Departamento de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

PRACTICA 9: Tipos genéricos.

1. OBJETIVO

El objetivo de esta práctica es entender la utilidad de los tipos genéricos en java y su utilización en la elaboración y reutilización de código, así como las restricciones que presenta su utilización.

2. TIPOS GENÉRICOS

Un tipo genérico es una clase o interfaz genérica que está parametrizada con tipos. Los tipos genéricos permiten que una clase o método pueda operar con objetos de tipos diferentes, proporcionando seguridad en los tipos en tiempo de compilación. Evitan el uso del operador `cast`. Una posible solución para poder operar con objetos de tipos diferentes, sin utilizar tipos genéricos, es utilizar una referencia de la clase `Object`, tal y como se muestra a continuación.

Fichero: Bolsa.java

```
package fp2.poo.practica09;
public class Bolsa{
    private Object objeto;
    public void agrega(Object objeto) {
      this.objeto = objeto;
    }
    public Object obtiene() {
         return objeto;
    }
}
```

Fichero: Ejemplo01.java

```
package fp2.poo.practica09;

public class Ejemplo01 {
    public static void main(String[] args){
        // SOLO objetos Integer en Bolsa!
        Bolsa cajaDeInteger = new Bolsa();
        cajaDeInteger.agrega(new Integer(10)); //Linea a comentar
        //cajaDeInteger.agrega(new String("10"));
        /* Con el cast */
        Integer unInteger = (Integer)cajaDeInteger.obtiene();
        /* Sin el cast */
        //Integer unInteger = cajaDeInteger.obtiene();
        System.out.println(unInteger);
    }
}
```

En la clase `Ejemplo01` se muestra el uso de la clase `Bolsa`, que contiene el atributo `objeto` de tipo `Object`. Usa el método `agrega`, para proporcionar un objeto de

tipo `Integer` que se guarda en el atributo `objeto`. También usa el método `obtiene`, para obtener el objeto.

Descargue el código de la plataforma para esta práctica y descomprímalo. Para ver el funcionamiento del programa ejecute la siguiente orden:

`make prueba01`

Comente la línea que usa el objeto de tipo `Integer`(`cajaDeInteger.agrega(new Integer(10)); //Linea a comentar`) y descomente la línea siguiente (`cajaDeInteger.agrega(new String("10"));`). Vuelva a compilar y ejecutar el código. En este caso, se generará una excepción en tiempo de ejecución. Compruebe que si no utiliza el cast genera un error en compilación.

Un tipo de dato genérico es una clase, o interfaz, que está parametrizada con los tipos de datos que se desea admitir, siempre que estos no sean tipos de datos primitivos. A los tipos de datos genéricos también se les denominan **tipos parametrizados**. Para declarar una clase o interfaz de tipo genérico (también llamadas clase o interfaz genérica) debemos utilizar el operador diamante (**<·>**) en cuyo interior se utiliza al menos una variable genérica (**`T`**, **`U`**, **`V`**, **`K`**, **`S`**, etc…) que será sustituida en tiempo de compilación por el tipo de dato especificado en su declaración. En el siguiente ejemplo se utiliza una clase genérica llamada `Caja`.

Fichero: Caja.java

```
package fp2.poo.practica09;

/**
 * Version con Genericos de la clase Caja.
 */
public class Caja<T> {

    private T objeto;

    /** Constructor sin parámetros.
     *  El constructor está sobrecargado.
     */
    public Caja( ){
        this.objeto = null;
    }

    /** Constructor con un parámetro.
     *  El constructor está sobrecargado.
     */
    public Caja( T objeto ){
        this.objeto = objeto;
    }

    /** Metodo para añadir un objeto. */
    public void agrega(T objeto) {
        this.objeto = objeto;
    }

    /** Metodo para obtener el objeto. */
    public T obtiene() {
        return objeto;
    }
}
```

