Generics

prof. Angelo Loula

Exemplo

- Observe esse trecho:

```
public class Box {
        private Object object;

        public void add(Object object) {
            this.object = object;
        }

        public Object get() {
            return object;
        }
}
```

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

                // coloque somente Integer neste Box!
                Box integerBox = new Box();

                integerBox.add(new Integer(10));
                Integer someInteger = (Integer)integerBox.get();
                System.out.println(someInteger);
            }
```

Aula baseada em http://java.sun.com/docs/books/tutorial

Exemplo

- Alguém faz uma manutenção:

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

        // coloque somente Integer neste Box!
        Box integerBox = new Box();

        integerBox.add(new Integer(10));
        Integer someInteger = (Integer)integerBox.get();
        System.out.println(someInteger);

       // código modificado por outro programador
        integerBox.add("10"); // objeto do tipo String

        // e um terceiro programador
        Integer someInteger = (Integer)integerBox.get();
        System.out.println(someInteger);
    }
```

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

Generics

- Erros de execução são mais difíceis de serem detectados!
- Erros de compilação podem ser consertados antes de executar o programa
- Generics permite transformar erros de execução em erros de compilação, reforçando checagem de tipos
- Aplicações evidente no Collections Framework

Generics

- tipo genérico de dados
  - declaração de um tipo sem informar qual
- variáveis de tipo T
  - variável que identifica o tipo genérico
  - parâmetro formal de tipo

```
public class Box<T> {
        private T myatt;

        public void add(T value) {
            this.myatt = value;
        }

        public T get() {
            return myatt;
        }
}
```

Generics

- Para usar uma classe parametrizada por um tipo genérico é preciso especificar o argumento do tipo a ser usado

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

        Box<Integer> integerBox = new Box<Integer>();

        integerBox.add(new Integer(10));

        Integer someInteger = integerBox.get(); // sem cast!

        System.out.println(someInteger);
    }
```

- Se for tentado isso: `integerBox.add("10");`

```
add(java.lang.Integer) in Box<java.lang.Integer>
    cannot be applied to (java.lang.String)
        integerBox.add("10");
                   ^
    1 error //um erro de compilação
```

Generics

- Convenção de nomeclatura de variáveis de tipos genéricos:
  - E - Element (como no Collections Framework)
  - K - Key
  - N - Number
  - T - Type
  - V - Value
  - S,U,V etc. - 2nd, 3rd, 4th types
- Convenção ajuda a não confundir com nome de classe ou interface

Generics

- Podemos ter métodos e construtores parametrizadas por tipos genéricos

```
public class Box<T> {
        private T myatt;

        public void add(T value) {
            this.myatt = value;
        }
        public T get() {
            return myatt;
        }
        public <U> void inspect(U u){
          System.out.println("T: " + myatt.getClass().getName());
          System.out.println("U: " + u.getClass().getName());
        }
        public static void main(String[] args) {
           Box<Integer> integerBox = new Box<Integer>();
           integerBox.add(new Integer(10));
           integerBox.inspect("some text");
        }
}
```

Generics

- Outro método parametrizado por tipo genérico

```
public static <U> void fillBoxes(U u, List<Box<U>> boxes) {
    for (Box<U> box : boxes) {
        box.add(u);
    }
}
```

note o vínculo

- Em outra parte:

```
Crayon red = ...;
List<Box<Crayon>> crayonBoxes = ...;
```

```
Box.<Crayon>fillBoxes(red, crayonBoxes);
```

ou assim

```
Box.fillBoxes(red, crayonBoxes);
// compilador infere que U é  Crayon
```

Generics

- Podemos limitar o tipo genérico

classe ou interface

```
public <U extends Number> void inspect(U u){
    System.out.println("T: " + t.getClass().getName());
    System.out.println("U: " + u.getClass().getName());
}

public static void main(String[] args) {
    Box<Integer> integerBox = new Box<Integer>();
    integerBox.add(new Integer(10));
    integerBox.inspect("some text"); // error!!
}
```

- **`<U extends Number>`**,U pode ser Number
- Para incluir mais uma interface, use &:

```
<U extends Number & MyInterface>
```

Generics

- Subtipos
  - Podemos fazer isso:

```
Box<Number> box = new Box<Number>();
box.add(new Integer(10));
box.add(new Double(10.1));
```

porque Integer e Double são subclasses de Number

  - Mas se temos:

```
public void boxTest(Box<Number> n){
        // method body omitted
    }
```

não podemos passar como argumento **`Box<Integer>`** ou **`Box<Double>`**.

`Box<Integer>` e **`Box<Double>`** são mais restritivos que **`Box<Number>`!**

Generics

- Wildcards (coringas)
  - Para especificar um tipo desconhecido:

```
public void boxTest(Box<? extends Number> n){
        // method body omitted
}
```

  - Então podemos fazer:

```
Box<Integer> intBox = new Box<Integer>;
boxTest(intBox);
```

  - Mas não podemos fazer isso:

```
Box<?> unknownBox = new Box<Integer>;
unknownBox.add(new Integer(10)); //erro de compilação,
                                 //tipo indeterminado
```

  - E podemos ter ainda **`<? super Integer>`** ou **`<?>`**

Generics

```
public abstract class Shape {
   public abstract void draw(Canvas c);
}
public class Circle extends Shape {
   private int x, y, radius;
   public void draw(Canvas c) { ... }
}
public class Rectangle extends Shape {
   private int x, y, width, height;
   public void draw(Canvas c) { ... }
}
//Para desenhar estes objetos em um canvas
public class Canvas {
  public void draw(Shape s) {
    s.draw(this);
  }
  public void drawAll(List<Shape> shapes) {
    for (Shape s: shapes)
       s.draw(this);
  }
}
```

Podemos chamar esse método com parâmetro `List<Circle>?`

```
public void drawAll(List<? extends Shape> shapes)
```


Generics

- Wildcards ou Métodos Genéricos?

```
static void fromArrayToCollection(Object[] a, Collection<?> c) {
    for (Object o : a) {
       c.add(o);  //erro de compilação!
    }
}
static <T> void fromArrayToCollection(T[] a, Collection<T> c) {
   for (T o : a) {
      c.add(o); // correto
   }
}

      Object[] oa = new Object[100];
      Collection<Object> co = new ArrayList<Object>();
      fromArrayToCollection(oa, co);// T inferred to be Object
      String[] sa = new String[100];
      Collection<String> cs = new ArrayList<String>();
      fromArrayToCollection(sa, cs);// T inferred to be String
      fromArrayToCollection(sa, co);// T inferred to be Object
```

Obs:Um array de String é subclasse de array de Object

Generics

- Wildcards ou Métodos Genéricos?

```
interface Collection<E> {
    public boolean containsAll(Collection<?> c);
    public boolean addAll(Collection<? extends E> c);
}
```

Ou assim?

```
interface Collection<E> {
    public <T> boolean containsAll(Collection<T> c);
    public <T extends E> boolean addAll(Collection<T> c);
}
```

•Use Métodos Genéricos com parametrização de tipo quando houver dependência entre tipos dos argumentos e/ou retorno

Generics

- Wildcards ou Métodos Genéricos?

Usando ambos!

```
class Collections {
    public static <T> void copy(List<T> dest, List<? extends T> src) {
    ...
}
```

- Wildcards podem ser usadas em variáveis:

```
static List<List<? extends Shape>> history =
                      new ArrayList<List<? extends Shape>>();
public void drawAll(List<? extends Shape> shapes) {
    history.addLast(shapes);
    for (Shape s: shapes) {
        s.draw(this);
    }
}
```

Generics

- Type Erasure
  - Na compilação, todos os tipos genéricos são apagados e há conversão para o tipo cru (raw type), que não tem parâmetros de tipo genérico
    - Box<String> é traduzido para o tipo Box (raw type)
  - Não podemos referenciar o tipo na execução:

```
public class MyClass<E> {
    public static void myMethod(Object item) {
        if (item instanceof E) {  //Compiler error
            ...
        }
    E item2 = new E();   //Compiler error
    ...
```

  - Permite operação com código legado (antes do generics, Java 1.5)

Aula baseada em http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf