

Genéricos y colecciones

Tecnología de la Programación

Curso 2019-2020

Jesús Correas – jcorreas@ucm.es

**Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid**

(Basado en material creado por Puri Arenas, utilizando los apuntes de
Marco Antonio Gómez y Jorge Gómez)

Introducción

- El objetivo de los genéricos es facilitar el desarrollo de código genérico y reutilizable.
- Vamos a comenzar viendo un ejemplo con una **Pila redimensionable**.
- Una pila tiene una serie de operaciones básicas
 - ▶ **push(elem)** para añadir un elemento a la pila,
 - ▶ **pop()** para extraer el último elemento añadido.
 - ▶ **estaVacía()** para saber si la pila está vacía.
- Estas operaciones son independientes de los datos que se introduzcan en la pila.
- Sin embargo, si se define una pila para almacenar enteros, **sólo sirve para este tipo de datos**.
- Si se define para objetos de una clase, solo se pueden utilizar objetos de esa clase (o subclases).

Clase Pila de enteros

```
public class Pila {
    private int[] datos;
    private int numDatos;
    private static final int TAM_INI = 10, TAM_INC = 10;
    public Pila() {
        datos = new int[TAM_INI];
        numDatos = 0;
    }
    public void push(int dato) {
        if (numDatos == datos.length) {
            int[] aux = new int[datos.length+TAM_INC];
            System.arraycopy(datos, 0, aux, 0, datos.length);
            datos = aux;
        }
        datos[numDatos++] = dato;
    }
    public int pop() {
        if (numDatos > 0) {
            numDatos--;
            return datos[numDatos];
        } else { /* lanza excepcion */ }
    }
}
```

Generalización de la clase `Pila`. Primera aproximación

- Podríamos diseñar una **pila genérica**, para datos de cualquier tipo.
- Si en lugar de un tipo específico del programa utilizamos **`Object`**, la pila sirve para cualquier tipo no primitivo.
- Para los tipos primitivos podemos utilizar los tipos no primitivos equivalentes (`Integer`, etc.) y *boxing/unboxing*.

```
public class PilaObject {  
    private Object[] datos;  
    ...  
    public PilaObject() {  
        datos = new Object[TAM_INI];  
        ... }  
    public void push(Object dato) {  
        if (numDatos == datos.length) {  
            Object[] aux = new Object[datos.length+TAM_INC];  
            ... }  
    public Object pop() {  
        ... }  
}
```

Generalización de la clase `Pila`. Primera aproximación

- Cuando se utiliza este enfoque y se almacenan objetos de una clase, es necesario hacer **cast** cuando se extraen elementos de la pila:

```
...
Pila p = new PilaObject();
p.push("primer elemento.");
p.push("segundo elemento.");
String s = p.pop(); // Error de compilacion: Incompatible types.
...
String s = (String)p.pop(); // Esto si compila.
```

- Pero este código es **inseguro**: durante la compilación no se puede garantizar que no se va a producir un error de ejecución:

```
...
Pila p = new PilaObject();
p.push(new Integer(37));
String s = (String)p.pop(); // Compila correctamente.
```

- Durante la ejecución:

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



Genéricos

- Desde Java 5 existe la posibilidad de referirse a un tipo de datos de forma genérica.
- Los genéricos de Java son *azúcar sintáctico* para no tener que hacer *cast* inseguros continuamente.
- La idea de los genéricos consiste en **parametrizar** la definición de una clase respecto a un tipo genérico.
- Para ello, se utilizan *parámetros* para representar un tipo de datos (también llamados *variables de tipo*).
- **Cuando se crea un objeto de esta clase** en otro punto del programa, **se especifica el tipo concreto** que va a tomar el parámetro.
- **Notación:** los parámetros suelen representarse con identificadores formados por una sola letra mayúscula.
- Otros lenguajes disponen de mecanismos más potentes de programación genérica (como las plantillas de C++).

