Estructuras de Datos y Algoritmos.
Unidad Didáctica I: Conceptos de Java para Estructuras de Datos.

Tema 3. Genericidad

Mabel Galiano Natividad Prieto

Departamento de Sistemas Informáticos y Computación
Escuela Técnica Superior de Ingeniería Informática

Índice

Objetivos y Bibliografía

La Genericidad es el complemento indispensable que necesita la Herencia para lograr una Reutilización efectiva del Software. Por ello, el objetivo principal de este tema es introducir los mecanismos que proporciona el lenguaje Java para la implementación de la Genericidad desde su versión `1.5`, esto es la terminología, sintaxis e interpretación asociada a las clases y métodos Java de Tipo parametrizado o genérico. Pero también, como en adelante se combinará el uso de la Herencia y la Genericidad para el diseño y uso en Java de las diferentes Estructuras de Datos (EDAs) que se introducen en esta asignatura, con esta tema se pretenden ilustrar suficientemente los motivos por los que las clases y métodos Java de Tipo parametrizado conllevan una mayor Reutilización del Software y, adicionalmente, un incremento de su legibilidad y fiabilidad.

En función de estos objetivos, el tema se estructura en tres apartados. En el primero de ellos se introduce el concepto de Genericidad y los motivos por los que es considerado junto con la Herencia un pilar de la Programación Orientada a Objetos (POO). En el segundo se presenta la forma en la que se implementan en Java a partir de la versión `1.5` las clases y métodos genéricos o de Tipo parametrizado -ámbito, declaración y uso de parámetros de tal tipo; todas las novedades que incorpora este apartado se ilustran con el estudio de ejemplos concretos, algunos extraídos del estándar de Java (`ArrayList` y `Comparable`).

Finalmente en el último apartado del tema se introducen las clases y métodos de Tipo genérico Restringido, o Acotado superiormente por otro, un caso particular de los estudiados en el segundo apartado si se considera que cualquier tipo o clase Java deriva de `Object`, i.e. está Restringido o Acotado superiormente por `Object`; su uso se ilustrará presentando un ejemplo de gran relevancia: la resolución del problema de la Ordenación de un `array` de Tipo genérico Restringido por `Comparable`.

Como bibliografía básica del tema se puede consultar el documento **Learning the Java Language. Generics**, que se puede encontrar publicado por *Sun Microsystems* en la dirección `http://java.sun.com/docs/books/tutorial/java/generics/index.html`.

NOTA. Las clases Java que aparecen en este tema se han ubicado en diferentes paquetes del proyecto BlueJ `eda`; en concreto,

- la clase `Ordenacion` se ha ubicado en el (nuevo) paquete `operacionesArray`, incluido a su vez en el subpaquete `util` de `librerias` de `eda`;
- las restantes clases se han ubicado en diversos subpaquetes de `tema3`, incluido a su vez en el paquete `ejemplos` de `eda`.

1. El papel de la Genericidad en la Reutilización del Software

En el Tema 1 se presentó la Herencia como el principal mecanismo de Reutilización del Software que proporciona un lenguaje POO; en particular se estudió cómo la definición de métodos abstractos en clases e interfaces Java permite especificar parte o toda la funcionalidad de éstas independientemente de su posterior implementación.

Ahora bien, para lo que no sirve la Herencia es para lograr independizar el diseño de una clase (o un método) del tipo de los Datos que usa como atributos (o parámetros). Así, por ejemplo, si se dispone de una interfaz `GrupoDeFiguras` y se quiere reutilizar para diseñar otra interfaz `GrupoDeString`, o `GrupoDeInteger` o `GrupoDeEmpleados, el método a seguir es el ya conocido "copy-paste & substitute-all" Figura` por `String`, o `Integer` o `Empleado`; claro está, además hay que descartar aquellos métodos de `GrupoDeFiguras` que no son aplicables a cualquiera de los Datos del resto de Grupos, como por ejemplo `area()`. Por lo tanto, **si se quiere dar un paso más hacia la efectiva Reutilización del Software es necesaria una Programación Genérica**, i.e. en la que el `Tipo` de los Datos se incluya como un parámetro `T` más en la cabecera de definición de una clase o un método -de ahí que el término parametrización también se suela utilizar con frecuencia en estos casos. De esta forma, nótese, el problema de la interfaz `GrupoDeFiguras` se puede resolver de una forma más segura y eficiente con el siguiente procedimiento:

- Diseñar primero y únicamente una interfaz genérica `GrupoDeDatos<T>` como la que se presenta a continuación, i.e. una interfaz cuyo tipo de Datos `T` es el «Factor Común» de los tipos de Datos concretos `Figura`, `String`, `Integer` y `Empleado`;

```
public interface GrupoDeDatos<T> {
  /** devuelve la talla de un Grupo de Datos de tipo T**/
  int talla();
  /** inserta el Dato d de tipo T en un Grupo **/
  void insertar(T d)
  /** devuelve el Dato de tipo T que ocupa la i-ésima posición del Grupo,
    * o lanza ElementoNoEncontrado para advertir que i no es una posición válida **/
  T recuperar(int i) throws ElementoNoEncontrado;
  /** devuelve la posición de la primera componente de un Grupo tal que equals(d),
    * o devuelve -1 si d no está en el Grupo **/
  int indiceDe(T d);
  /** elimina la primera componente de un Grupo tal que equals(d) y devuelve true,
    * o devuelve false si d no está en el Grupo **/
  boolean eliminar(T d);
  /** devuelve un array con los talla() Datos de tipo T de un Grupo **/
  T[] toArray();
}
```

- Instanciar después -o substituir literalmente- el `Tipo` genérico `T` de la interfaz `GrupoDeDatos` por uno concreto, `Figura` o `String` o `Integer` o `Empleado`, para obtener la interfaz correspondiente a dicho tipo de Datos, `GrupoDeDatos<Figura>` o `GrupoDeDatos<String>` o `GrupoDeDatos<Integer>` o `GrupoDeDatos<Empleado>` respectivamente.

Yendo aún más lejos, dado que el soporte en memoria de los Datos de un Grupo y las estrategias de Exploración de éstos (Recorrido y/o Búsqueda) son independientes de su tipo concreto,

la Implementación de una interfaz genérica como `GrupoDeDatos` no puede ser más que genérica o parametrizada por el Tipo genérico `T`, un Esquema de Clase (*Template*) cuyas instancias se obtienen de nuevo substituyendo literalmente `T` por el nombre de un tipo concreto -`Figura`, `String`, `Integer` o `Empleado`; así por ejemplo, una Implementación Contigua de `GrupoDeDatos`, o que utiliza un `array` de tipo genérico `T` como soporte de Datos en memoria, sería como sigue:

```
public class ArrayGrupoDeDatos<T> implements GrupoDeDatos<T> {
  // un Grupo de Datos de tipo T representado mediante un array TIENE UN
  private T elArray[];
  private int talla;
  private static final int CAPACIDAD_POR_DEFECTO = 10;

  /** construye un Grupo vacío, i.e. con cero Datos de tipo T **/
  public ArrayGrupoDeDatos(){...}
  /** devuelve la talla de un Grupo de Datos de tipo T */
  public int talla(){ return this.talla; }
  /** inserta el Dato d de tipo T en un Grupo **/
  public void insertar(T d){
      if ( this.talla == this.elArray.length ) duplicarArray();
      this.elArray[this.talla] = d;
      this.talla++;
  }
  private void duplicarArray(){...}
  /** devuelve el Dato de tipo T que ocupa la i-ésima posición del Grupo,
    * o lanza ElementoNoEncontrado para advertir que i no es una posición válida **/
  public T recuperar(int i) throws ElementoNoEncontrado{
      if ( 0 > i || i >= this.talla )
      throw new ElementoNoEncontrado("Al recuperar: no existe un Dato en la posición "+i);
      return this.elArray[i];
  }
  /** devuelve la posición de la primera componente de un Grupo tal que equals(d),
    * o devuelve -1 si d no está en el Grupo  **/
  public int indiceDe(T d){...}
  /** elimina la primera componente de un Grupo tal que equals(d) y devuelve true,
    * o devuelve false si d no está en el Grupo **/
  boolean eliminar(T d){...}
  /** devuelve un array con los talla() Datos de tipo T de un Grupo **/
  public T[] toArray(){...}
}
```

Un segundo ejemplo que ilustra de forma paradigmática las ventajas de la Programación Genérica es el **diseño de un método genérico de Ordenación**, i.e. el diseño de un método que permita ordenar los Datos de un `array a` de tipo genérico `T` y talla dada `a.length`, y por tanto cualquier `array` de tipo concreto que lo instancie -`Figura`, `String`, `Integer`, `Empleado`, etc. Recuérdese que hasta la fecha, y a pesar de conocer tres estrategias genéricas para hacerlo (**Inserción**, **Selección** e **Intercambio**), sólo se han podido diseñar métodos *ad hoc* para cada tipo de Datos concreto a ordenar y, atención, para cada criterio de comparación de Datos establecido para dicho tipo. Así por ejemplo, en la asignatura Programación se usó el siguiente método Java para ordenar por Inserción un `array a` de `int`:

```
public static void insercionDirecta(int a[]){
  for( int i = 1; i < a.length ; i++ ){
    int aIns = a[i]; int posIns = i;
    for(;posIns>0 && aIns < a[posIns-1];posIns--) a[posIns]=a[posIns-1];
    a[posIns] = aIns;
  }
}
```