Fichero: Ejemplo02.java

```
package fp2.poo.practica09;

public class Ejemplo02 {
    public static void main ( String[] args ) {
        /* Se crea un objeto Caja especificando como el tipo "String"   */
        Caja<String> caja01DeString = null; /* Declaracion de la variable*/
        caja01DeString = new Caja<String>(); /* Construccion del obj.    */
        caja01DeString.agrega("Fundamentos de Programación II"); /* Uso.*/
        System.out.println(caja01DeString.obtiene());

        Caja<String> caja02DeString = null;
        caja02DeString = new Caja<String>("Fundamentos de Programación II");
        System.out.println(caja02DeString.obtiene());

        /* Se crea un objeto Caja especificando como el tipo "Integer" */
        Integer unEntero = null;
        unEntero = new Integer(2);
        Caja<Integer> cajaDeInteger = new Caja<Integer>();
        cajaDeInteger.agrega(unEntero);
        System.out.println("Fundamentos de Programación "
                           +cajaDeInteger.obtiene());
    }
}
```

En la clase `Ejemplo02.java` se crean objetos de la clase `Caja` indicando los tipos `String` e `Integer`.

Para ver el funcionamiento del programa ejecute la siguiente orden:

`make prueba02`

Observe que en este caso no es necesario el cast al obtener el objeto de `Caja`. Esto es debido a que los tipos genéricos evitan el uso de los operadores casts, que realizan una conversión explícita al tipo nombrado en el operador cast, ya que el compilador sabe el tipo de dato que se está admitiendo, permitiendo que las clases implementen algoritmos genéricos. Observe que uno de los constructores de la clase `Caja` admite un parámetro de tipo `T`.

Los tipos genéricos añaden estabilidad al código permitiendo la detección de fallos en tiempo de compilación así, por ejemplo, si tenemos en el código una caja genérica, **`Caja<T>`**, podemos conseguir que admita datos de tipo cadena (**`Caja<String>`**), de tipo entero (**`Caja<Integer>`**) o un booleano (**`Caja<Boolean>`**), sin embargo, si intentamos añadir a un objeto de la clase **`Caja<String>`** (que admite un tipo de dato cadena) otro tipo de dato (como puede ser un número entero), en tiempo de compilación dará un error debido a que son tipos de datos incompatibles.

En el siguiente ejemplo se ha declarado un objeto del tipo Caja parametrizado con el tipo Integer. Al compilar **`Ejemplo03.java`**, se produce un error debido a que se intenta invocar el método **`agrega`** de **`Caja`** con un `String`.

Fichero: Ejemplo03.java

```
package fp2.poo.practica09;

public class Ejemplo03 {
    public static void main(String[] args) {

        // SOLO objetos Integer en Caja!
        Caja<Integer> cajaInteger = null;
        cajaInteger = new Caja<Integer>();


       // Se intenta agregar un objeto de tipo String
        cajaInteger.agrega("10"); // es un String

        //cajaInteger.agrega(new Integer(10));
        System.out.println(cajaInteger.obtiene());
    }
}
```

Para ver el funcionamiento del programa ejecute la siguiente orden:

`make prueba03`

Cambie el programa para que se compile y ejecute correctamente.

Los errores que ocurren en tiempo de compilación son más rápidos de detectar y corregir que los que ocurren en tiempo de ejecución, por eso los tipos parametrizados ofrecen más seguridad (que la utilización de la referencia a `Object`).

Dado que estamos trabajando con parámetros, también es posible utilizar más de un tipo para parametrizar una clase genérica.

Fichero: Pareja.java

```
package fp2.poo.practica09;

public class Pareja<X,Y>  {
  private X primero;
  private Y segundo;

  public Pareja(X a1, Y a2) {
    this.primero  = a1;
    this.segundo  = a2;
  }
  public X getPrimero()  {
     return this.primero;
  }
  public Y getSegundo() {
     return this.segundo;
  }
  public void setPrimero(X arg) {
     this.primero = arg;
  }
  public void setSegundo(Y arg) {
     this.segundo = arg;
  }
}
```

Fichero: Ejemplo04.java

```
package fp2.poo.practica09;

public class Ejemplo04{
    public static void main (String args[]){
        Pareja<String,Integer> par
                = new Pareja<String,Integer>("cadena",new Integer(1));
        System.out.println("Parametro primero: " + par.getPrimero());
        System.out.println("Parametro segundo: " + par.getSegundo());
        par.setPrimero("cambio cadena");
        par.setSegundo(new Integer(21));
        System.out.println("Parametro primero: " + par.getPrimero());
        System.out.println("Parametro segundo: " + par.getSegundo());
    }
}
```

En este ejemplo se muestra que a la clase `Pareja` se le proporcionan dos tipos representados mediante `X` e `Y`. La clase `Pareja` se utiliza en el método `main` de la clase `Ejemplo04`.

