();`
- A partir da Java SE 7 é possível omitir a indicação do tipo, sempre que o compilador consiga determiná-lo. Utilizando-se a chamada notação *diamante* `<>`:
  - `Lista<String> nomes = new Lista<>();`
  - `Lista<Pessoa> pessoas = new Lista<>();`
- Na utilização dos métodos da classe não é necessário fazer qualquer modificação. Continua-se a poder utilizar elementos de classes filhas:
  - `pessoas.set(4, new Funcionario("Matos", 47));`

Métodos genéricos - Utilização

- A chamada a um método genérico deve indicar o tipo de dados a utilizar:

```
public class Generics {
   ...
   public static <T> void informacao(T t) {
     System.out.println("T: " + t.getClass().getName());
   }
 ...
}

public static void main(String[] args) {
   Lista<String> nomes = new Lista<>();
   Generics.<Lista<String>>informacao(nomes);
}
```

- Podemos omitir caso o compilador consiga determinar o tipo:

```
Generics.informacao(nomes);
```

Múltiplos Tipos Parametrizados

- Podem ser indicados mais do que um **`type parameter`**:

```
public class Associacao<K, V> {
    private K chave;
    private V valor;

    public Associacao(K chave, V valor) {
        this.chave = chave;
        this.valor = valor;
    }

    public K getChave() { return chave; }
    public void setChave(K chave) { this.chave = chave; }
    public V getValor() { return valor; }
    public void setValor(V valor) { this.valor = valor; }
}
```

Limitar o Tipo Parametrizado

- É possível restringir o **`type parameter`** a um determinado tipo ou seus descendentes (através do uso de **`extends`**):

```
public class Associacao<K extends Number, V extends Pessoa> {

    private K chave;
    private V valor;
    ...
}
```

- Desta forma podemos utilizar os métodos conhecidos do tipo:

```
@Override
public String toString() {
    return chave + "-" + valor.getNome();
}
```

Nota: **`Number`** tem como descendentes **`AtomicInteger`**, **`AtomicLong`**, **`BigDecimal`**, **`BigInteger`**, **`Byte`**, **`Double`**, **`Float`**, **`Integer`**, **`Long`**, **`Short`** ou outros que sejam definidos

Limitar o Tipo Parametrizado

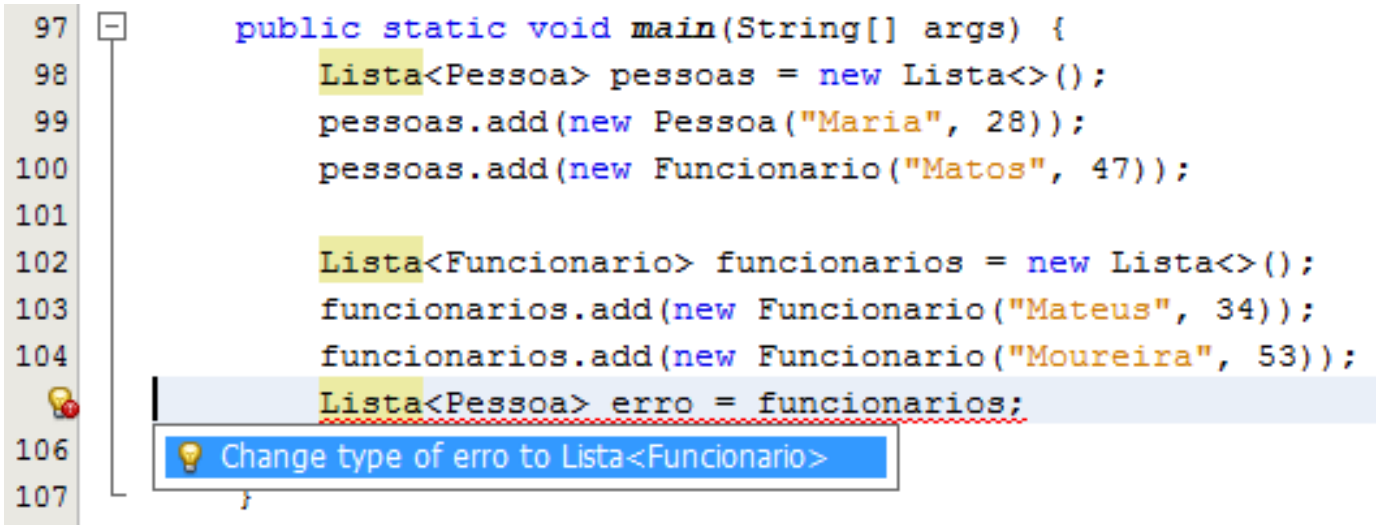
- A restrição pode ser feita de forma múltipla:

```
public class A {
   ...
}
public interface B {
   ...
}
public interface C {
   ...
}

public class D <T extends A & B & C> {
   ...
}
```

- A classe deverá ser indicada em primeiro lugar

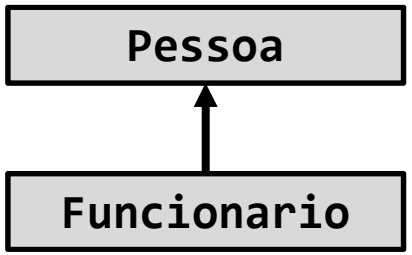
Erro comum na perceção da Herança

- Apesar de se poder atribuir a uma lista de pessoas elementos que são de classes filhas (ex.: **`Funcionario`**). Não existe nenhuma relação entre **`Lista<Pessoa>`** e **`Lista<Funcionario>`**, não sendo permitido a sua "mistura":