Asimismo, cuando este curso se ha empleado la misma estrategia de Inserción pero para ordenar por área un `array` de tipo `Figura` se ha usado el siguiente método, una copia del anterior *salvo* por dos substituciones: la del tipo de Datos a ordenar, `int` por `Figura`, y la del criterio de comparación asociado al tipo de Datos, valor `int` por valor del área.

```
public static Figura[] insercionDirecta(Figura a[]){
  for( int i = 1; i < a.length ; i++ ){
    Figura aIns = a[i]; int posIns = i;
    for(;posIns>0 && aIns.area() < a[posIns-1].area();posIns--) a[posIns]=a[posIns-1];
    a[posIns] = aIns;
  }
}
```

Obsérvese que si en lugar de un `array` de tipo `Figura` se quisiera ordenar uno de tipo `Empleado`, bastaría de nuevo con copiar el método `insercionDirecta` que ordena un `array` de Figuras y substituir en él `Figura` por `Empleado` y el criterio de comparación de `Figura` por su equivalente en `Empleado`, por ejemplo el de antigüedad en la empresa que obtiene el método `antig()`; se tendría entonces el siguiente método de Ordenación de Empleados por antigüedad:

```
public static void insercionDirecta(Empleado a[]){
  for( int i = 1; i < a.length ; i++ ){
    Empleado aIns = a[i]; int posIns = i;
    for(;posIns>0 && aIns.antig() < a[posIns-1].antig();posIns--) a[posIns]=a[posIns-1];
    a[posIns] = aIns;
  }
}
```

Con los distintos diseños presentados para un mismo método de Ordenación se pretende evidenciar que los dos elementos que resultan suficientes y necesarios para diseñar en Java un método genérico de Ordenación de un `array` son:

1. **un Tipo genérico `T` que represente el tipo de los Datos del `array` a ordenar**, del que por ejemplo serían instancias los tipos concretos `Figura` y `Empleado`;
2. **un método que represente el criterio de comparación de dos objetos de Tipo genérico `T`**, cuya especificación y perfil podrían ser los siguientes:

   ```
   /** devuelve el resultado de comparar un Dato de tipo T con otro, un valor int
    *  menor que cero si this es menor que otro, mayor que cero si this es mayor
    *  que otro e igual a cero si this es igual a otro **/
   int compararCon(T otro);
   ```

   Obviamente este método sólo puede ser abstracto, debiéndose sobrescribir por tanto obligatoriamente en cada clase que instancie a `T`; por ejemplo, en la clase `Empleado` se tendría: `public int compararCon(Empleado otro) return (this.antig() - otro.antig());`

   Tampoco resulta difícil deducir que si `compararCon` se define en una clase `interface` entonces puede ser heredado por cualquier clase que instancie `T`, `Figura` o `Empleado`, independientemente de si dicha clase hereda o no otras propiedades de más de una superclase distinta de `Object`.

En resumen, disponiendo de los dos elementos que se acaban de enunciar, el código para la Ordenación de un `array` de tipo genérico `T` por Inserción Directa podría ser el siguiente:

```
for( int i = 1; i < a.length ; i++ ){
  T aIns = a[i]; int posIns = i;
  for(;posIns>0 && aIns.compararCon(a[posIns-1]) < 0;posIns--) a[posIns]=a[posIns-1];
  a[posIns] = aIns;
}
```

Nótese que el código presentado se puede ejecutar para tantas instancias de `T` como se quiera gracias la combinación de Genericidad y Enlace Dinámico.

