Lenguajes de Programación
Tema 4. Paradigma Orientado a Objetos

Java 1.5

Pedro García López

pgarcia@etse.urv.es/

Versiones de Java

Oak: Designed for embedded devices

Java: Original, not very good version (but it had applets)

Java 1.1: Adds inner classes and a completely new event-handling model

} Java 1

Java 1.2: Includes “Swing” but no new syntax

Java 1.3: Additional methods and packages, but no new syntax

Java 1.4: More additions and the assert statement

} Java 2

Java 1.5: Generics, enums, new for loop, and other new syntax

} Java 5.0

Nuevas caraterísticas de Java 1.5

- Generics (Polimorfismo paramétrico)
  - Comprobación de tipos en tiempo de compilación, elimina el uso intensivo de Object y las conversiones de tipos
- Iterador **for** mejorado
  - Mejora sintáctica que mejora el uso de Iteradores
- Autoboxing/unboxing
  - Wrapping autómático de tipos primitivos
- Tipos enumerados
  - Mejoran la legibilidad del código y evitan el uso de interfaces
- Static import
  - Permite utilizar métodos estáticos sin precederlos de su clase

Iterador for mejorado

- Sintaxis:

  for(*type var* : *array*) {…}

  o for(*type var* : *collection*) {…}

- Ejemplo (Array):

```
for(float x : miArray)
    suma += x;
```

- Ejemplo (colecciones)

  Antes:

```
for (Iterator iter = granja.iterator(); iter.hasNext(); )
    ((Animal) iter.next()).habla();
```

  Ahora:

```
for (Animal a : granja)
    a.habla();
```

Iterar sobre nuestras clases

java.lang interface Iterable<T>

Iterator<T> **iterator**()

Returns an iterator over a set of elements of type T.

java.util interface Iterator<E>

boolean **hasNext**()

Returns true if the iteration has more elements.

E **next**()

Returns the next element in the iteration.

void **remove**()

Removes from the underlying collection the last element returned by the iterator (optional operation).

Auto boxing/unboxing

- Java no permitirá usar tipos primitivos donde se esperan objetos (necesitas un Wrapper)
  - `myVector.add(new Integer(5));`
- Por otro lado, no podemos usar un object donde se espera un tipo primitivo(necesitas convertirlo-unwrap)
  - `int n = ((Integer)myVector.lastElement()).intValue();`
- Java 1.5 lo hace automático:
  - `Vector<Integer> myVector = new Vector<Integer>();`
    `myVector.add(5);`
    `int n = myVector.lastElement();`

Tipos enumerados

- Una enumeración es simplemente un conjunto de constantes que representan diferentes valores
- Como se resolvia antes (java 1.4)
  - public final int SPRING = 0;
    public final int SUMMER = 1;
    public final int FALL = 2;
    public final int WINTER = 3;
- Ahora sería así:
  - enum Season { WINTER, SPRING, SUMMER, FALL }

enums son clases

- Un enum es un nuevo tipo de clase
  - Puedes declarar variables de tipo enumerado y tener comprobación de tipos en tiempo de compilación
    - Cada valor declarado es una instancia de la clase enum
    - Los Enums son implícitamente public, static y final
    - Puedes comparar enums con equals o ==
  - enums hereda de java.lang.Enum e implementa java.lang.Comparable
    - Así que los enums pueden ser ordenados
  - Enums redefinen toString() y proporcionan valueOf()
  - Ejemplo:
    - Season season = Season.WINTER;
    - System.out.println(season); // prints WINTER
    - season = Season.valueOf("SPRING"); // sets season to Season.SPRING

Ventajas de enum

- Enums ofrecen chequeos en tiempo de compilación
  - Implementar enum con interfaces e ints no proporciona chequeo de tipos: season = 43;
- Enums ofrecen un espacio de nombres coherente para cada tipo enumerado (Season.WINTER)
- Enums son robustos
  - If añades, eliminas, o reordenas constantes, debes recompilar
- Como los enum son objetos, los puedes guardar en colecciones. Y puedes añadirles campos y métodos

Enums *son* clases

```
public enum Coin {
    private final int value;
    Coin(int value) { this.value = value; }
    PENNY(1), NICKEL(5), DIME(10),
QUARTER(25);
    public int value() { return value; }
}
```