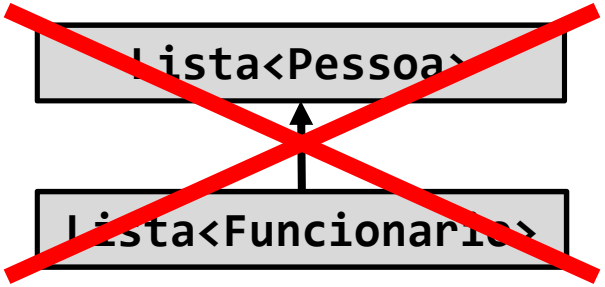


Erro comum na perceção da Herança

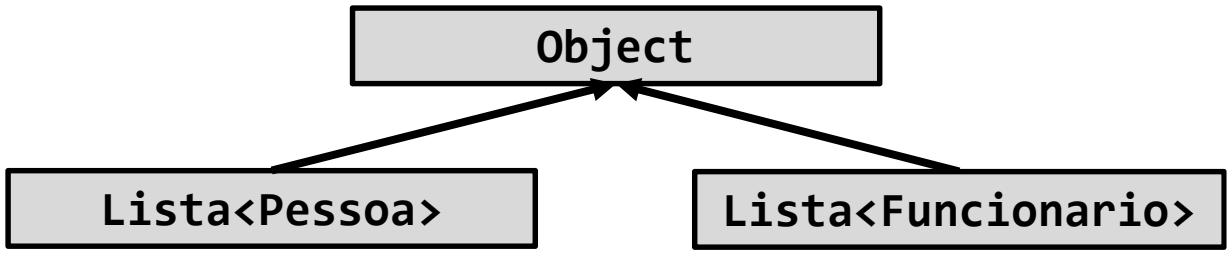
- **`Funcionario`** herda de **`Pessoa`**:



- Mas **`Lista<Funcionario>`** não herda de **`Lista<Pessoa>`**:



- Ambas **`Lista<Funcionario>`** e **`Lista<Pessoa>`** herdam de **`Object`**:



Uso de ? (*wildcard*)

- O problema fica resolvido através da indicação de que o tipo de elementos da lista pode ser qualquer tipo (indicado através de **?**) que herde de **Pessoa**:

```
//Lista<Pessoa> erro = funcionarios;
Lista<? extends Pessoa> naoErro = funcionarios;
```

- **? extends Pessoa** indica qualquer tipo de herde de **Pessoa** (inclusive). Desta forma indicamos não um tipo de lista mas sim uma "família de tipos de listas" que estão relacionados pela relação de herança dos seus elementos.

- Também é possível a notação **? super T**. Onde seriam aceites elementos que fossem super classes do tipo **T** (inclusive).

Limitações ao uso de Tipos Parametrizados

- Não é possível utilizar um tipo primitivo como substituição de um **type parameter**:

```
Lista<int> exemplo = new Lista<>(); //ERRO!
```

- Devemos utilizar apenas tipos não primitivos:

```
Lista<Integer> exemplo = new Lista<>(); //OK!
```

- Não é possível criar instâncias de um **type parameter**:

```
new E(); //new E[TAMANHO_OMISSAO];
```

- Não podem ser declarados atributos **static** cujo tipo sejam um **type parameter**:

```
private static E exemplo;
```

Limitações ao uso de Tipos Parametrizados

- Não é possível fazer *cast* com um **`type parameter`**:

```
exemplo = (E)qualquer;
```

- Não é possível fazer **`instanceof`** com um **`type parameter`**:

```
if (exemplo instanceof E)
```

- Não é possível criar **`arrays`** de Tipos Parametrizados:

```
Lista<Integer>[] arrayListas = new Lista<Integer>[2];
```

- Tipos Parametrizados não podem ser criados para fazer **`throw`** ou **`catch`**.

- Não pode ser feito polimorfismo de métodos que diferem apenas em Tipos Parametrizados:

```
public class Errado {
    public void print(Lista<String> listaString) { }
    public void print(Lista<Integer> listaInteger) { }
}
```

Resumindo

- O Java Collections Framework (JCF), que iremos estudar em profundidade nas próximas aulas, disponibiliza classes que permitem armazenar um número variável de elementos
- O uso de Tipos Parametrizados permite definir classes e/ou métodos genéricos que envolvem tipos que apenas serão concretizados no momento da utilização
- Os Tipos parametrizados são, normalmente, representados por uma letra que indica o que o tipo representa.
- É possível omitir na utilização de métodos o tipos envolvido desde que o compilador o consiga determinar (poderá ser necessário usar a notação **<>**)
- Podem ser utilizados múltiplos tipos parametrizados e podemos limitar a gama de tipos a utilizar
- Pode ser necessário recorrer ao uso de **?** para indicar relações entre tipos parametrizáveis.
- Existem algumas situações em que não é possível usar tipos parametrizáveis.

Leitura Complementar

- Capítulo 8.3
  - Páginas 257 a 259

- http://docs.oracle.com/javase/tutorial/java/generics/index.html