Recapitulando ya para concluir, la clave para el diseño de clases y métodos genéricos en Java es la definición de un tipo genérico de Datos `T` del que cualquier otro tipo Java, i.e. clase Java, sea una instancia ¿Pero cuál? Tal como se ha planteado el problema la solución más obvia que se le puede dar, por poca experiencia de programación que se tenga en Java, **es definir `Object` como el tipo genérico `T` por excelencia**; de hecho la solución de implementar la Genericidad vía Herencia es la elegida por el propio lenguaje Java hasta su versión `1.4`. El sencillo ejemplo que se pasa a enunciar permite ilustrarla y permite señalar, a modo de preámbulo del siguiente apartado, los problemas de legibilidad y fiabilidad que tiene el Software que con ella se produce:

Sea la siguiente clase la que representa en Java un almacén de un `dato` de tipo `Integer`:

```
public class Caja {
  // una Caja TIENE UN dato de tipo Integer como atributo
  private Integer dato;
  /** crea una Caja vacía, i.e. que almacena un objeto null **/
  public Caja(){ super(); }
  /** devuelve el Integer almacenado en una Caja **/
  public Integer getDato(){ return this.dato; }
  /** substituye por d el Integer almacenado en una Caja **/
  public void setDato(Integer d){ this.dato = d; }
}
```

En base a la especificación de `Caja`, el siguiente programa `TestCaja` almacena en una `caja` vacía un objeto `Integer` con valor `46`, deposita el contenido de tal `caja` en una variable `Integer x` y, finalmente, muestra `x` por pantalla:

```
public class TestCaja {
  public static void main(String args[]){
    Caja caja = new Caja(); caja.setDato(new Integer(46));
    Integer x = caja.getDato(); System.out.println(x);
  }
}
```

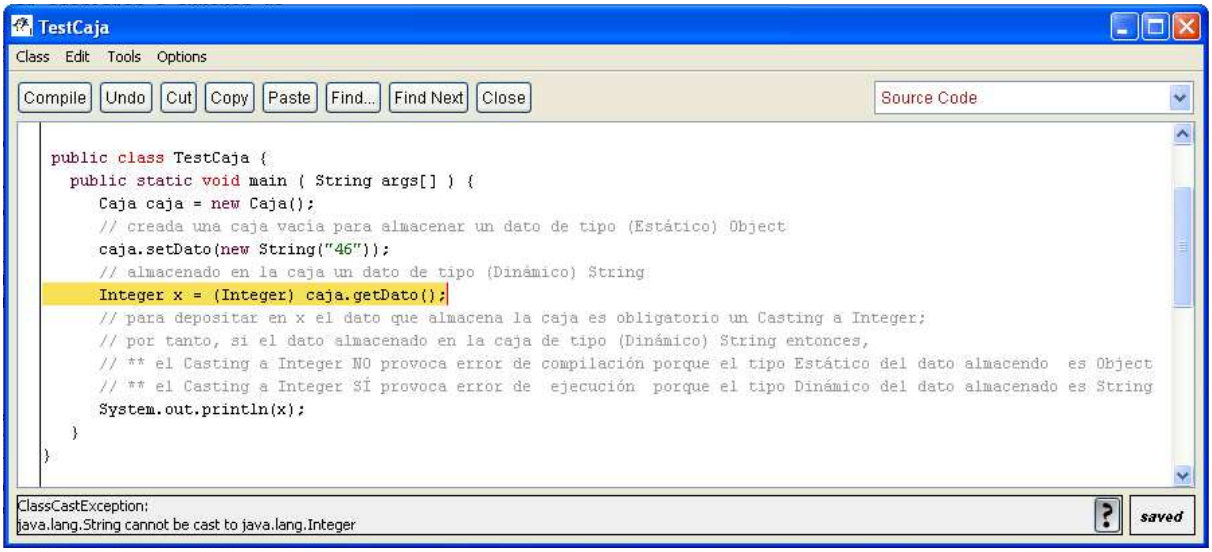
Pues bien, haciendo uso de la Herencia la clase `Caja` pasaría a ser genérica si el `dato` que almacena fuese de tipo `Object`, porque como ya es sabido cualquier tipo o clase Java es compatible con `Object`; la clase Java genérica que se tendría sería entonces la siguiente:

```
public class Caja {
   private Object dato;
   public Caja(){ super(); }
   public Object getDato(){ return dato; }
   public void setDato(Object d){ dato = d; }
}
```

Asimismo, teniendo en cuenta las reglas de compatibilidad de tipos que establece la Herencia, el siguiente programa reutiliza la clase `Caja` genérica instanciando su tipo `Object` a `Integer`:

```
public class TestCaja {
   public static void main(String args[]){
      Caja caja = new Caja(); caja.setDato(new Integer(46));
      Integer x = (Integer) caja.getDato(); System.out.println(x);
   }
}
```