Clase Pila utilizando genéricos

```
public class PilaGenerica<T> {  
    private Object[] datos; // internamente se usa un array de Object  
    private int numDatos;  
    private static final int TAM_INI = 10, TAM_INC = 10;  
    public PilaGenerica() { ... }  
  
    public void push(T dato) {  
        if (numDatos == datos.length) {  
            Object[] aux = new Object[datos.length+TAM_INC];  
            System.arraycopy(datos, 0, aux, 0, datos.length);  
            datos = aux;  
        }  
        datos[numDatos++] = dato;  
    }  
  
    @SuppressWarnings("unchecked")  
    public T pop() throws Exception {  
        if (numDatos == 0)  
            throw new Exception("horror");  
        numDatos--;  
        return (T) datos[numDatos]; //este casting siempre es seguro  
    }  
}
```

Uso de la clase genérica Pila

- Internamente se utilizan arrays de objetos **Object** porque no es posible crear objetos ni arrays de tipos genéricos.
- Sin embargo, cuando se utiliza un objeto de tipo Pila, **no es necesario hacer cast:**

```
...
PilaGenerica<String> p = new PilaGenerica<String>();
p.push("primer elemento.");
p.push("segundo elemento.");
String s = p.pop(); // no es necesario cast.
...
```

- Además es **seguro**: se comprueba el tipo de los datos en tiempo de compilación.

```
...
PilaGenerica<String> p = new PilaGenerica<String>();
p.push(new Integer(37)); // Error de compilacion.
Integer v = p.pop(); // Error de compilacion.
```

- De esta forma se puede **controlar el tipo de objetos** que se pueden

Uso de la clase genérica Pila

- Al crear una Pila genérica de un tipo específico, **todos sus elementos deben ser del tipo especificado o sus subclases:**

```
PilaGenerica<Number> p = new PilaGenerica<Number>(); // (*)  
p.push(new Integer(44));  
p.push(new Double(37.5));
```

- El tipo que se utiliza en (*) puede ser **cualquier tipo no primitivo**.
 - ▶ Utilizando *boxing/unboxing* se pueden utilizar las clases que representan tipos primitivos:

```
PilaGenerica<Integer> p = new PilaGenerica<Integer>();  
p.push(44); p.push(37);  
int x = p.pop();
```



Clases e interfaces Genéricas. Herencia

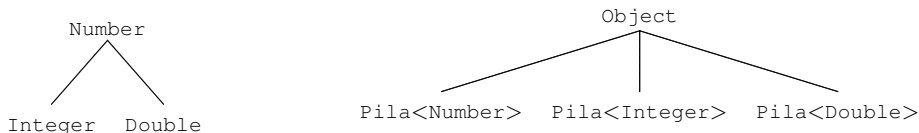
- Se pueden crear tanto clases como interfaces genéricas: la siguiente interfaz permite iterar sobre una colección de objetos de tipo E.

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove(); }
```

- Se puede heredar de clases genéricas o implementar interfaces genéricas. Ejemplos de `java.util`:

```
public class ArrayList<E> extends AbstractList<E>  
    implements List<E>, ... { ... }
```

- La herencia de clases/interfaces genéricos **debe ser explícita**:
Number es superclase de **Integer**, pero
Pila<Number> **no es superclase de Pila<Integer>**.



Limitación de la genericidad

- Hay casos en los que es conveniente limitar las posibles clases que pueden utilizarse como parámetros en una clase genérica.
- Por ejemplo, se pueden limitar los tipos que se pueden utilizar en la declaración de la pila:

```
public class PilaGenerica<T extends Number> {  
    ...  
}
```

- En este caso, las únicas clases que se pueden utilizar para crear una pila son `Number` o sus subclases.
- En este contexto, la palabra **extends** se utiliza tanto para referirse a **herencia** como para **implementación** de interfaces.
- Esto nos permite **mejorar el control sobre los objetos** que se pueden utilizar en determinados casos. Lo vemos con un ejemplo.



Limitación de la genericidad. Ejemplo

- En el tema anterior se ilustró el uso de interfaces con un interfaz `Comparable` y las clases `Rational` y `Fecha`. Después se podía crear un método para **ordenar** un array de objetos que implementaban el interfaz.
- Una dificultad que encontramos es **cómo controlar la posibilidad de que se comparen un `Rational` con una `Fecha`**.
 - ▶ Una solución es **lanzar una excepción**.
- Mediante genéricos podemos limitar **en tiempo de compilación** los tipos de objetos que podemos comparar entre sí.
- Definimos un interfaz `Comparable` de la siguiente forma:

```
public interface Comparable<T> {  
    boolean less(T c);  
    boolean equal(T c);  
}
```

(Nota: El interfaz `java.lang.Comparable` es ligeramente diferente del expuesto en este documento).

Limitación de la genericidad. Ejemplo

- Las clases que implementan Comparable se definen:

```
public class Rational implements Comparable<Rational> {
    private int num, den;
    public Rational(int n, int d){ num = n; den = d; }
    @Override
    public boolean less(Rational r) { // solo comparo con racionales
        return (((double)this.num)/this.den < ((double)r.num) / r.den);
    }
}

public class Fecha implements Comparable<Fecha> {
    private int anno, mes, dia;

    public Fecha(int d, int m, int a) { dia = d; mes = m; anno = a; }
    @Override
    public boolean less(Fecha f) { // ahora comparo con fechas
        return (this.anno<f.anno || (this.anno==f.anno &&
            (this.mes<f.mes || (this.mes==f.mes && this.dia<f.dia))));
    }
}
```

Limitación de la genericidad. Ejemplo

- Ahora podemos ordenar cualquier array de objetos que implementen Comparable **y sean comparables entre sí:**

```
public class SelectionAlgorithm<T extends Comparable<T>> {  
    public void sortArray(T[] v) {  
        T minx;  
        int minj;  
        for (int i = 0; i < v.length-1; i++) {  
            minj = i;  
            minx = v[i];  
            for (int j = i+1; j < v.length; j++) {  
                if (v[j].less(minx)) {  
                    minj = j;  
                    minx = v[j];  
                }  
            }  
            v[minj] = v[i];  
            v[i] = minx;  
        }  
    }  
}
```



Métodos genéricos

- Igual que se pueden definir clases genéricas, se pueden definir **métodos genéricos**.
- En el ejemplo anterior, se puede indicar que el método `sortArray` es genérico:

```
public class SelectionAlgorithm {  
    public static <T extends Comparable<T>> void sortArray(T[] v) {  
        T minx;  
        ...  
    } }  

```

- La clase no es genérica, solo el método.
- El ámbito del parámetro genérico **T** es el método. Fuera del método, **T** no está definido.
- Para llamar a un método (estático) genérico:

```
Fecha[] fec = {new Fecha(...), ...};  
...  
SelectionAlgorithm.<Fecha>sortArray(fec);  

```

- **Esta es la forma de definir métodos estáticos genéricos.**

Uso de comodines al referirse a genéricos

- A veces es necesario poder referirse de forma general a varias instanciaciones posibles de una clase genérica.
- Por ejemplo, si tenemos un método que recibe como argumento una pila de `Number`,

```
public void escribePilaNumeros(Pila<Number> p) {  
    while (!p.vacia()) {  
        Number n = p.pop();  
        System.out.println(n);  
    }  
}
```

- **No es posible llamar a este método con una pila de `Integer`:**

```
Pila<Integer> pi = new Pila<Integer>();  
...  
escribePilaNumeros(pi); // Error de compilacion!
```

- No es correcto porque `Pila<Integer>` **no es subclase de** `Pila<Number>`.

Uso de comodines al referirse a genéricos

- Este problema se puede resolver utilizando **comodines** (*wildcards*):

```
public void escribePilaNumeros(Pila<? extends Number> p) {...}
```

- Esta notación permite referirse a distintas *instanciaciones* de `Pila` de forma general.
- Sin embargo, en este método no es posible realizar algunas operaciones sobre la pila, pues **el tipo de los elementos de la pila es desconocido**.
- Por ejemplo, **no es posible añadir elementos a la pila**:
 - ▶ Se podría intentar añadir un elemento de tipo `Double` a una pila de objetos de tipo `Integer`, pues ambas clases heredan de `Number`.
- **Sí se pueden utilizar todos los métodos de la clase `Number`** sobre los elementos de la pila.
- Si se desea utilizar un comodín para referirse **a cualquier clase posible**, se puede utilizar la siguiente notación:

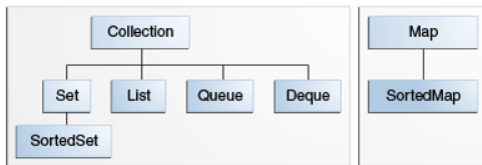
```
public void escribePilaNumeros(Pila<?> p) {...}
```



Colecciones

- Los genéricos se utilizan fundamentalmente para definir estructuras de datos en el API de Java.
- Las estructuras de datos están definidas mediante el “**Collections framework**”.
- Es un conjunto de clases e interfaces definidas en `java.util` que forman **dos jerarquías de clases e interfaces distintas** cuyas raíces son:
 - ▶ La interfaz `Collection<E>`, para representar grupos de objetos en los que se pueden añadir y eliminar elementos.
 - ▶ La interfaz `Map<K, V>`, que generalizan el concepto de array: permiten seleccionar un elemento (V) a partir de su “posición” (K).

Fuente: <http://docs.oracle.com/javase/tutorial/collections/>



Interfaz Collection<E>

- Representa un grupo de objetos en el sentido más general posible. Los subinterfaces más relevantes de este interfaz son:
 - ▶ **Set<E>** representa un grupo de elementos que no contiene duplicados.
 - ★ implementaciones: **TreeSet**.
 - ▶ **List<E>** representa un grupo de elementos ordenados, que puede contener elementos duplicados. Se puede acceder a los elementos por su posición en la lista, buscar un elemento y eliminar un elemento.
 - ★ implementaciones principales: **ArrayList**, **LinkedList**.
- El interfaz `Collection<E>` declara un método para que todas las implementaciones devuelvan un **iterador**.
- Un iterador (interfaz **Iterator<E>**) permite recorrer todos los elementos de una colección de forma sencilla. Por ejemplo:

```
ArrayList<Tipo> col = new ArrayList<Tipo>();  
...  
Iterator<Tipo> it = col.iterator();  
while (it.hasNext()) {  
    System.out.println(it.next());  
}
```



Interfaz Map

- Un objeto **Map<K, V>** almacena un grupo de objetos en memoria y permite acceder a ellos a través de una clave.
- Tiene dos parámetros: **K** es el tipo de datos utilizado para la clave y **V** el tipo de los objetos almacenados.
- Clases que implementan este interfaz: **TreeMap**, **HashMap**.
- Por ejemplo, podemos utilizar un **Map<String, Persona>** para mantener en memoria un grupo de objetos **Persona** indexados por su **dni**, que es un **String**:

```
class Persona {  
    private String dni;  
    private String nombre;  
    private int edad; ...  
}  
  
...  
TreeMap<String,Persona> grupo = new TreeMap<String,Persona>();  
Persona p1 = new Persona("111", "Orson Welles", 37);  
grupo.put(p1.getDni(), p1); //para sacar el dato: grupo.get("111");  
...
```

Interfaz Map

- El interfaz **Map** no contiene métodos específicos para recorrer los elementos que contiene.
- Pero sí dispone de métodos para devolver colecciones que representan su contenido:
 - ▶ **Set<K> keySet()** devuelve el conjunto de claves almacenados en el **Map**.
 - ▶ **Collection<V> values()** devuelve una colección con los valores almacenados (pues pueden estar repetidos).
 - ▶ **entrySet()** devuelve el conjunto de pares (clave,valor).