Para ver el funcionamiento del programa ejecute la siguiente orden:

`make prueba04`

Los tipos genéricos se pueden aplicar tanto a métodos como a constructores. Un ejemplo se puede ver en `Caja05.java`.

Fichero: Caja05.java

```
package fp2.poo.practica09;

public class Caja05 {

    public  Caja05 () {
      System.out.println("Constructor Caja05 sin parametros");
    }

    public <T> Caja05 ( T tipo ) {
      System.out.println("\nConstructor Caja05 con parametros");
      System.out.println("Tipo y su valor: " + tipo.getClass().getName()
                        +"\t"+ tipo  +"\n");
       }

    public <U> void metodoConGenericos( U parametro ){
      System.out.println("Tipo y su valor: "
                         + parametro.getClass().getName() +"\t"+ parametro);
    }

    public static  <M> void metodoConGenericosEstatico( M variable ){
      System.out.println("Tipo y su valor: "
                          + variable.getClass().getName() +"\t"+ variable);
    }
}
```

Fichero: Ejemplo05.java

```
package fp2.poo.practica09;

public class Ejemplo05 {
    public static void main( String args[] ){
      Caja05 ref01 = new Caja05();
      Caja05 ref02 = new Caja05(new Long(10L));
      Caja05 ref03 = new <Long>Caja05(new Long(10L));

      System.out.println("Con metodo no estático.\n");
      ref01.<String> metodoConGenericos( "Fundamentos de Programacion II");
      ref01.<Integer>metodoConGenericos( new Integer( 10)   );
      ref01.<Double> metodoConGenericos( new Double ( 10.0) );

      System.out.println();
      System.out.println("Con metodo estático.\n");

      Caja05.<String> metodoConGenericosEstatico( "Fundamentos de Programacion II");
      Caja05.<Integer>metodoConGenericosEstatico( new Integer( 10)   );
      Caja05.<Double> metodoConGenericosEstatico( new Double ( 10.0) );
    }
}
```

El primer aspecto a destacar en este ejemplo es que la clase `Caja05` no contiene parámetro de tipo (es decir, no aparece como `Caja05<T>`) sino que los tipos son proporcionados a los métodos (estáticos o no estáticos) y a uno de los dos constructores de la clase para poder proporcionar un parámetro local.

Para ver el funcionamiento del programa ejecute la siguiente orden:

`make prueba05`

La forma de crear un objeto de la clase **`Caja05`**, es invocando a alguno de sus constructores.

Para invocar al primer constructor (**`Caja05()`**) no se proporciona ningún parámetro (**`Caja05 ref01 = new Caja05();`**).

Para invocar al segundo constructor (**`public <T> Caja05 ( T tipo )`**) se proporciona el tipo después de **`new`** y antes de invocar al constructor (es decir, **`Caja05 ref02 = new <Long>Caja05(new Long(10L));`**), esta operación se puede realizar también sin indicar el tipo después de **`new`** (**`Caja05 ref02 = new Caja05(new Long(10L));`**)

En este ejemplo se presenta además la invocación a dos métodos uno estático y otro no estático. En el caso del método no estático la invocación al método se realiza mediante la referencia al objeto proporcionando el tipo antes de indicar el método. En el caso del método estático se proporciona el nombre de la clase en lugar de la referencia al objeto.

Algunas restricciones al uso de genéricos

Existen algunas restricciones al uso de tipos genéricos y parte de ellas las mostramos en estos ejemplos. Para el uso efectivo de los tipos genéricos en Java, debemos tener en cuenta las siguientes consideraciones:

- No se pueden instanciar tipos genéricos con tipos de datos primitivos (**`int`**, **`long`**, **`boolean`**, **`short`**, **`float`**, **`double`** y **`char`**) en su lugar podemos utilizar sus correspondientes envolventes: **`Integer, Long, Boolean, Short, Float, Double`** y **`Character`**.