Como se puede observar en el `main` de `TestCaja`, en una `Caja` de `Object` se puede almacenar un `Integer` porque un `Integer` ES UN `Object`; ahora bien, como un `Object` **NO** ES UN un `Integer`, es necesario un *Casting* a `Integer` para poder extraerlo de una `Caja` de `Object`. Nótese también que este obligado uso del *Casting* en la Genericidad vía Herencia puede conducir en ocasiones a errores de ejecución; así, si en el main de `TestCaja` se almacena el `String "46"` en lugar del `Integer 46`, entonces el posterior *Casting* a `Integer` que sufre dicho objeto provoca el error de ejecución `ClassCastException` que se observa en la siguiente figura:



2. Parametrización del Tipo de los Datos en Java

Si hasta la versión `1.4` de Java la Genericidad se conseguía implícitamente vía Herencia, definiendo `Object` como el Tipo genérico de clases y métodos, a partir de la versión `1.5` el lenguaje proporciona un mecanismo explícito que incrementa la fiabilidad y legibilidad del Software desarrollado: la Parametrización de clases y métodos mediante las denominadas variables de Tipo `T`. Su ámbito, declaración y uso en una clase se estudian en la primera sección de este apartado, que por tanto se dedica al diseño y uso de clases genéricas en Java `1.5` y posteriores; será en la segunda sección cuando todos los conceptos introducidos sobre clases genéricas se ilustren mediante dos ejemplos del estándar de Java: la clase genérica `ArrayList`, que representa en Java una Secuencia de Datos de Tipo genérico `T` mediante `array` redimensionable, y la interfaz genérica `Comparable`, la clase Raíz de Tipo genérico `T` que debe implementar en Java cualquier clase cuyos objetos sean comparables entre sí. Finalmente, la tercera sección del apartado se dedica al diseño y uso de métodos genéricos en Java `1.5` y posteriores y, por tanto, al estudio del ámbito, declaración y uso en un método de la variable de Tipo `T`.

2.1. Diseño y uso de una clase genérica

Desde la versión `1.5` de Java, la definición de una clase genérica se realiza incorporando a su cabecera de definición un nuevo parámetro formal `T` que representa el Tipo de sus Datos, la denominada variable o parámetro formal de Tipo de tal clase; en cuanto a la sintaxis a utilizar para ello, el ámbito y declaración de dicho parámetro queda perfectamente ilustrado por el diseño que se presenta a continuación, el que la clase `Caja` genérica definida en el apartado anterior debe tener a partir de Java `1.5`:

```
package ejemplos.tema3.lasCajas;
public class Caja<T> {
   private T dato;
   public Caja(){ super(); }
   public T getDato(){ return dato; }
   public void setDato(T d){ dato = d; }
}
```

Nótese en el ejemplo que el ámbito de aplicación de la variable de Tipo o parámetro `T` es toda la clase, por lo que se puede utilizar sin problema alguno para declarar sus atributos y en los perfiles y cuerpos de sus métodos. En cuanto al identificador `T` utilizado, se ha seguido el convenio habitual entre los programadores de Java para la nomenclatura de las variables de Tipo: una única letra mayúscula `T`, `E`, `V`, `U`, etc. Cabe advertir también, pues no es evidente en el ejemplo de `Caja`, que si una clase genérica tiene distintas variables de Tipo cada una de ellas se separa de la siguiente mediante una coma: `public class Clase<T, U> {...}`.

