Estruturas de Dados

Genéricos

2025/2026

Métodos Genéricos

- Pretende-se uma método que imprime o contéudo de um *array*:

```
public static void imprime(String [] m)
{
for(String a : m)
   System.out.println(a);
}

public static void main (String args[]){
String m[]={"José","António"};
imprime(m);
}
```

Métodos Genéricos

- A função anterior só funciona para *arrays* de *Strings*.
- No entanto, pode ser *generalizada* para aceitar *arrays* com qualquer tipo de elementos

```
public static <T> void imprime(T [] m)
{
for(T a: m)
   System.out.println(a);
}

public static void main (String args[]){
String m[]={"José","António"};
Integer n[]={2, 3};
imprime(m);
imprime(n);
}
```

Métodos Genéricos

```
public static <T> T escolheAleat(T a, T b)
{
T ret;
if(Math.random()>0.5)
     ret=a;
   else
     ret=b;
return ret;
}
```

Classes Genéricas

- O conceito de genéricos também pode ser aplicado a classes.

```
public class Par <T,S> {
T primeiro;
S segundo;
public Par(T a, S b){
primeiro=a;
segundo=b;
}
public T getPrimeiro(){return primeiro;}
public S getSegundo(){return segundo;}

public setPrimeiro(T a){primeiro=a;}
public setSegundo(S b){segundo=b;}
}
```

Versão sem Genéricos

- No entanto, também é possível obter um comportamento similar através da utilização da classe *Object*. Sendo assim, qual a vantagem de usar genéricos?

```
public class Par{
Object primeiro;
Object segundo;
public Par(Object a, Object b){
primeiro=a;
segundo=b;
}
public Object getPrimeiro(){return primeiro;}
public Object getSegundo(){return segundo;}

public setPrimeiro(Object a){primeiro=a;}
public setSegundo(Object b){segundo=b;}
}
```

Comparação – Versão Sem Genéricos

```
public static void main(String args[])
{
Par p1=new Par(“Par1”,1);
Par p2=new Par(2,”Par2”);

Integer i1=(Integer)(p1.getSegundo());  //ok
Integer i2=(Integer) (p2.getSegundo());
  // erro de execução! - ClassCastException
}
```

Comparação – Versão Genérica

```
public static void main(String args[])
{
//Java 7+
Par<String, Integer> p1=new Par<>(“Par1”,1);

// Java 5,6
Par<Integer, String> p2=new Par<Integer,String>(2,”Par2”);

Integer i1=p1.getSegundo();  //ok
Integer i2=p2.getSegundo();  // erro de compilação!
}
```

A utilização de genéricos, em vez da classe *Object*, possibilita a detecção durante a *compilação* de erros de tipo.

Wildcards

- É possível impor restrições ao parâmetros formais. Por exemplo, neste caso, T:

```
public static <T extends Number> boolean maior (T p1, T p2){
      return p1.doubleValue()>p2.doubleValue();
}
public static void main(String args[]){
if(maior(33,4))
   System.out.println("maior");
boolean m=maior("ola","adeus"); //erro de compilação
   //String não estende Number.
}
```

Wildcards

- As *wildcards* podem ser especificadas da seguinte forma:
  - **X extends Y** :
    - X é igual a ou estende a classe Y (ou implementa interface Y)
  - **X super Y**:
    - Y é igual a ou estende a classe X (ou implementa interface X)
  - X ou Y podem ser substituidos por '?', o que significa "qualquer classe".

Wildcards - Exemplo

- Os *wildcards* podem ser usados para substituir parâmetros formais que são usados uma única vez:
- Exemplo 1:
  - `public static <T,S> void f(par<T,S> p)`

  Pode ser escrito como
  - `public static void f(par<?,?> p)`
- Exemplo 2:
  - `public static <T, S extends T> void f(par<T,S> p)`
- Pode ser escrito como
  - `public static <T> void f(par<T, ? extends T> p)`

Suporte de Polimorfismo com Wildcards

```
public class Par <T,S> {
T primeiro;
S segundo;
public Par(T a, S b){
primeiro=a;
segundo=b;
}
public T getPrimeiro(){return primeiro;}
public S getSegundo(){return segundo;}

…

public void copia(Par<T,S> outroPar)
    {
    primeiro=outroPar.getPrimeiro();
    Segundo=outroPar.getSegundo();
    }
}
```

Suporte de Polimorfismo com Wildcards

```
class Rect{
…
};
class Quad extends Rect{
…
}
public static void main(String args[])
{
Quad q1=new Quad(1), q2=new Quad(2);
Rect r1=new Rect(2,3), r2= new Rect (3,4);
Par<Quad,Rect> p1=new Par<>(q1,r1);
Par<Quad,Rect> p2=new Par<>(q2,r2);
p2.copia(p1);  //ok
Par<Rect,Rect> p3=new Par<>(q1,r1);
Par<Rect,Rect> p4=new Par<>(q2,r2);
p3.copia(p4);   //ok
p3.copia(p1);   // erro de compilação:
//Par<Quad,Rect> não pode ser usado
// no lugar de Par<Rect,Rect>
}
```

Suporte de Polimorfismo com Wildcards

```
public class Par <T,S> {
T primeiro;
S segundo;
public Par(T a, S b){
primeiro=a;
segundo=b;
}
public T getPrimeiro(){return primeiro;}
public S getSegundo(){return segundo;}

…
public void copia
(Par<? extends T,? extends S> outroPar)
    {
    primeiro=outroPar.getPrimeiro();
    Segundo=outroPar.getSegundo();
    }
}
```

