CONCEITOS BÁSICOS DE ORIENTAÇÃO A OBJETOS

ACH 2003 — COMPUTAÇÃO ORIENTADA A OBJETOS

Daniel Cordeiro

4 de março de 2016

Escola de Artes, Ciências e Humanidades | EACH | USP

EP ZERO — NÚMEROS ROMANOS

Usar TDD para implementar uma classe **Conversor** com dois métodos estáticos:

```
public static int deRomanoParaArabico(String numeroRomano)
public static String deArabicoParaRomano(int numeroArabico)
```

Exemplo de uso:

```
int    x = Conversor.deRomanoParaArabico("MMCMLXXXVIII"); // x = 2988
String y = Conversor.deArabicoParaRomano(2016);           // y = "MMXVI"
```

Entrega: 18 de março (daqui 2 semanas)

HERANÇA

- Classes definem a estrutura e comportamento de objetos
- Herança é a característica de linguagens OO que permite que novas classes sejam criadas usando classes pré-existentes como base
- A nova classe *herda* os campos e métodos da classe original
- Dizemos que a nova classe é uma *subclasse* da original; e a classe utilizada como base é chamada de *superclasse*.

Vantagens:

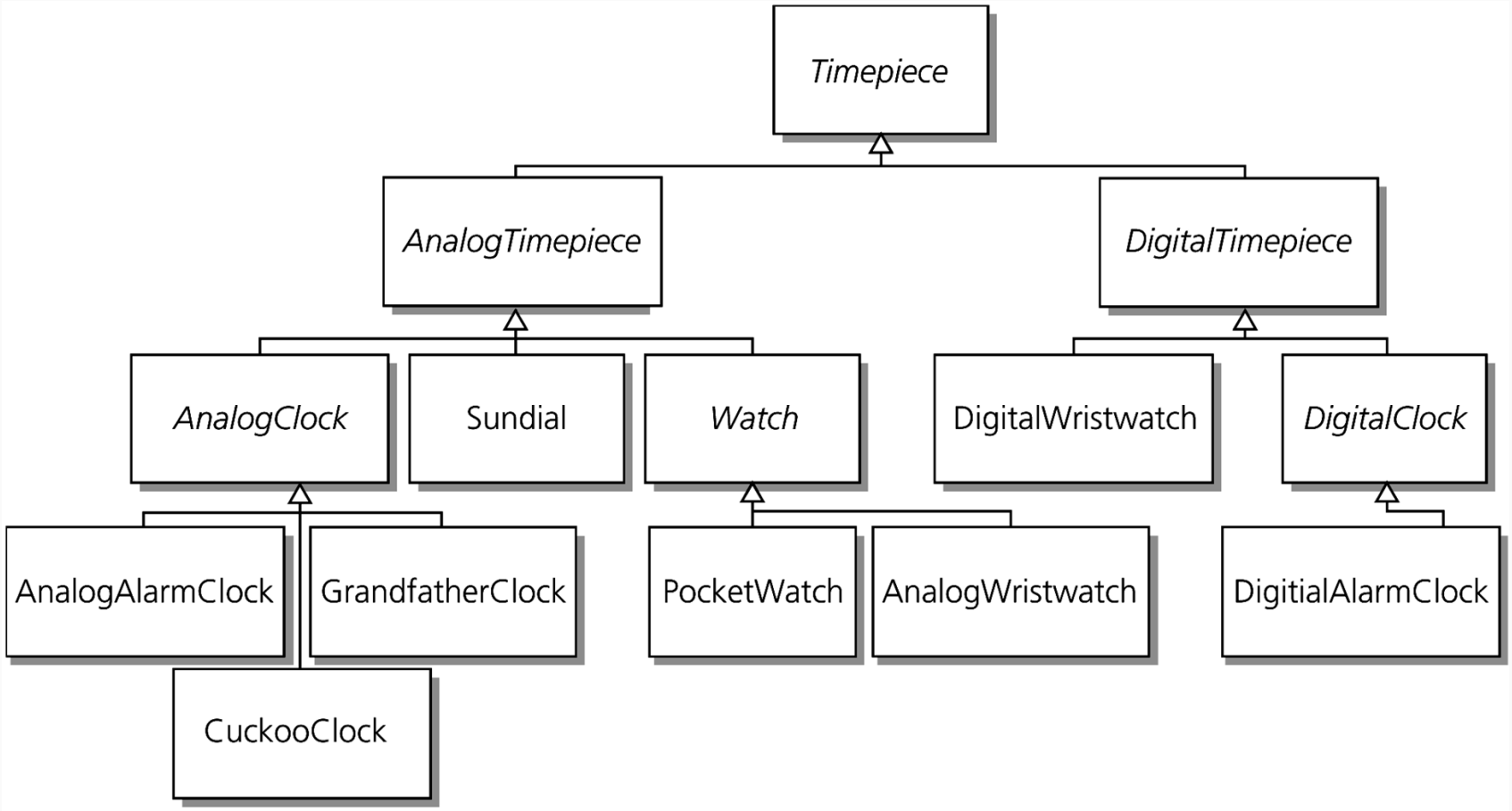
- Melhor modelagem conceitual — hierarquias de especialização são comuns na vida real
- Fatorização — herança permite que propriedades comuns sejam fatorizadas, i.e., definidas apenas uma vez
- Refinamento do projeto e validação — construção de classes com base em outras bem testadas produzirá menos defeitos
- Polimorfismo (mais sobre isso daqui a pouco)