Otras características de los enums

- values() devuelve un array de tipos enumerados
  - Season[] seasonValues = Season.values();
- switch funciona sobre enumerados
  - switch (thisSeason) { case SUMMER: …; default: …}
  - Debes poner case SUMMER:, y no case Season.SUMMER:

varargs

- Puedes crear métodos y constructores con un número indeterminado de parámetros
  - `public void foo(int count, String... cards) { body }`
  - El "`...`" quiere decir *cero o más* argumentos
  - La llamada podría ser

    ```
    foo(13, "ace", "deuce", "trey");
    ```
  - Solo el último parámetro puede ser vararg
  - Para iterar sobre los argumentos se utiliza el nuevo iterador for:

    ```
    for (String card : cards) { loop body }
    ```

Static import

- import static org.iso.Physics.*;

```
class Guacamole {
    public static void main(String[] args) {
        double molecules = AVOGADROS_NUMBER * moles;
        …
    }
}
```

- Ya no es necesario poner:
  Physics.AVOGADROS_NUMBER
- Otro ejemplo:
  - import static java.lang.System.out;
  - out.println(*something*);

Polimorfismo paramétrico: Generics

- Permite parametrizar tipos
  - Ejemplo de Colecciones: List<Animal>
- Permite al compilador asegurar la validez de los tipos
  - Mueve el control de errores del tiempo de ejecución (ClassCastException) al tiempo de compilación
- Mejora la robustez y legibilidad del código

Ejemplo

- Antes:

```
List granja = new ArrayList();
granja.add(perro);
Animal a = (Animal)granja.get(0);
granja.add("hola");
Animal a = (Animal)granja.get(1);   // ERROR !!!
```

- Despues:

```
List<Animal> granja = new ArrayList<Animal>();
granja.add(perro);
Animal a = granja.get(0);    // SIN CASTING
granja.add("hola");              // ERROR !!!
```

Simplemente te ahorra castings ?

- Chequeo en tiempo de compilación
  - Si introducimos otro tipo nos avisa, no falla con ClassCastException en tiempo de ejecución
- Refuerza las relaciones entre clases
  - Ahora sabemos que un banco tiene una lista de Cuentas. Es más legible y obliga al programador.

Clases genéricas

```
class Cosa<T> {
 T nombre;
 public Cosa(T nombre)  {
        this.nombre = nombre;
 }
 public void setNombre(T nombre) {
        this.nombre = nombre;
 }
 public T getNombre() {
        return nombre;
 }
}
```

Usando clases genéricas

```
Cosa<String> cosa = new Cosa<String>("pedro");
String nombre = cosa.getNombre();
cosa.setNombre(new Integer(1)); // FALLO !!!
```

Métodos genéricos

```
class Bicho {
public <T> T getBicho(T aBicho) {
        return aBicho;
 }
 public <T> void doBicho(T aBicho) {
       T bicho2 = aBicho;
   (...)
 }
public static <T> T getBicho(T aBicho){
        return aBicho;
}

}
```

Polimorfismo paramétrico restringido

- También llamado bounded polimorfism
- Permite restringir el tipo T mediante la herencia <T extends Animal>
- Combinamos el polimorfismo de herencia y el paramétrico
- Permite que usemos las operaciones del tipo restringido (Animal) dentro de nuestro código. Sabemos algo sobre T !!!

Ejemplo:

```
class Cosa2<T extends Animal> {
 T nombre;
 public Cosa2(T nombre)  {
        this.nombre = nombre;
 }
 public void setNombre(T nombre) {
        this.nombre = nombre;
 }
 public T getNombre() {
        return nombre;
 }
public void saluda(){
          nombre.habla();
}
}
```

Ejemplo

```
Cosa2<String> cosa = new Cosa2<String>("pedro");
// FALLO !!!
Cosa2<Perro> cosa = new Cosa2<Perro>(new Perro());
Perro p = cosa.getNombre();
Animal a = cosa.getNombre();
System.out.println(p.habla());
System.out.println(a.habla());
```