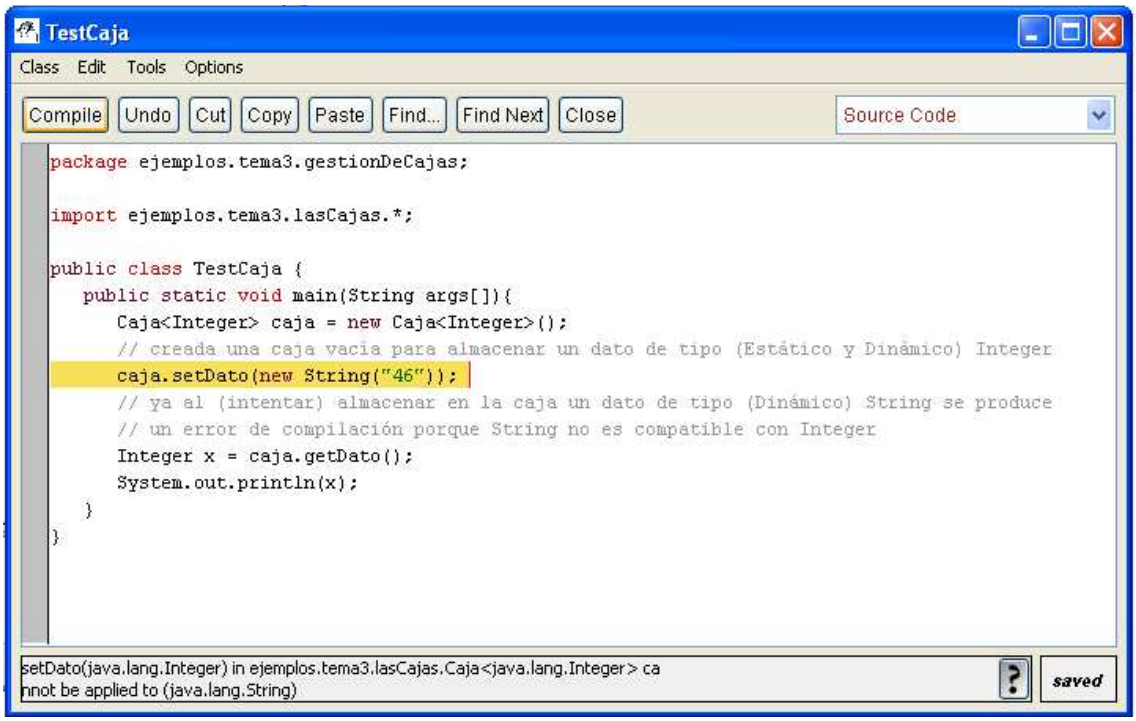
En lo referente a los pasos a dar para reutilizar una clase genérica de Tipo `T`, decir que en primer lugar se debe invocar el nombre de la clase *pero* instanciando su variable de Tipo `T` al nombre de un tipo o clase Java concreto, el único "valor"que puede tomar tal variable y lo que la diferencia cualquier otra variable Java. Así por ejemplo, se pueden declarar dos variables `cajaInteger` y `cajaString` instanciando la variable `T` de la clase `Caja` a `Integer` y `String` como sigue: `Caja< Integer > cajaInteger; Caja< String > cajaString;` sin embargo, nótese, no resultaría válida cualquier instanciación de una variable de Tipo al nombre de un tipo primitivo Java (`int`, `double`, `boolean`, etc.) porque no existe clase alguna para representarlo o, equivalentemente, es un "`unexpected type`"para el intérprete Java en tiempo de compilación. El ejemplo de declaración de `cajaInteger` y `cajaString` permite observar también que el uso de una clase genérica es análogo al de un método con parámetros *salvo* que los parámetros actuales que se le pasan al invocarla sólo pueden ser nombres de tipos o clases Java concretos.

Una vez declaradas, para que las variables `cajaInteger` y `cajaString` referencien un objeto `Caja<Integer>` y uno `Caja<String>` basta con emplear la siguiente instrucción: `cajaInteger = new Caja< Integer >(); cajaString = new Caja< String >();` nótese que, de nuevo, en ella se invoca al constructor de `Caja<T>` instanciando su variable de Tipo al nombre del correspondiente tipo Java.

Una vez que las variables `cajaInteger` y `cajaString` referencian a sus correspondientes objetos ya se les pueden aplicar los métodos definidos en `Caja<T>` **sin necesidad de *Casting* alguno**, pues el tipo de Datos `Integer` o `String` de la `Caja` a la que referencian es conocido en tiempo de compilación. El siguiente programa, una versión del que se presentó en el apartado anterior como ejemplo del uso de una clase genérica vía Herencia, lo ilustra perfectamente:

```
package ejemplos.tema3.gestionDeCajas;
public class TestCaja {
   public static void main(String args[]){
      Caja<Integer> caja = new Caja<Integer>();
      caja.setDato(new Integer(46));
      Integer x = caja.getDato(); // sin Casting
      System.out.println(x);
   }
}
```

Además, y como se observa en la siguiente figura, si ahora se intenta almacenar en `caja` un dato de tipo incompatible con `Integer`, `String` por ejemplo, se tendría un error de compilación en lugar del error de ejecución que se producía en el apartado anterior al hacer lo mismo; es ésta precisamente una de las ventajas del uso de variables o parámetros de Tipo: el intérprete Java realiza en tiempo de compilación las comprobaciones de tipo necesarias para evitar que puedan ocurrir errores inesperados en tiempo de ejecución.



Un último detalle importante que queda por señalar sobre la instanciación de una variable de Tipo de una clase es el siguiente: si «`Caja<B> cajaTipoB = new Caja<B>();`» entonces `cajaTipoB` puede almacenar no sólo un `dato` de tipo `B` sino también de cualquier otra Derivada de `B`; **sin embargo**, una `Caja<D>` **NO** ES UNA `Caja<B>` aunque sea cierto que `D` ES UNA `B`.