Fichero: Caja06.java

```
package fp2.poo.practica09;

public class Caja06<T>{

    private T t;

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

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

Fichero: Ejemplo06.java

```
package fp2.poo.practica09;

class Ejemplo06 {
    public static void main(String[] args) {
           // El uso de genéricos con tipos de datos simples
              // no está permmitido.
              // Estos ejemplos generarán error en compilación.

        Caja06<int>    caja01  = new Caja06<int>();
        Caja06<long>   caja02 = new Caja06<long>();
        Caja06<float>  caja03 = new Caja06<float>();
        Caja06<double> caja04 = new Caja06<double>();


             // Estos ejemplos NO generarán error en compilación.
        Caja06<Integer> caja01  = new Caja06<Integer>();
        Caja06<Long>    caja02 = new Caja06<Long>();
        Caja06<Float>   caja03 = new Caja06<Float>();
        Caja06<Double>  caja04 = new Caja06<Double>();

        caja01.add(new Integer (1));
        caja02.add(new Long    (2L));
        caja03.add(new Float   (3.f));
        caja04.add(new Double  (4.d));

             System.out.println(caja01.get());
             System.out.println(caja02.get());
             System.out.println(caja03.get());
             System.out.println(caja04.get());
    }
}
```

Para ver el funcionamiento del programa ejecute la siguiente orden:

`make prueba06`

Compruebe que el ejemplo genera error en compilación debido a que se están utilizando los tipos de datos simples. Comente las líneas de código que generan error. Recompile y ejecute.

- No se puede crear una instancia a partir de un tipo proporcionado como parámetro. Por tanto el siguiente ejemplo genera un error.

Fichero: Ejemplo07.java

```
package fp2.poo.practica09;

public class Ejemplo07 {
    public static void main(String[] args) {
            Ejemplo07.metodo();
    }
    public static <E> void metodo() {
        E elem = new E();  // error en compilacion
    }
}
```

Para ver el funcionamiento del programa ejecute la siguiente orden:

`make prueba07`

- No se puede declarar campos estáticos cuyos tipos son de tipo genérico.

Fichero: Caja08.java

```
package fp2.poo.practica09;

public class Caja08<T>{

    private static T atributoPrivado;      //ERROR
    private T objeto;

    public void agrega(T objeto) {
        this.objeto = objeto;
    }

    public T obtiene() {
        return objeto;
    }
}
```

Para ver el funcionamiento del programa ejecute la siguiente orden:

`make prueba08`

- No puede usarse el operador **`instanceof`** con tipos genéricos. Esto se debe a que el compilador de Java borra todos los tipos genéricos, y no se puede verificar qué tipo parametrizado se está utilizando al ejecutarse el código. Por tanto el siguiente código generará error en compilación.

Fichero: Caja09.java

```
package fp2.poo.practica09;

public class Caja09<T>{
    private T objeto;

    public Caja09( ){
        this.objeto = null;
    }

    public void agrega(T objeto) {
        this.objeto = objeto;
    }

    public T obtiene() {
        return objeto;
    }
}
```

Fichero: Ejemplo09.java

```
package fp2.poo.practica09;

public class Ejemplo09 {
    public static void main(String[] args)throws Exception  {
        Caja09<Integer> ref = new Caja09<Integer>();
        if(ref instanceof Caja09<Integer> ) {
           System.out.println("Error en compilacion...");
        }
        //if(ref instanceof Caja09<?> ) {
        //    System.out.println("No hay error en compilacion...");
        //}
    }
}
```

Para ver el funcionamiento del programa ejecute la siguiente orden:

`make prueba09`

En esta prueba generará un error en compilación al usar el operador `instanceof`.

No se puede saber a priori que la variable genérica **`T`** sea en el futuro un **`Integer`**. En su lugar podemos utilizar las **wildcards**. Comente la sentencia **`if`** que genera el error y descomente el **`if`** que se proporciona comentado, y pruebe su compilación.

- No pueden crearse matrices a partir de tipos parametrizados.

Fichero: Caja10.java