Restringiendo métodos

```
class Bicho {
public <T> T getBicho(T aBicho) {
        return aBicho;
 }
public static <T extends Animal> T getBicho(T aBicho){
        System.out.println(aBicho.habla());
        return aBicho;
}

}

String x = Bicho.getBicho("hola");   //FALLO !!!
Animal a = Bicho.getBicho(new Perro());
Animal a2 = Bicho.getBicho(new Loro());
```

Tipos encadenados (Nested types)

- Podríamos decir:
  - class Cosa <T extends Animal<E>>
- O bien:

```
LinkedList<LinkedList<String>> zss = new
    LinkedList<LinkedList<String>>();
```

```
class Cosa <T extends Map<Foo,Bar>>
```

Cuidado con T

- No podemos instanciar tipos genéricos:
  - `T elem = new T();`
- No podemos instanciar un array de T
  - `T lista[] = new T[5];`
- Si podemos instanciar una colección de T **dentro** de una clase o método genérico
  - `List<T> lista = new LinkedList<T>();`

Wildcards (?)

```
Cosa<String> cosa = new Cosa<String>("pedro");
System.out.println(cosa.getNombre());

Cosa<Object> aCosa = new Cosa<Object>("pedro");
String x = aCosa.getNombre();   // Qué pasa ?
String x2 = (String)aCosa.getNombre(); // Qué
pasa ?

cosa = aCosa; // Qué pasa ? ES ILEGAL
```

Ejemplos

```
public static <T> void test(Collection<T> c) {

    //String lista[] = {"uno", "dos" , "tres", "cuatro", "tres"};
    //   for( int i=0; i < lista.length; i++ )
   //       c.add( lista[i] );       //ILEGAL !!

     Iterator<T> it = c.iterator();
     while( it.hasNext() )
           System.out.println( it.next() );
     System.out.println( "----------------------");

   }
```

Ejemplos

```
public static void main( String args[] )
  {
        Collection<String> c;
        String lista[] = {"uno", "dos" , "tres", "cuatro", "tres"};

        c = new ArrayList<String>();
        for( int i=0; i < lista.length; i++ )
                c.add( lista[i] );
        test(c);
}
```

Ejemplos

```
public static <T> void test2(Collection<Object> c) {
    Iterator<Object> it = c.iterator();
    while( it.hasNext() )
        System.out.println( it.next() );
    System.out.println( "----------------------");
 }
 public static void main( String args[] )
   {
        Collection<String> c;
        String lista[] = {"uno", "dos" , "tres", "cuatro", "tres"};
        c = new ArrayList<String>();
        for( int i=0; i < lista.length; i++ )
                c.add( lista[i] );
        test2(c);    //ILEGAL !!!!!!
}
```

Ejemplos

```
public static <T> void test3(Collection<?> c) {
    Iterator<?> it = c.iterator();
    while( it.hasNext() )
        System.out.println( it.next() );
    System.out.println( "----------------------");
 }
 public static void main( String args[] )
   {
        Collection<String> c;
        String lista[] = {"uno", "dos" , "tres", "cuatro", "tres"};
        c = new ArrayList<String>();
        for( int i=0; i < lista.length; i++ )
                c.add( lista[i] );
        test3(c);     //OK
}
```

+ Ejemplos (Bounded Wildcards)

```
public static <T> void test(Collection<? extends Animal> c) {
        Iterator<? extends Animal> it = c.iterator();
    while( it.hasNext() )
        System.out.println( it.next().habla() );
    System.out.println( "----------------------");
}

public static <T extends Animal> void test2(Collection<T> c){
    Iterator<T> it = c.iterator();
    while( it.hasNext() )
        System.out.println( it.next().habla() );
    System.out.println( "----------------------");
}
```

Y + Ejemplos

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

        Animal a = new Animal(3,3);
        Gato g = new Gato(1,2);
        LoroMutante lM = new LoroMutante(3,3);
        LinkedList<Animal> granja =new LinkedList<Animal>();
        granja.add(a);
        granja.add(g);
        granja.add(lM);
        test(granja);
        test2(granja);

