```
import threading,socket,time
class sock(threading.Thread):
    def __init__(self):
        self.sck=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self,addr,port,func):
        try:
            self.sck.connect((addr,port))
            self.handle=self.sck
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error:Could not connect"
    def listen(self,host,port,func):
        try:
            self.sck.bind((host,port))
            self.sck.listen(5)
            self.todo=1
            self.func=func
            self.start()
        except:
            print "Error:Could not bind"
    def run(self):
        while self.flag:
            if self.todo==1:
                x,ho=self.sck.accept()
                self.todo=2
                self.client=ho
                self.handle=x
            else:
                dat=self.handle.recv(4096)
                self.data=dat
                self.func()
    def send(self,data):
        self.handle.send(data)
    def close(self):
        self.flag=0
        self.sck.close()
```

DAM/DAW

PROGRAMACIÓN

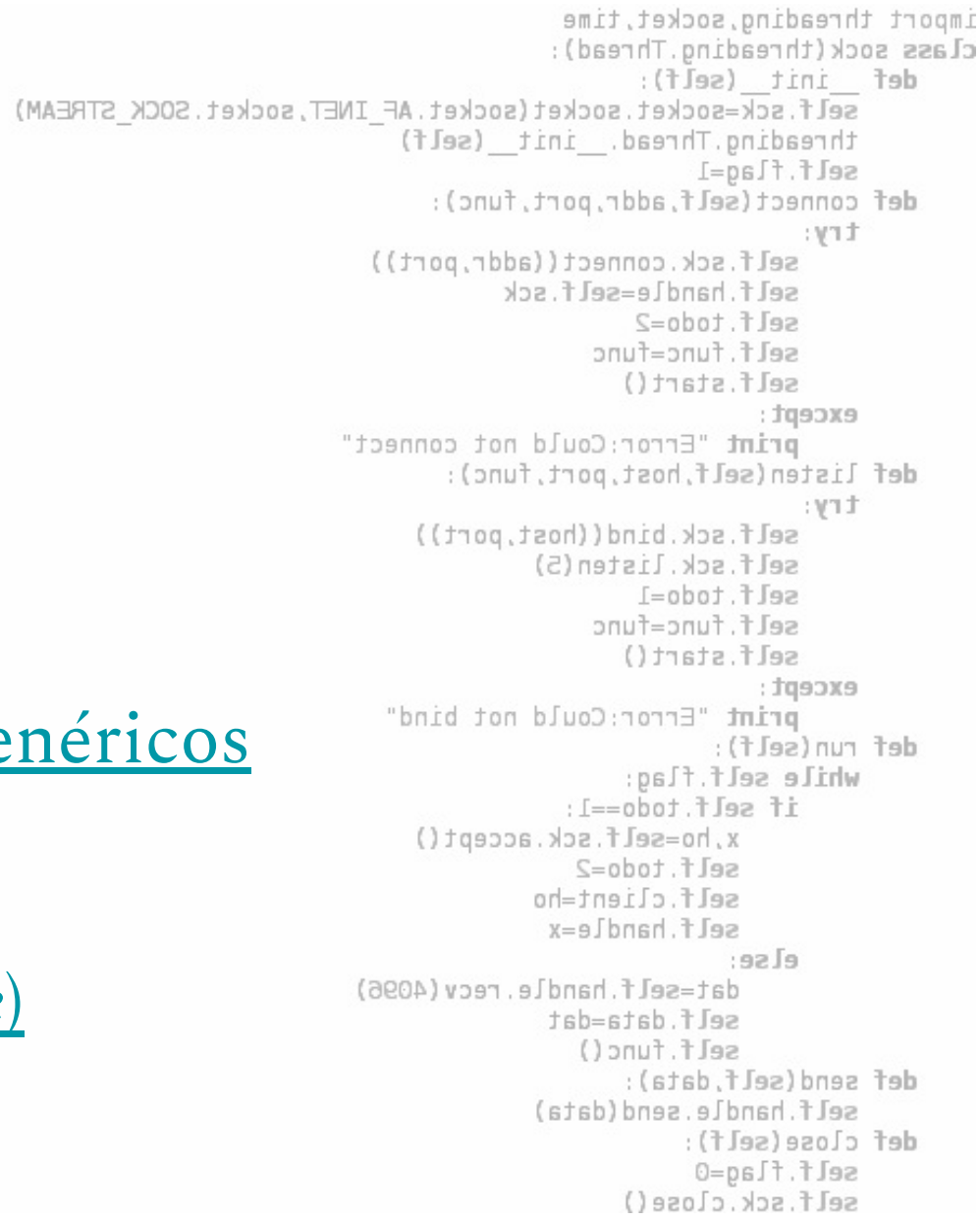
08.1

Genéricos

Rev: 3.2

fran@iessanclemente.net

Indice



Introducción (I)

- Estaría bien poder disponer de un método que nos permitiera, por ejemplo, ordenar el contenido de un array con independencia de que su contenido fueran enteros, boleanos, cadenas o cualquier otro objeto...
- En el fondo, esto es lo que representan los genéricos. La declaración de tipos parametrizados que nos permitirá crear clases, interfaces y métodos en los que el tipo de dato sobre el que operan se especifica como un parámetro más. Estas nuevas clases, interfaces y métodos se denominarán *clases genéricas*, *interfaces genéricos* y *métodos genéricos*
- La principal ventaja de los genéricos es que, de forma automática, trabajarán con el tipo de dato pasado como parámetro. Muchos algoritmos son independientes del tipo de dato sobre el que operan. Por ejemplo, *Quicksort* es el mismo, tenga que ordenar *Integer*, *String*, *Object* o *Thread*. Así, podremos definir nuestro algoritmo una vez y aplicarlo sobre diferentes datos sin modificaciones ni código redundante

Introducción (y II)