Cuestiones:

- Discútase si el siguiente segmento de código provocaría algún error de compilación; en caso afirmativo propónganse las modificaciones necesarias para evitarlo.

```
Caja<Number> cajaNumero = new Caja<Number>();
cajaNumero.setDato(new Integer(20));
Integer entero = cajaNumero.getDato();
```

- ¿Es `Caja<Integer>` una clase Derivada de `Caja<Number>`?

Finalmente, a modo de resumen, cabe recordar que las ideas más importantes presentadas en esta sección son las siguientes:

- La Genericidad se implementa en Java `1.5` y posteriores definiendo clases Parametrizadas; éstas no son realmente clases Java sino Esquemas de Clase (*Templates*) a partir de los cuales, vía instanciación, se obtienen las verdaderas clases.
- Para trabajar con tipos primitivos -`int`, `double`, etc.- en clases genéricas es imprescindible recurrir a sus correspondientes clases Envoltorio -`Integer`, `Double`, etc.
- La implementación de la Genericidad en Java `1.5` y posteriores permite escribir código más fácil de comprender y mantener, permite la detección de errores en tiempo de compilación, evita el uso de *Casting* explícitos y permite la comprobación de tipos en estructuras homogéneas.

2.2. Estudio de dos casos concretos del estándar de Java: `ArrayList` y `Comparable`

En la Jerarquía `Collection` del API de Java hay multitud de definiciones de clases genéricas; de todas ellas, a modo de ejemplo, en esta sección se estudian de manera sucinta dos: la clase genérica `ArrayList` y la interfaz genérica `Comparable`.

La clase genérica `ArrayList`

La clase genérica `ArrayList` de `java.util` es una de las implementaciones de la interfaz `List<E>`, la representación en el estándar de Java de un Grupo de `E`lementos de tipo `E` a los que se puede acceder por posición. En concreto, `ArrayList` implementa la interfaz `List` mediante un `array` redimensionable y, entre otros, su Especificación incluye los siguientes métodos:

```
// Construye una Lista vacía con CAPACIDAD_POR_DEFECTO 10
public ArrayList()

// Añade el Elemento e al final de una Lista: incrementa la talla de la Lista
// en uno y si es necesario incrementa su capacidad
public boolean add(E e)

// Devuelve el Elemento situado en la posición i de una Lista,
// o lanza IndexOutOfBoundsException si i está fuera de rango (i < 0 || i >= size())
public E get(int i)

// Devuelve la posición de la primera aparición de o en una Lista, aquel r para el que
// r.equals(o)==true; si el objeto o no está en la Lista devuelve -1 para advertirlo
public int indexOf(Object o)

// Borra la primera aparición del Elemento e de una Lista:
// si e está en la Lista desplaza los Elementos situados a continuación de él
// hacia la izquierda, resta uno a su talla y devuelve true; si el objeto o no
// está en la Lista no la modifica y devuelve false para advertirlo
public boolean remove(E e)

// Devuelve el número de Elementos de una Lista
public int size()
...
```

El siguiente programa muestra cómo se puede usar esta clase genérica del estándar de Java para representar y manipular un Grupo de Figuras `g`; notar que este programa sólo se diferencia del homónimo presentado en el tema anterior en que representa `g` mediante un `ArrayList<<Figura>` en vez de con un `ArrayGrupoDeFiguras` y, por tanto, permite observar también las diferencias que existen entre reutilizar una clase genérica y una que no lo es.

```
package ejemplos.tema3.gestionFigurasV4;
//Para poder usar ArrayList hay que importar el package donde se ubica
import java.util.*;
import ejemplos.tema2.lasFigurasV3.*; import entradaNoEstandar.LecturaValida;
public class TestExcepcionesGrupoDeFiguras {
  public static void main(String args[]){
    Scanner teclado = new Scanner(System.in).useLocale(new Locale("es", "US"));
    // Para representar un Grupo de Figuras mediante un ArrayList
    // se instancia su variable de tipo E al nombre del tipo Java Figura
    ArrayList<Figura> g = new ArrayList<Figura>();

    System.out.println("Introduzca la talla del Grupo");
    int numF = LecturaValida.leerInt(teclado, 0, Integer.MAX_VALUE);
    for ( int i = 0; i < numF; i++ ){
      Figura f = menuFigura(teclado); f.leer(teclado);
      boolean insertada = g.add(f);
      // Alternativamente: g.add(f);
    }
    System.out.println("Tras insertar "+g.size()+" Figuras, el Grupo es:");
    System.out.println(g.toString());

    System.out.println("Figura a buscar en el Grupo ... ");
    Figura aBuscar = menuFigura(teclado); aBuscar.leer(teclado);
    int posF = g.indexOf(aBuscar);
    System.out.println("Primera aparición de "+aBuscar+" en posición "+posF+" del Grupo");

    try{
      Figura recuperada = g.get(posF);
      System.out.println("la Figura en posición "+posF+" del Grupo es "+recuperada);
    }catch(IndexOutOfBoundsException eUnchecked){
      System.out.println("Al recuperar: no existe Figura en posición "+posF));
    }

    System.out.println("Figura a borrar del Grupo ..."+aBuscar);
    boolean borrada = g.remove(aBuscar);
    System.out.println("¿Borrada la primera aparición de "+aBuscar+" del Grupo? "+borrada);
    System.out.println("Por tanto, el Grupo actual es:"); System.out.println(g.toString());

    System.out.print("Área total del Grupo: ");
    // Obviamente, area() NO es un método de ArrayList; se debe implementar
    // por tanto mediante un Recorrido del ArrayList, en el que se usa el
    // método get para recuperar cada una de sus Figuras y obtener su área
    double area = 0.0;
    for ( int i = 0; i < g.size(); i++ ) area += g.get(i).area();
    // Alternativamente: for ( Figura i: g ) area += i.area();
    System.out.println(String.format(new Locale("US"), "%.2f", area));
    ...
  }
  private static Figura menuFigura(Scanner teclado){...}
}
```