Suporte de Polimorfismo com Wildcards

```
class Rect{
…
};
class Quad extends Rect{
…
}
public static void main(String args[])
{
Quad q1=new Quad(1), q2=new Quad(2);
Rect r1=new Rect(2,3), r2= new Rect (3,4);
Par<Quad,Rect> p1=new Par<>(q1,r1);
Par<Quad,Rect> p2=new Par<>(q2,r2);
p2.copia(p1);  //ok
Par<Rect,Rect> p3=new Par<>(q1,r1);
Par<Rect,Rect> p4=new Par<>(q2,r2);
p3.copia(p4);   //ok
p3.copia(p1);   // ok!
}
```

Polimorfismo e Wildcards

Considere-se a seguinte hierarquia:

```
class Figura implements Comparable<Figura>{
int compareTo(Figura f){...}
}

Class Rect extends Figura{...}
...
```

É possível fazer isto:

```
Rect r=new Rect(2,3), q=new Rect(3,4);
r.compareTo(q);  //ok!
// é válido, apesar de Rect não ser Comparable<Rect>
// porque é Comparable<Figura>
```

Sendo assim, como usar *wildcards* para definir um método que recebe um parametro *Rect* e qualquer outro que possa ser comparado com ele?

Polimorfismo e Wildcards

Tentativa 1:

```
static int compara (Rect r, Comparable<Rect> c){
 //não funciona bem...  Qualquer figura é comparável
// com Rect
}
```

Tentativa 2:

```
static int compara (Rect r, Comparable<Figura> c){
// Já funciona melhor, mas infelizmente obriga-nos a saber
// que é exactamente a nível da classe Figura que se
// define a comparação..

//Se também for possível comparar uma outra classe não
//relacionada com figura (p.ex: Colorido) com um
//rectangulo, então não irá funcionar nesse caso
}
```

Como generalizar?

Polimorfismo e Wildcards

Basta que o objecto recebido por parâmetro seja comparável com uma classe antecessora de *Rect*!

```
static int compara (Rect r, Comparable<? super Rect> c){
   ...
  // Já funciona bem em todos os casos
  // pode reber um objecto que seja comparável com Rect,
  // ou então com Figura, ou então com Object...
}
```

Generalizando o *Rect* para que a função possa receber qualquer tipo de objectos

```
static <T> int compara (T r, Comparable<? super T> c){
return c.compareTo(r);  //ok!
}
```

Limitações dos Genéricos

- Quando um programa é compilado, os parâmetros dos genéricos são removidos, sendo utilizados *Objects* (ou outra superclasse adequada) no seu lugar.
  - *A informação de genéricos é usada apenas durante o processo de compilação para fazer verificação de tipos.*
  - *Este processo é designado por Type Erasure*
- Isto condiciona as operações possíveis com esses tipos.

É implementado como… →

```
public class Valor<T>{
T obj;
T f(T x) {…}
}
```

```
public class Valor{
Object obj;
Object f(Object x) {…}
}
```

Limitações dos Genéricos

- Sendo T um parâmetro formal, não é possível:
  - Criar arrays de classes parametrizadas:
    - `Par<X,Y> m[]=new Par<>[10];`
  - Criação de Objectos de tipo T:
    - `T obj=new T();`
  - Extensão:
    - `class <X> extends X`
  - Utilizar o tipo em membros ou métodos estáticos:
    - `private static T valor;`
    - `public static void f(T in);`
  - Obter dinamicamente informação sobre o tipo:
    - `O instanceof T`
  - Ser usado em enumerações:
    - `enum State<X> { …}`

Exercício

- Como definir uma classe ParComparável, sabendo que esta representa um Par cujos elementos são comparáveis. A classe ParComparável deve por sua vez ser também comparável: a comparação deve ser decidida através da comparação dos dois primeiros elementos de cada par, desempatando com a comparação dos segundos.

Exercício

- Passo 1:
- *"(...) classe ParComparável, sabendo que esta representa um Par cujos elementos são comparáveis. (...)"*

```
public class ParComparavel
<S extends Comparable<? super S>, T
  extends Comparable<? super T>>
extends Par<S,T>  {...}
```

Porque não apenas `S extends Comparable<S>`?

Tal como apresentado nos slides anteriores, isto permite que, por exemplo, a classe Rect definida anteriormente seja usada apesar de não ser Comparable<Rect>, mas sim Comparable<Figura>

Exercício

- “... A classe ParComparável deve por sua vez ser também comparável..”

```
public class ParComparavel
<S extends Comparable<? Super S>, T
  extends Comparable<? Super T>>
extends Par<S,T>
implements Comparable<ParComparavel<?
  extends S,? extends T>>{...}
```

Exercício

- "... a comparação deve ser decidida através da comparação dos dois primeiros elementos de cada par, desempatando com a comparação dos segundos."

```
class ParComparável ...{
 ...
 int compareTo(ParComparavel<? extends S,? extends T> p)
 {
 int compPrimeiro=
    p.getPrimeiro().compareTo(getPrimeiro());
 if(compPrimeiro==0)
    return p.getSegundo(getSegundo());
 else
    return compPrimeiro;
 }
 ...
}
```