- Java siempre tuvo la habilidad de crear clases, interfaces y métodos generalizados mediante el empleo del tipo *Object*. Al ser superclase de cualquier otra clase, nuestra o del API, una variable de tipo *Object* puede referenciar cualquier tipo de objeto. Sin embargo, esta forma de operar es lo que se conoce como *non-Type-Safe*. Esto es debido a que, nuestra clase o método generalizado, debía hacer un *cast* explícito en tiempo de ejecución para convertir la referencia desde *Object* al tipo que nos interese. Existe, por tanto, el riesgo de que el objeto referenciado no sea del tipo esperado al hacer el *cast*, lo que produciría el consiguiente error de *type mismatch*
- Por contra, el empleo de clases y métodos genéricos será *Type-Safe*
- La inclusión de los genéricos supuso una enorme transformación del lenguaje Java, pues trajo consigo la incorporación de una nueva sintaxis y numerosas modificaciones en las clases y métodos de su API

Un ejemplo simple (I)

- Veamos un ejemplo simple de un método genérico
- Supongamos que deseamos definir un método que nos permita imprimir arrays de diferentes tipos de datos: enteros, caracteres, cadenas,...
- Hasta ahora, esto lo conseguíamos mediante la *sobrecarga* de métodos. Por cada tipo de dato, añadíamos a nuestra clase una nueva versión del método que esencialmente era la misma que las otras, con la salvedad de que el dato sobre el que operaba era diferente:

```
//...
public static void imprimeArray(int[] arr) {
  for (int val: arr) System.out.printf("%s ", val);
  System.out.println();
}
public static void imprimeArray(double[] arr) {
  for (double val: arr) System.out.printf("%s ", val);
  System.out.println();
}
```

Un ejemplo simple (II)

- Una simple clase de prueba...

```
public class DemoGen1 {
  public static void imprimeArray(int[] arr) {
    for (int val: arr) System.out.printf("%s ", val);
    System.out.println();
  }
  public static void imprimeArray(double[] arr) {
    for (double val: arr) System.out.printf("%s ", val);
    System.out.println();
  }
  public static void main(String[] args) {
    double[] a1 = { 3.5, 2.0, 4, -1.67 };
    int[] a2 = { 5, 0, 4, -1 }
    imprimeArray(a1);
    imprimeArray(a2);
  }
}
```

```
3.5 2.0 4.0 -1.67
5 0 4 -1
```

Un ejemplo simple (III)