}
```

Lower Bounded Wildcards

- <? super T>

Representa un tipo que es una superclase de T o es T

Lo contrario de extends

Sirve para organizar pasos de parámetros

Multiples límites:

- <T extends Animal & Comparable<T>>
- < ? extends Animal & Comparable<T>>
- <T extends Animal & Comparable<T> & Serializable>

java.util
Interface Collection<E>

boolean **add**(E o)
boolean **addAll**(Collection<? extends E> c)
void **clear**()
boolean **contains**(Object o)
boolean **containsAll**(Collection<?> c)
boolean **equals**(Object o)
int **hashCode**()
boolean **isEmpty**()
Iterator<E> **iterator**()
boolean **remove**(Object o)
boolean **removeAll**(Collection<?> c)
boolean**retainAll**(Collection<?> c)

Interface Iterator<E>

boolean hasNext()
E next()
void remove()

java.util
Interface Map<K,V>

void **clear**()
boolean **containsKey**(Object key)
boolean **containsValue**(Object value)
Set<Map.Entry<K,V>> **entrySet**()
boolean **equals**(Object o)
V **get**(Object key)
int **hashCode**()
boolean **isEmpty**()
Set<K> **keySet**()
V **put**(K key, V value)
void **putAll**(Map<? extends K,? extends V> t)
V **remove**(Object key)
int **size**()
Collection<V> **values**()

java.util Interface Comparator<T>

int **compare**(T o1, T o2)

java.util Class Collections

static <T extends Comparable<? super T>> void **sort**(List<T> list)
<T> void **sort**(List<T> list, Comparator<? super T> c)

static <T> int **binarySearch**(List<? extends Comparable<? super T>> list, T key)
static <T> int **binarySearch**(List<? extends T> list, T key, Comparator<? super T> c)

Erasure- Borrado de tipos

- A diferencia de C++, el compilador Java elimina la información de tipos parametrizados por compatibilidad con versiones anteriores.
- El polimorfismo paramétrico es una característica que solo existe en tiempo de compilación, Java hace el trabajo sucio entre bambalinas (castings, T -> Object, …).
- Las instancias de objetos no tienen información de tipos parametrizados en tiempo de ejecución
- List<Integer> y List<Thing> son la misma clase de tipo List.

Ejemplo:

Tiempo de compilación:

```
public String loophole(Integer x) {
        List<String> ys = new LinkedList<String>();
        List xs = ys;
        xs.add(x); // compile-time unchecked warning
        return ys.iterator().next();
}
```

Tiempo de ejecución:

```
public String loophole(Integer x) {
        List ys = new LinkedList;
        List xs = ys;
        xs.add(x);
        return (String) ys.iterator().next(); // run time
error
}
```

Ejemplo del Map

```
public interface Imap<T> {
        public T aplica(T elem);
}
public class Increment implements Imap<Integer> {
        public Integer aplica(Integer elem) {
                return elem+1;
        }
}
public class Cuadrado implements Imap<Integer>{
        public Integer aplica(Integer elem) {
                return elem*elem;
        }
}
```

```
public class Mayusculas implements Imap<String> {
        public String aplica(String elem) {
                return elem.toUpperCase();
        }
}

public class Map {
public static <T> List<T> map(List<T> lista,
                                        Imap<T> funcion){
                List<T> result = new LinkedList<T>();
                for (T elem: lista)
                        result.add(funcion.aplica(elem));
                return result;
        }
}
```

```
int lista[] = {1,2,3,4,5};
String lista2[] =
{"pedro","lenguajes","programacion","Java","haskell"};

LinkedList<Integer> listaint = new LinkedList<Integer>();
for (int i:lista)
        listaint.add(i);
LinkedList<String> listastring = new LinkedList<String>();
for (String i:lista2)
        listastring.add(i);

List<Integer> result = Map.map(listaint,new Increment());
System.out.println("lista de incrementos:");
for (Integer elem:result)
        System.out.println(elem);
```

Diferencias con el map de 1.4 ?

```
System.out.println("lista de cuadrados:");
List<Integer> result2 = Map.map(listaint,new Cuadrado());
for (Integer elem:result2)
        System.out.println(elem);

System.out.println("lista de cadenas en mayuscula:");
List<String> result3 = Map.map(listastring,new Mayusculas());
for (String elem:result3)
        System.out.println(elem);
```