EXEMPLO DE HERANÇA

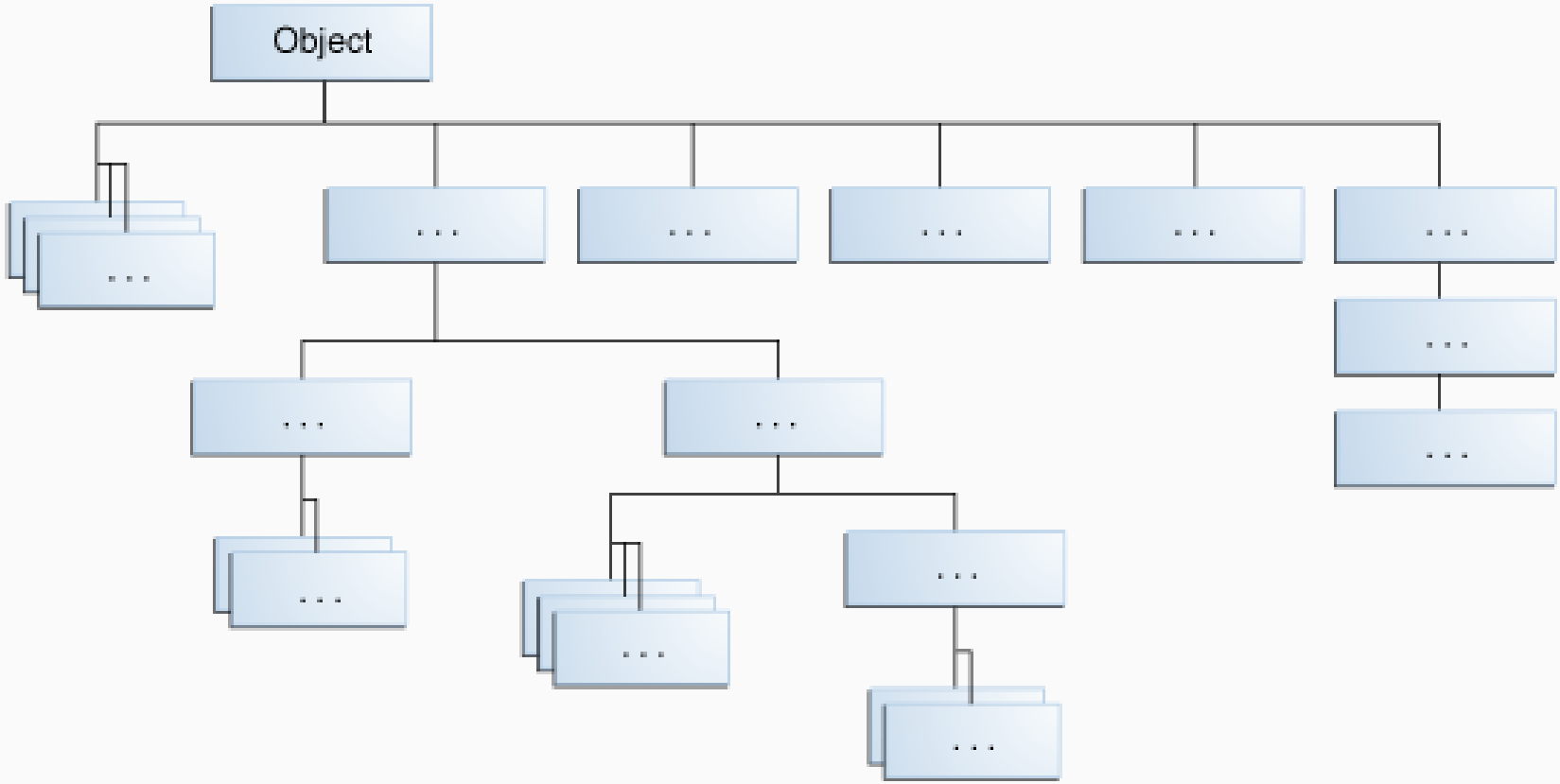
```java
public class Bicycle {

  // a classe Bicycle tem três campos
  public int cadence, gear, speed;

  // the Bicycle class has one constructor
  public Bicycle(int startCadence,
                 int startSpeed,
                 int startGear) {
      gear = startGear;
      cadence = startCadence;
      speed = startSpeed;
  }

  // a classe Bicycle tem quatro métodos
  public void setCadence(int newValue) {
      cadence = newValue;
  }
  public void setGear(int newValue) {
      gear = newValue;
  }
  public void applyBrake(int decrement) {
      speed -= decrement;
  }
  public void speedUp(int increment) {
      speed += increment;
  }

}
```

```java
public class MountainBike extends Bicycle {

  // a subclasse MountainBike adiciona um campo
  public int seatHeight;

  // a subclasse MountainBike tem um construtor
  public MountainBike(int startHeight,
                      int startCadence,
                      int startSpeed,
                      int startGear) {
      super(startCadence, startSpeed, startGear);
      seatHeight = startHeight;
  }

  // a subclasse MountainBike adiciona um método
  public void setHeight(int newValue) {
      seatHeight = newValue;
  }
}
```

EXEMPLO #2


Timepiece
AnalogTimepiece
DigitalTimepiece
AnalogClock
Sundial
Watch
DigitalWristwatch
DigitalClock
AnalogAlarmClock
GrandfatherClock
PocketWatch
AnalogWristwatch
DigitialAlarmClock
CuckooClock

HIERARQUIA DE CLASSES



Polimorfismo pode ser resumido da seguinte forma:

1. Em tempo de compilação, uma variável de um tipo mais geral pode ser usado para referenciar, em tempo de execução, um objeto de qualquer tipo mais específico
2. Em tempo de execução, o objeto referenciado por essa variável se comporta como realmente é, não como o tipo mais geral define

EXEMPLO DE POLIMORFISMO

```java
public class Animal {
  public String nome;

  public Animal(String meuNome) { this.nome = meuNome; }
  public abstract void falar();
  public void responderChamado(String chamado) { falar(); }
}

public class Cachorro extends Animal {
  public void falar() { System.out.println("Au! Au!"); }
}

public class Gato extends Animal {
  public void falar() { System.out.println("Miau!"); }
  public void responderChamado(String chamado) {
    if(chamado.contains("comer")
      falar();
    else
       // quem pensa que é pra ficar me chamando sem motivo?
  }}

Animal toto = new Cachorro("totó");   // Cachorro É-UM Animal
Animal bichano = new Gato("bichano"); // Gato É-UM Animal
toto.responderChamado("Oi, totó!");
bichano.responderChamado("Gatinho, vem comer!");
```

HERANÇA MÚLTIPLA

- Em Java, uma classe não pode estender duas classes diferentes.
- Java implementa múltipla herança:
  - de tipos (uma classe pode implementar múltiplas interfaces)
  - com métodos `default` de interfaces

GENÉRICOS

Defeitos de software

São um fato da vida. Acostume-se. :)

Defeitos de software

São um fato da vida. Acostume-se. :)

- Planejamento, programação cuidadosa e testes podem reduzir seu número
- Alguns são difíceis de encontrar, só ocorrem durante a execução
- Mas outros podem ser encontrados ainda em tempo de compilação.
- **Genéricos** (*Generics*) aumentam o número de defeitos que conseguimos encontrar já durante a compilação

POR QUE GENÉRICOS?

O uso de genéricos permite que o tipo (classes ou interface) seja usado como um parâmetro da definição de novas classes, interfaces e métodos. O uso de genéricos garante que o código seja:

- mais robusto; os tipos podem ser verificados em tempo de compilação
- mais genérico; permitem escrever algoritmos que funcionam em coleções com elementos de tipos diferentes, mas com checagem de tipos
- mais simples; eliminam conversão de tipos (casts)

```java
List list = new ArrayList();
list.add("olá!");
String s = (String) list.get(0);
```

```java
List<String> list = new ArrayList<String>();
list.add("olá!");
String s = list.get(0); // sem cast
```

TIPOS GENÉRICOS

```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}

Box integerBox = new Box();
integerBox.set(8);      // OK
integerBox.set(3.1415); // funciona, tudo é Object
                        // mas será que era o esperado?
```

- o código acima permite o uso de qualquer tipo de objeto
- não há como verificar em tempo de compilação como a classe é utilizada

```java
/**
 * Versão genérica da classe Box
 * @param <T> o tipo do valor que será guardado
 */
public class Box<T> {
    // T significa "Tipo"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}

Box<Integer> integerBox = new Box<Integer>();
integerBox.set(8);      // OK
integerBox.set(3.1415); // Erro de compilação
```

Tipo genérico

em inglês, *generic type* é um tipo que possui um ou mais parâmetros de tipo. Box<T> é um tipo genérico e T é seu parâmetro de tipo.

DEFINIÇÃO

- Uma classe genérica é definida da seguinte forma: `class nome<T1, T2, ..., Tn> { /* ... */ }`
- A seção entre parênteses angulares (`< >`) especifica os parâmetros de tipo (ou variáveis de tipo) `T1, T2, ..., Tn`
- No exemplo, trocamos "`public class Box`" por "`public class Box<T>`"; todas as ocorrências de `Object` foram substituídas por `T`.
- `T` pode ser qualquer tipo não primitivo

```java
/**
 * Versão genérica da classe Box
 * @param <T> o tipo do valor que será guardado
 */
public class Box<T> {
    // T significa "Tipo"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}

Box<Integer> integerBox = new Box<Integer>();
integerBox.set(8);      // OK
integerBox.set(3.1415); // Erro de compilação
```

CONVENÇÃO PARA O NOME DOS PARÂMETROS

Por convenção, os parâmetros são nomeados com uma única letra maiúscula (para não confundir com outras classes ou interfaces. Os tipos parametrizados da linguagem Java são indicados por:

E — elemento (muito usado pelas classes de coleção)

K — chave (*key*)

N — número

T — tipo

V — valor

S,U,V,etc. — 2º, 3º, 4º, ... tipos

EVOCAÇÃO E INSTANCIAÇÃO DE UM TIPO GENÉRICO

Para referenciar uma classe genérica você deve realizar a evocação de um tipo genérico, ou seja, substituir o parâmetro de tipo `T` por um argumento de tipo como, por exemplo, `Integer`:

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

Em Java ≥ 7, se o tipo puder ser inferido pelo compilador, você pode escrever simplesmente:

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

MÚLTIPLOS PARÂMETROS DE TIPO

```java
public interface Pair<K, V> {
  public K getKey();
  public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {
  private K key;
  private V value;

  public OrderedPair(K key, V value) {
    this.key = key;
    this.value = value;
  }

  public K getKey()   { return key; }
  public V getValue() { return value; }
}

Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
Pair<String, String>  p2 = new OrderedPair<String, String>("hello", "world");
```

TIPOS PARAMETRIZADOS

Um tipo parametrizado é um tipo como outro qualquer. Em particular, você pode substituir um parâmetro de tipo (*i.e.*, K ou V) com um tipo parametrizado (*i.e.*, `List<String>`)

```
OrderedPair<String, Box<Integer>> p =
                new OrderedPair<>("primos", new Box<Integer>(...));
```

TIPOS RAW

- Um tipo genérico instanciado sem parâmetros de tipos
  `Box rawBox = new Box();`
- Funciona como se `T` fosse substituído por `Object`
- Feito para permitir que código legado continue funcionando

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;                 // OK
```

Mas você pode receber um *warning* se:

```
Box rawBox = new Box();          // rawBox tem tipo raw Box<T>
Box<Integer> intBox = rawBox;  // warning: unchecked conversion
```

MÉTODOS GENÉRICOS

- Métodos genéricos introduzem seus próprios parâmetros de tipos
- Parecido com tipo genérico, mas o escopo dos parâmetros de tipos é limitado ao método

```
public class Util {
  public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
    return p1.getKey().equals(p2.getKey()) &&
           p1.getValue().equals(p2.getValue());
  }
}

Pair<Integer, String> p1 = new OrderedPair<>(1, "apple");
Pair<Integer, String> p2 = new OrderedPair<>(2, "pear");

// as duas chamadas seguintes são equivalentes
boolean same = Util.<Integer, String>compare(p1, p2);
boolean same = Util.compare(p1, p2); // tipos inferidos pelo compilador
```

PARÂMETROS DE TIPO LIMITADOS

- Os parâmetros de tipo podem ou não ser limitados
- O limite de um parâmetro de tipo restringe os tipos que podem ser usados como argumento. O limite pode ser uma classe e/ou várias interfaces
- O parâmetro de tipo limitado dá acesso aos métodos do "tipo limitante"