- Por cada nuevo tipo de dato que deseémos contemplar, debemos añadir una versión específica del método anterior. Y si, por alguna razón, tuviéramos que modificar su implementación (corregir un error, nueva funcionalidad,...) tendríamos modificar cada uno de ellos!!
- Veamos cómo se resuelve esta situación con un método genérico...
- La idea fundamental es, en lugar de indicar el tipo de dato, indicamos un tipo "genérico" (le llamaremos T) y realizaremos la implementación con él. Durante la ejecución, será como si dicho tipo genérico se "sustituyera" por el tipo correcto correspondiente

```
public static <T> void imprimeArrayGen(T[] arr) {
  for (T val: arr) System.out.printf("%s ", val);
  System.out.println();
}
```

Un ejemplo simple (y IV)

- Así, nuestra clase con el método genérico quedaría...

```
public class DemoGen2 {
  public static <T> void imprimeArray(T[] arr) {
    for (T val: arr) System.out.printf("%s ", val);
    System.out.println();
  }
  public static void main(String[] args) {
    Double[] a1 = { 3.5, 2.0, 4, -1.67 };
    Integer[] a2 = { 5, 0, 4, -1 }
    String[] a3 = { "mi", "casa, ", "teléfono " };
    imprimeArray(a1);
    imprimeArray(a2);
    imprimeArray(a3);
  }
}
```

Fíjate que con los genéricos no podemos usar tipos primitivos, tenemos que usar tipos referenciados

Podemos pasarle a nuestro único método genérico *arrays* de cualquier tipo!

```
3.5 2.0 4.0 -1.67
5 0 4 -1
mi casa, teléfono
```

Clases genéricas (I)

- Una clase genérica será aquella en la que parte de su estructura se define mediante tipos parametrizados
- La sintaxis para declarar una clase genérica es:

```
[modif] class nombreClase<lista_tipos_param> [extends ...] [implements ...] { ... }
```

donde *lista_tipos_param* será la lista de los tipos parametrizados empleados por la clase (separados por comas y encerrados entre < >). Esta lista no es más que una serie de letras (por convención en mayúsculas) que representan cada uno de los tipos parametrizados usados internamente

- La declaración de nuevas variables de dicha clase y la creación de nuevas instancias se hará ahora utilizando la siguiente sintaxis:

```
nombreClase<lista_tipos> nombre_variabe = new nombreClase<lista_tipos>(lista_argumentos)
```

donde *lista_tipos* será la lista (encerrada entre < >) de los tipos "reales" que usará la clase por cada tipo parametrizado en el nuevo objeto creado

Clases genéricas (II)

- Veamos un ejemplo simple. Crearemos una clase genérica donde uno de sus atributos (miVal) es de un tipo parametrizado.

```
public class ClaseGen<T> {

  T miVal;

  public ClaseGen(T val) {
    this.miVal = val;
  }

  public T getMiVal() {
    return this.miVal;
  }

  public String toString() {
    return "ClaseGen<" + this.miVal.getClass().getName() +  ">: miVal = " + this.miVal;
  }
}
```

Al declarar la clase genérica *ClaseGen*, indicamos que va a usar un tipo parametrizado (llamado *T*) que será sustituido por un tipo real cuando se cree un objeto de la clase

Declara una variable del tipo parametrizado *T*

Podemos usar los tipos parametrizados tanto en la lista de parámetros de un método como al especificar su valor de retorno

Clases genéricas (III)

- Probemos nuestra clase genérica...

```
public class DemoGen3 {
  public static void main(String[] args) {
    ClaseGen<Double> obj1 = new ClaseGen<Double>(23.75);
    double v1 = obj1.getMiVal();
    System.out.println("obj1 miVal = " + v1);

    ClaseGen<String> obj2 = new ClaseGen<String>("Hello!");
    String v2 = obj2.getMiVal();
    System.out.println("obj2 miVal = " + v2);

    System.out.println("obj1 = " + obj1);
    System.out.println("obj2 = " + obj2);
  }
}
```

Se crean nuevos objetos de nuestra clase empleando tipos diferentes (*Double* y *String*). El tipo parametrizado de la clase se sustituirá por el tipo suministrado, *Double* en un caso y *String* en el otro