```
package fp2.poo.practica09;

public class Caja10<T>{

    private T objeto;

    public Caja10( ){
        this.objeto = null;
    }

    public void agrega(T objeto) {
        this.objeto = objeto;
    }

    public T obtiene() {
        return objeto;
    }
}
```

Fichero: Ejemplo10.java

```
package fp2.poo.practica09;

public class Ejemplo10 {
    public static void main(String[] args)throws Exception  {
        Caja10<Integer> ref [] = new Caja10<Integer>()[10];
    }
}
```

Para ver el funcionamiento del programa ejecute la siguiente orden:

`make prueba10`

Al compilarlo genera un error debido a que no se pueden crear matrices a partir de tipos parametrizados.

- No se puede crear (`new`), capturar (`catch`) o lanzar (`throw`) objetos de excepción de tipos parametrizados.

  Una clase genérica no puede heredar de la clase `Throwable`, directa o indirectamente.

Fichero: ExcepcionConGenericos.java

```
package fp2.poo.practica09;

class ExcepcionConGenericos<T> extends Exception {
    // error en tiempo de compilacion
    /* ... */
}
```

Para ver el funcionamiento del programa ejecute la siguiente orden:

`make prueba11`

Fichero: ThrowableConGenericos.java

```
package fp2.poo.practica09;

public class ThrowableConGenericos<T> extends Throwable {
    /* ... */
         // error en tiempo de compilacion
}
```

Para ver el funcionamiento del programa ejecute la siguiente orden:

`make prueba12`

En ambos casos generan error en compilación.

Sin embargo puede usarse un parámetro genérico en la sentencia **`throws`** del método:

Fichero: Caja13.java

```
package fp2.poo.practica09;

class Caja13<T  extends Exception> {
    private Integer matriz [];


    public void get() throws T {
             // Hacer algo...
                  matriz[0]= 1;
        }
}
```

Fichero: Ejemplo13.java

```
package fp2.poo.practica09;

public class Ejemplo13 {
    public static void main(String[] string) {
        Caja13<NullPointerException> obj
                                = new Caja13<NullPointerException>();
                try {
            obj.get();
             } catch (NullPointerException e) {
                     //Geston de la excepcion
                          System.out.println("Excepcion capturada");
                }
    }
}
```

Para ver el funcionamiento del programa ejecute la siguiente orden:

`make prueba13`

En este ejemplo la excepción se lanza en el método `get` de la clase `Caja13<T extends Exception>`, y se captura en el método `main`. La excepción se lanza ya que la matriz no está instanciada y se accede a la posición 0.

- No se pueden sobrecargar métodos que usan tipos parametrizados, esto es debido al borrado que se produce de la variable genérica. Suponiendo que la variable genérica es **`T`** ¿por qué tipo se sustituye? ¿Por **`String`** o **`Integer`**? No puede ser de los dos tipos.

Fichero: Caja14.java

```
package fp2.poo.practica09;

public class Caja14<T> {
    private T objeto;

    public Caja14( T objeto ){
        this.objeto = objeto;
    }

    public void agrega(T objeto) {
        this.objeto = objeto;
    }

    public T obtiene() {
        return objeto;
    }
}
```

Fichero: Ejemplo14.java

```
package fp2.poo.practica09;

class Utilidades {
        public static void imprime(Caja14<Integer> c){
            System.out.println(c.obtiene());
        }

        //public static void imprime(Caja14<String> c){
        //    System.out.println(c.obtiene());
        //}
}

public class Ejemplo14 {
    public static void main(String[] string) {
        Caja14<Integer> obj = new Caja14<Integer>(new Integer(7));
                Utilidades.imprime( obj );
    }
}
```

Para ver el funcionamiento del programa ejecute la siguiente orden:

`make prueba14`

Para probar la sobrecarga de método quite los comentarios al código y vuelva a ejecutarlo. Comprobará que se genera un error en compilación.

Ejercicio 1. Cree una clase genérica denominada "**`ExpendedoraGenerica`**" para ello:

- Cree tres atributos privados. Uno de los atributos de tipo **`Random`** (**`java.util.Random`**), otro de tipo int que se usa como contador de las inserciones que se realizan en una matriz, y el tercero la matriz (**`T listaElementos[]`**). En el constructor de la clase se proporciona la matriz instanciada.

- Implemente un método genérico "**`agregarElemento`**" (**`public void agregarElemento(T elemento)`**) que permita añadir un nuevo elemento a la lista.

- Implemente un método llamado "**`aleatorio`**" (**`public T aleatorio()`**) que devuelva un objeto de la lista de forma aleatoria. Utilice el método **`nextInt`** (clase **`Random`**) y ayúdese del tamaño de la lista para acotar el generador de números aleatorios.

- Cree una clase principal llamada "**`Main`**" y pruebe en ella que la expendedora genérica devuelve un elemento de forma aleatoria.

Wildcard

En un código genérico de Java, el signo de interrogación (**?**), representa un tipo desconocido. Los "**wildcard**" o comodines pueden ser utilizados en varias situaciones: como tipo de un parámetro, un atributo o una variable local; En ocasiones como un tipo de retorno.

Fichero: Caja15.java