```
public class NaturalNumber<T extends Integer> {
  private T n;

  public NaturalNumber(T n)  { this.n = n; }

  public boolean isEven() {
    return n.intValue() % 2 == 0;
  }
}
```

Obs: o método `isEven()` pode chamar o método `intValue()` da classe *Integer*.

MÚLTIPLOS LIMITANTES DE PARÂMETROS DE TIPO

- Um parâmetro de tipo pode ser limitado por mais de um limitante: `<T extends L1 & L2 & L3>`
- Se um dos limitantes for uma classe, ele deve ser especificado primeiro

```
Class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }

class D <T extends A & B & C> { /* ... */ }

// o seguinte comando gera um erro de compilação:
class D <T extends B & A & C> { /* ... */ }
```

LIMITANTES: EXEMPLO

Considere o código seguinte. Há algo de errado nele?

```
public static <T> int countGreaterThan(T[] anArray, T elem) {
  int count = 0;
  for (T e : anArray)
    if (e > elem)
      ++count;
  return count;
}
```

LIMITANTES: EXEMPLO

Considere o código seguinte. Há algo de errado nele?

```
public static <T> int countGreaterThan(T[] anArray, T elem) {
  int count = 0;
  for (T e : anArray)
    if (e > elem)  // erro de compilação
      ++count;
  return count;
}
```

LIMITANTES: EXEMPLO

```
public interface Comparable<T> {
  public int compareTo(T o);
}

public static <T extends Comparable<T>> int countGreaterThan(T[] anArray,
                                                             T elem) {
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;
    return count;
}
```

CURINGAS

- ? (interrogação) representa um tipo desconhecido
- use quando:
  - você escrever um método que pode ser implementado usando as funcionalidades providas pro `Object`
  - quando o código utilizar métodos da classe genérica que não dependem do tipo de parâmetro

```
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
    System.out.println();
}

List<Integer> li = Arrays.asList(1, 2, 3);
List<String>  ls = Arrays.asList("one", "two", "three");
printList(li);
printList(ls);
```

IMPONDO LIMITES AOS CURINGAS

Pode-se impor limites aos curingas:

Limites superiores: `<? extends T>`

- Aceita `T` e todos os seus descendentes
- Ex: se `T` for `Collection`, então aceita `List`, `ArrayList`, etc.

```
public static void process(List<? extends Foo> list) {
    for (Foo elem : list) {
        // ...
    }
}

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

IMPONDO LIMITES AOS CURINGAS

Pode-se impor limites aos curingas:

Limites inferiores: <? super T>

- Aceita T e seus acendentes
- Ex: se T for `ArrayList`, então aceita `List`, `Collection`, `Object`

```
public static void addNumbers(List<? super Integer> list
    for (int i = 1; i <= 10; i++) {
        list.add(i);
    }
}
```

APAGAMENTO

- O compilador Java cria uma única representação de bytecode para cada tipo genérico ou método genérico
- Todas as evocações do tipo genérico e do método genérico são mapeadas para essa representação usando uma técnica chamada de Apagamento (*Type Erasure*)

Antes do apagamento

```
public class Node<T> {

    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) }
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
    // ...
}
```

APAGAMENTO

- O compilador Java cria uma única representação de bytecode para cada tipo genérico ou método genérico
- Todas as evocações do tipo genérico e do método genérico são mapeadas para essa representação usando uma técnica chamada de Apagamento (*Type Erasure*)

Como `T` não é limitado, o compilador o substitui por `Object`:

```
public class Node {

    private Object data;
    private Node next;

    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }

    public Object getData() { return data; }
    // ...
}
```

APAGAMENTO

- O compilador Java cria uma única representação de bytecode para cada tipo genérico ou método genérico
- Todas as evocações do tipo genérico e do método genérico são mapeadas para essa representação usando uma técnica chamada de Apagamento (*Type Erasure*)

Antes do apagamento, com um parâmetro de tipo limitado:

```
public class Node<T extends Comparable<T>> {

    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
    // ...
}
```

APAGAMENTO

- O compilador Java cria uma única representação de bytecode para cada tipo genérico ou método genérico
- Todas as evocações do tipo genérico e do método genérico são mapeadas para essa representação usando uma técnica chamada de Apagamento (*Type Erasure*)

O compilador substitui `T` por `Comparable`:

```
public class Node {

    private Comparable data;
    private Node next;

    public Node(Comparable data, Node next) {
        this.data = data;
        this.next = next;
    }

    public Comparable getData() { return data; }
    // ...
}
```

BIBLIOGRAFIA

- The Java™ Tutorials — Generics: `http://docs.oracle.com/javase/tutorial/java/generics/`