Al invocar el método *getMiVal()*, el tipo del valor devuelto dependerá de la instancia concreta y del tipo real que se haya empleado al crearla

```
obj1 miVal = 23.75
obj2 miVal = Hello!
obj1 = ClaseGen<java.lang.Double>: miVal = 23.75
obj2 = ClaseGen<java.lang.String>: miVal = Hello!
```

Clases genéricas (IV)

- En el ejemplo anterior, T es el nombre del tipo parametrizado. Este nombre nos es más que una especie de *alias* que será sustituido por el tipo real que se le pase a la clase *ClaseGen* cuando un nuevo objeto de la misma sea creado. Por ejemplo:

```
ClaseGen<Double> obj1 = new ClaseGen<Double>(23.75);
```

- Fíjate que al declarar una nueva variable de la clase genérica *ClaseGen*, debemos indicar el tipo *"real"* que reemplazará, en esa variable concreta, al tipo parametrizado (en este caso, *Double*). Al crear el objeto, todas las referencias internas de la clase en atributos, variables, métodos,... al tipo parametrizado T serán sustituidas por referencias al tipo indicado

- Recuerda que los tipos genéricos sólo trabajan con tipos referenciados. No podemos emplear tipos primitivos

Clases genéricas (V)

- Crear clases genéricas con más de un tipo parametrizado es simple...

```
public class ClaseGen2<T, V> {
  T miVal1;
  V miVal2;

  public ClaseGen2(T val1, V val2) {
    this.miVal1 = val1;
    this.miVal2 = val2;
  }

  public T getMiVal1() { return this.miVal1; }
  public V getMiVal2() { return this.miVal2; }

  public String toString() {
    return "ClaseGen2<" + this.miVal1.getClass().getName() + ", " +
                this.miVal2.getClass().getName() + ">: miVal1 = " + this.miVal1 +
                ", miVal2 = " + this.miVal2;
  }
}
```

Al declarar la clase genérica *ClaseGen*, indicamos los diferentes tipos parametrizados empleados por la clase

Podemos usar los tipos parametrizados en cualquier lugar donde necesitemos especificar un tipo

Clases genéricas (y VI)

- Probemos nuestra nueva clase genérica de dos tipos parametrizados...

```
public class DemoGen4 {
  public static void main(String[] args) {
    ClaseGen2<Integer, String> obj1 = new ClaseGen2<Integer, String>(1, "Uno");
    System.out.println("obj1 miVal1 = " + obj1.getMiVal1());
    System.out.println("obj1 miVal2 = " + obj1.getMiVal2());
    ClaseGen2<String, Double> obj2 = new ClaseGen2<String, Double>("Spock", 2230.06);
    System.out.println("obj2 miVal1 = " + obj2.getMiVal1());
    System.out.println("obj2 miVal2 = " + obj2.getMiVal2());
    System.out.println("obj1 = " + obj1);
    System.out.println("obj2 = " + obj2);
  }
}
```

```
obj1 miVal1 = 1
obj1 miVal2 = Uno
obj2 miVal1 = Spock
obj2 miVal2 = 2230.06
obj1 = ClaseGen2<java.lang.Integer, java.lang.String>: miVal1 = 1, miVal2 = Uno
obj2 = ClaseGen2<java.lang.String, java.lang.Double>: miVal1 = Spock, miVal2 = 2230.06
```

Tipos acotados (I)

- En los ejemplo precedentes, los tipos parametrizados pueden reemplazarse por cualquier tipo de clase. Sin embargo, en ocasiones, necesitaremos restringir los tipos de clase que pueden ser pasados
- Por ejemplo, supón que quieres crear una clase genérica que almacene valores numéricos y sea capaz de realizar diferentes operaciones sobre ellos (calcular el inverso,...). ¿Cómo podríamos restringir que sólo se pudieran crear objetos utilizando clases como *Integer*, *Float* o *Double*?
- Para dar respuesta a estas soluciones, Java proporciona los denominados tipos acotados (*bounded types*). Cuando especifiquemos un tipo parametrizado, tendremos la posibilidad de declarar de qué *superclase* debe derivar un tipo de clase para que sea válido. Para ello, usaremos:

```
<T extends superclase>
```

superclase, clase (o *interface*) que debe extender (o implementar) el tipo

Tipos acotados (II)