```
package fp2.poo.practica09;

public class Caja15<T>{
    /** Variable genérica */
    private T t;

    /** Constructor */
    public  Caja15(T t){
      this.t = t;
      //System.out.println("T: " + t.getClass().getName());
    }

    /** Método que asigna un nuevo objeto */
    public void agrega(T t) {
       this.t = t;
    }

    /** Método que obtiene el objeto almacenado */
    public T obtiene() {
        return t;
    }
}
```

Fichero: Ejemplo15.java

```
package fp2.poo.practica09;

class Ejemplo15 {
    public static void main(String[] string) {
        Caja15<?> obj = new Caja15<Integer>(new Integer(7));
             System.out.println(obj.obtiene());
    }
}
```

Para ver el funcionamiento del programa ejecute la siguiente orden:

`make prueba15`

En este ejemplo se usa una referencia del tipo **`Caja15<?>`**, y se utiliza para obtener el objeto mediante el método **`obtiene()`**.

Limitar los tipos parametrizados

En ocasiones se desea limitar los tipos parametrizados que pueden ser usados como argumentos en un método. Por ejemplo, si un método opera con números podríamos aceptar únicamente instancias del tipo **`Number`** y sus subclases, estamos, por lo tanto, especificando un tipo paramétrizado limitado. Las limitaciones pueden ser de dos tipos:

- **Limitación superior:** Se denota por **`<? extends T>`** y permite el tipo **`T`** y a los tipos que herede de **`T`** (todos sus descendientes).

- **Limitación inferior:** Se denota por **`<? super T>`** y permite el tipo **`T`** y los tipos que son superclase de **`T`** (todos sus ascendentes).

Fichero: Ejemplo16.java

```
package fp2.poo.practica09;
public class Caja16<T>{
    /** Variable genérica */
    private T t;

    /** Constructor */
    public  Caja16(){
      this.t = null;
    }

    /** Constructor */
    public  Caja16(T t){
      this.t = t;
      //System.out.println("T: " + t.getClass().getName());
    }

    /** Método que asigna un nuevo objeto */
    public void agrega(T t) {
       this.t = t;
    }

    /** Método que obtiene el objeto almacenado */
    public T obtiene() {
        return t;
    }
}
```

Fichero: Ejemplo16.java

```
package fp2.poo.practica09;

public class Ejemplo16 {
    public static void main (String args[]){
     Caja16<Integer> obj1 = new Caja16<Integer>();
     Caja16<Integer> obj2 = new Caja16<Integer>(new Integer(12));
     System.out.println("Valor :" + obj1.obtiene());
     System.out.println("Valor :" + obj2. obtiene ());
     obj1.agrega(new Integer(15));
     System.out.println("Valor :" + obj1. obtiene ());

     Caja16<? extends Number> obj3 = new Caja16<Long>(new Long(10L));
     Caja16<? super   String> obj4 = new Caja16<String>(new String());
     obj3 = obj1;
     //obj3 = obj4; //ERROR
    }
}
```

Para ver el funcionamiento del programa ejecute la siguiente orden:

```
make prueba16
```

Descomente la línea `//obj3 = obj4; //ERROR`

Y compruebe que se genera un error al ejecutar el programa.

Borrado de tipos en la compilación

Los genéricos en el lenguaje Java proporciona comprobaciones más estrictas a la hora de compilar y permite soportar la programación genérica. Para implementar la programación genérica, el compilador de Java reemplaza todos los tipos paramétricos genéricos por los correspondientes objetos o tipos ligados. El bytecode producido, contiene únicamente clases, interfaces y métodos ordinarios, sin distinguir ningún tipo genérico.