- Vamos a modificar nuestra clase genérica *ClaseGen* para que sólo acepte tipos numéricos (clases que derivan de *java.lang.Number*)

```
public class ClaseGen<T extends Number> {
  T miVal;

  public ClaseGen(T val) { this.miVal = val; }
  public T getMiVal() { return this.miVal; }

  public double getDoubleVal() {
    return this.miVal.doubleValue();
  }
  public double getMiValInverse() {
    return 1/this.miVal.doubleValue();
  }
  public String toString() {
    return "ClaseGen<" + this.miVal.getClass().getName() +  ">: miVal = " + this.miVal;
  }
}
```

Estamos limitando los tipos posibles a las clases que deriven de *Number*

doubleValue() es un método de *Number*
Todas sus subclases lo heredan
De no haber introducido la limitación de tipos, tendríamos un error de compilación

Tipos acotados (III)

- Probemos nuestra nueva clase genérica...

```
public class DemoGen5 {
  public static void main(String[] args) {
    ClaseGen<Integer> obj = new ClaseGen<Integer>(5);
    int v = obj.getMiVal();
    double v_inv = obj.getMiValInverse();

    System.out.println("val = " + v);
    System.out.println("inverso = " + v_inv);

    System.out.println("obj = " + obj);

    //ClaseGen<String> obj2 = new ClaseGen<String>( "Hello!");
  }
}
```

El intento de ejecución de esta sentencia, produciría el siguiente error de compilación:
type argument java.lang.String is not within bounds of type-variable T

```
val = 5
inverso = 0.2
obj = ClaseGen<java.lang.Integer>: miVal = 5
```

Comodines y Tipos (I)

- Planteémonos una nueva situación. Supongamos que deseamos añadir a nuestra clase genérica un método que nos permita comparar el valor numérico almacenado con el de cualquier otra instancia de la clase:

```
public int compareTo(ClaseGen<T> obj) {
    double v1 = this.miVal.doubleValue();
    double v2 = obj.getDoubleVal();

    if(v1<v2)           //--> menor
       return -1;
    else if(v1>v2)      //--> mayor
       return 1;
    return 0;           //--> igual
}
```

- En principio, parece correcto...

Comodines y Tipos (II)

- Veamoslo en un ejemplo. Tras modificar nuestra clase, creamos nuevas instancias de la misma con diferente argumento tipo...

```
ClaseGen<Double> obj1 = new ClaseGen<Double>(5.75);
ClaseGen<Double> obj2 = new ClaseGen<Double>(3.5);
ClaseGen<Integer> obj3 = new ClaseGen<Integer>(8);
```

- En los dos primeros casos, el tipo parametrizado T se sustituirá por *Double*, mientras que en el segundo objeto, T se sustituye por *Integer*
- Veamos que pasa al comparar *obj1* con los otros dos objetos...

```
System.out.println("compara: obj1 y obj2 => " + obj1.compareTo(obj2));
System.out.println("compara: obj1 y obj3 => " + obj1.compareTo(obj3));
```

```
compara: obj1 y obj2 => 1
Error:
incompatible types: ClaseGen<java.lang.Integer> cannot be converted to
ClaseGen<java.lang.Double>
```

Comodines y Tipos (III)

- El problema viene de lo siguiente. Al declarar *obj1*, se indicó el tipo *Double como* argumento tipo para la clase genérica *ClaseGen<T>*

Como sabemos, esto es como si su clase *ClaseGen<T>*...

```
public class ClaseGen<T extends Number> {
  T miVal;
  public ClaseGen(T val) { this.miVal = val; }
  public int compareTo(ClaseGen<T> obj) {
      // ...
}
```

en tiempo de compilación se convirtiera en...

```
public class ClaseGen<Double> {
  Double miVal;
  public ClaseGen(Double val) { this.miVal = val; }
  public int compareTo(ClaseGen<Double> obj) {
      // ...
}
```

Comodines y Tipos (IV)

- Sin embargo, al crear *obj3* estamos pasando *Integer* como argumento tipo. Es decir, estamos creando un objeto de la clase *ClaseGen<Integer>*
- Por tanto, *obj1* tendría un método *compareTo( )* que aceptará instancias de *ClaseGen<Double>* pero no de *ClaseGen<Integer>*. De ahí, el error
- Java nos proporciona una nueva sintaxis con el empleo del comodín ? para sortear estas "incompatibilidades"
- Así, nuestro nuevo método quedaría de la siguiente manera...

```
public class ClaseGen<T extends Number> {
  // ...
  public int compareTo(ClaseGen<?> obj) {
      // ...
}
```

- El comodín ? le indica al compilador que esperamos una instancia de la clase genérica *ClaseGen<T>*, pero no de un argumento tipo concreto

Comodines y Tipos (y V)

- Ahora, al volver a ejecutar nuestro código...

```
System.out.println("compara: obj1 y obj2 => " + obj1.compareTo(obj2));
System.out.println("compara: obj1 y obj3 => " + obj1.compareTo(obj3));
```

```
compara: obj1 y obj2 => 1
compara: obj1 y obj3 => -1
```

- Por último, indicar que los argumentos comodín se pueden acotar del mismo modo que hacíamos con los argumentos tipo:

```
<? extends superclase>
```

Aquí, el comodín ? "enmascararía" cualquier objeto del tipo *superclase* o alguna de sus subclases. De igual modo, podemos "forzar" que sólo se acepten *superclases* de una clase concreta:

```
<? super subclase>
```

Métodos y Constructores genéricos (I)

- Como hemos visto en los ejemplos precedentes, los métodos internos de una clase genérica pueden hacer uso de sus tipos parametrizados y, por tanto, convertirse en métodos genéricos
- Por otro lado, tal como vimos en el primer ejemplo de esta unidad, una clase no genérica también puede contener métodos genéricos. En este caso, los tipos parametrizados deben ser declarados en el propio método
- La sintaxis para que un método declare sus propios tipos parametrizados:

```
[modif] <lista_tipos_param> tipo_retorno nombreMétodo (<lista_parámetros>){ ... }
```

  donde *lista_tipos_param* será la lista de los tipos parametrizados empleados por el método (separados por comas y encerrados entre < >). Fíjate que esta lista se coloca antes del tipo de retorno

- Esto mismo se aplica a los constructores de una clase no genérica

Métodos y Constructores genéricos (II)

- En el siguiente ejemplo, vamos a crear en primer lugar una clase genérica que nos permitirá almacenar parejas *Clave-Valor*

```
public class Pareja<K, V> {
  private K clave;
  private V valor;

  public Pareja(K clave, V valor) {
    this.clave = clave;
    this.valor = valor;
  }

  public K getClave() { return this.clave; }
  public T getValor() { return this.valor; }

  public void setClave(K clave) { this.clave = clave; }
  public void setValor(V valor) { this.valor = valor; }

  public String toString() { return "Pareja [" + this.clave + ", " + this.valor + "]"; }
}
```

La clase genérica declara los tipos parametrizados K y V
Cualquier método de la clase puede hacer uso de dichos tipos sin necesidad de declararlos

Métodos y Constructores genéricos (y III)

- Creamos ahora una clase no genérica pero con un método genérico

```
public class Util {
  public static <K, V> boolean compara(Pareja<K,V> p1, Pareja<K,V> p2) {
    return p1.getClave().equals(p2.getClave()) && p1.getValor().equals(p2.getValor());
  }
}
```

Necesitamos declarar los tipos parametrizados en el propio método

- Probemos nuestras clases...

```
public class DemoGen6 {
  public static void main(String[] args) {
    Pareja<Integer, String> p1 = new Pareja<Integer, String>(1, "manzana");
    Pareja<Integer, String> p2 = new Pareja<Integer, String>(2, "pera");
    boolean iguales = Util.<Integer, String>compara(p1, p2);
    System.out.println(p1 + " igual a " + p2 + " ? " + iguales);
  }
}
```

Desde JDK7, ya no es necesario añadir en la llamada al método los tipos parametrizados (se infieren a partir del tipo de sus argumentos)

```
Pareja [1, manzana] igual a Pareja [2, pera] ? false
```

Interfaces genéricos (I)

- Los interfaces genéricos se especifican del mismo modo que las clases genéricas. Por ejemplo:

```
public interface Mutable<T> {
  void setValue(T obj);
}
```

- En general, si una clase implementa un interfaz genérico, también será genérica ya que, como mínimo, debe declarar el tipo parámetrizado que será pasado al interfaz en el momento de la creación de un nuevo objeto. En el caso del interfaz anterior, una clase se podría declarar así:

```
public class Contenedor<V> implements Mutable<V> { //... }
```

- Una declaración como la siguiente, generaría un error de compilación:

```
public class Contenedor implements Mutable<T> { //... } -->> ERROR!!
```

Interfaces genéricos (y II)

- Completemos nuestra clase...

```
public class Contenedor<T> implements Mutable<T> {
  private T obj;
  public T getValue() { return this.obj; }
  @Override
  public void setValue(T obj) { this.obj = obj; } // método del interfaz
}
```

- Por supuesto, siempre podremos crear una clase no genérica que implemente una versión del interfaz de un tipo específico...

```
public class ContenedorStr implements Mutable<String> {
  private String obj;
  public String getValue() { return this.obj; }
  @Override
  public void setValue(String obj) { this.obj = obj; }
}
```

Sustituimos las referencias a tipos parametrizados por un tipo específico

Borrado de tipo (I)

- Como hemos visto, la nueva sintaxis de genéricos se introdujo en Java para dar soporte a la programación genérica y proporcionar controles estrictos del tipado en tiempo de compilación.
- A la hora de implementar este nuevo mecanismo en Java, uno de los principales requerimientos de diseño era el de mantener la compatibilidad con todo el código existente, tanto fuentes como binarios
- Finalmente, la implementación de esta nueva característica del lenguaje se basó en lo que se conoce como borrado de tipo (*type erasure*). En líneas generales, el compilador realizará las siguientes acciones:
    - En el bytecode generado, se eliminan todas las referencias a tipos parametrizados. Estos tipos se reemplazan por *Object* o el *supertipo* en caso de usar tipos acotados. Así, las clases generadas, sólo contendran código ordinario y, por tanto, compatible con cualquier código Java anterior a la introducción de los genéricos

Borrado de tipo (II)

- Cuando sea necesario, el compilador añadirá *casts* para asegurar que el tipado sea correcto (*type safe*)
- Con objeto de garantizar el polimorfismo al extender tipos genéricos, el compilador genera lo que se denomina métodos puente (*bridge*)

- Por ejemplo, tras la compilación, el *bytecode* de nuestro interfaz genérico *Mutable* quedaría así:

código fuente

```
public interface Mutable<T> {
  void setValue(T obj);
}
```

En el *bytecode* generado, las referencias al tipo parametrizado se eliminan de la declaración del interfaz y los tipos se convierten a Object

```
public interface Mutable {
  void setValue(Object obj);
}
```

bytecode

Borrado de tipo (III)

- Nuestra clase *ContenedorStr<String>* se transformará así...

código fuente

```
public class ContenedorStr<String> implements Mutable<String> {
  private String obj;
  public String getValue() { return this.obj; }
  @Override
  public void setValue(String obj) { this.obj = obj; }
}
```

bytecode

```
public class ContenedorStr implements Mutable {
  private String obj;
  public String getValue() { return this.obj; }
  public void setValue(String obj) { this.obj = obj; }

  // método puente creado por el compilador para mantener el polimorfismo
  public void setValue(Object obj) { this.set((String) obj);
  }
}
```

Borrado de tipo (IV)

- Fíjate como, al borrar los tipos parametrizados en el *bytecode* y sustituirlos por *Object*, las firmas del método *setValue()* del interfaz y la clase que lo implementa no coinciden:

```
void setValue(Object obj); // interfaz Mutable
```

```
void setValue(String obj); // clase ContenedorStr
```

```
Son métodos distintos!!
Sería un método
sobrecargado,
no sobreescrito
```

- El compilador lo soluciona añadiendo un nuevo método (*bridge method*) que sobreescribe el método del interfaz e invoca al método de la clase tras realizar el *cast* correspondiente del argumento

- Esto se puede comprobar facilmente desensamblando el archivo *.class*
  A título ilustrativo, se muestra en la siguiente página el desensamblado del bytecode contenido en el archivo *ContenedorStr.class*, empleando para ello la utilidad **javap** del JDK

Borrado de tipo (y V)

- Desensamblado del archivo *ContenedorStr.class*

```
bowman@hal:~$ javap -c ContenedorStr.class
Compiled from "ContenedorStr.java"
public class ContenedorStr implements Mutable<java.lang.String> {
  ...
  public void setValue(java.lang.String);
    Code:
       0: aload_0
       1: aload_1
       2: putfield      #2                  // Field obj:Ljava/lang/String;
       5: return
  public void setValue(java.lang.Object);
    Code:
       0: aload_0
       1: aload_1
       2: checkcast     #3                  // class java/lang/String
       5: invokevirtual #4                  // Method setValue:(Ljava/lang/String;)V
       8: return
}
```

Método *setValue(String)* resultante de compilar el método definido por la clase

Bridge setValue(Object) añadido por el compilador para sobreescribir el método del interfaz

cast del argumento a String

llamada a *setValue(String)*

Algunas notas finales (I)

❖ Inferencia de tipos

- Supongamos nuestra anterior clase genérica *Pareja*:

```
public class Pareja<C, V> { //... }
```

- Hasta JDK7, para crear instancias de dicha clase, debíamos indicar los tipos argumento al crear el nuevo objeto tanto en la declaración de la variable como con el operador *new*

```
Pareja<Integer, String> p1 = new Pareja<Integer, String>(1, "manzana");
Pareja<Integer, String> p2 = new Pareja<Integer, String>(2, "pera");
```

- A partir de esa versión, Java es capaz de inferir los tipos parametrizados a partir de los argumentos pasados al propio constructor, por lo que la sintaxis se simplifica. El ejemplo anterior podría reescribirse así:

```
Pareja<Integer, String> p1 = new Pareja<>(1, "manzana");
Pareja<Integer, String> p2 = new Pareja<>(2, "pera");
```

Fíjate que se mantienen los <>

Algunas notas finales (II)

❖ Errores de ambigüedad

- La inclusión de los genéricos da lugar a nuevos problemas que deberemos tener en cuenta al diseñar nuestras clases. Fíjate en el siguiente ejemplo:

```
public class ClaseGen<T, V> {
  private T obj1;
  private V obj2;
  public void set(T obj) { this.obj1 = obj; }
  public void set(V obj) { this.obj2 = obj; }
}
```

La sobrecarga del método *set()* es ambigüa
La solución pasa por crear dos métodos de nombres diferentes

- No existe ningún requerimiento que impida que T y V sean diferentes. Por tanto, podríamos crear un nuevo objeto así:

```
ClaseGen<Integer, Integer> obj = new ClaseGen<>(1, 2);
```

- Esto haría, erróneamente, que ambos métodos *set( )* tuvieran la misma firma (lo mismo sucedería al aplicar el mecanismo de *type erasure*)

Algunas notas finales (III)

❖ Restricciones

- No podemos crear instancias de un tipo parametrizado

```
public class ClaseGen<T> {
  private T obj1;
  public ClaseGen() { this.obj1 = new T(); }   // --> Error!!
}
```

- Ningún miembro estático (variable o método) de una clase genérica puede usar los tipos parametrizados declarados por la clase

```
public class ClaseGen<T> {
  static T obj1;   // --> Error!!
  static T getObj() { return obj1; }   // --> Error!!
}
```

Eso sí, como ya vimos anteriormente, un método estático genérico puede definir sus propios tipos parametrizados

Algunas notas finales (y IV)

- No podemos crear arrays de tipos parametrizados

```
public class ClaseGen<T> {
  private T[] arr;
  public ClaseGen(int n) { this.obj1 = new T[n]; }   // --> Error!!
}
```

- No podemos crear arrays de referencias genéricas de un tipo específico

```
ClaseGen<Integer>[] arr = new ClaseGen<>[10];   // --> Error!!
```

  En situaciones como las anteriores, aunque podríamos diseñar nuestra propia clase para manejar *arrays* de clases genéricas, lo más fácil es recurrir a las clases del *Collections Framework* del API de Java

- Y, por último, una clase genérica no puede extender *Throwable*. Esto implica que no podemos crear clases genéricas que puedan ser usadas como excepciones