Ejercicios propuestos:

1. Diséñese un programa Java que construya un `ArrayList` de `String` con los colores amarillo, azul, rojo y verde y que muestre el resultado de comprobar si un color dado está en él así como los colores que contiene.
2. Diséñese un programa Java que construya un `ArrayList` de `Caja<Integer>` con los 10 primeros impares y que muestre por pantalla el resultado de sumarlos.
3. Tras crear en `ejemplos.tema3` el paquete `lasFigurasV4` y copiar en él todas las clases de `ejemplos.tema2.lasFigurasV3`, diséñese la clase `ArrayListGrupoDeFiguras` que implementa la siguiente interfaz `GrupoDeFiguras`:

```
package ejemplos.tema3.lasFigurasV4;
import librerias.excepciones.*;
public interface GrupoDeFiguras{
    /** devuelve la talla de un Grupo de Figuras **/
    int talla();
    /** inserta la Figura f en un Grupo **/
    void insertar(Figura f);
    /** devuelve la Figura que ocupa la i-ésima posición del Grupo,
      * o lanza ElementoNoEncontrado para advertir que i no es una posición válida **/
    Figura recuperar(int i) throws ElementoNoEncontrado;
    /** devuelve la posición de la primera aparición de f en un Grupo de Figuras,
      * o devuelve -1 si f no está en el Grupo **/
    int indiceDe(Figura f);
    /** elimina la primera aparición de f en un Grupo de Figuras y devuelve true,
      * o devuelve false si f no está en el Grupo **/
    boolean eliminar(Figura f);
}
```

   Nota: modificar la clase `ArrayGrupoDeFiguras` para que implemente `GrupoDeFiguras`.

La interfaz genérica `Comparable`

La interfaz `Comparable` de `java.lang` describe el Modelo genérico y estándar de comparación en Java, i.e. define como sigue un (único) método de comparación de cualesquiera dos objetos de Tipo genérico `T` en el estándar de Java:

```
/** devuelve el valor int resultado de comparar dos Datos de tipo T, this y otro,
  * menor que 0 si this es menor que otro, mayor que 0 si this es mayor que otro
  * e igual a 0 si this es igual a otro **/
int compareTo(T otro);
```

Como se puede ver en la documentación del API, son muchas y bien conocidas las clases del estándar de Java que implementan esta interfaz: `Integer`, `Double`, `Date`, `String`, etc. El motivo por el que lo hacen es bastante obvio: si los objetos de una clase Java `C` (estándar o no) son susceptibles de ser comparados entre sí en base a un cierto criterio, por ejemplo para poder ordenar un Grupo de ellos o determinar su máximo o mínimo, entonces `C` debe definir un método de comparación de dos de sus objetos lo más reutilizable posible; como la clase `Object` **no** proporciona ninguno, la clase `C` debe sobrescribir obligatoriamente para especializarlo un método de comparación tan abstracto y estándar como el `compareTo` de `Comparable<T>`, lo mismo que suele hacer por motivos similares con los métodos `equals` y `toString` de `Object`.

Cuestión: ¿Por qué motivo la clase `Object` no define el método `compareTo`?

Así por ejemplo, si se quiere reutilizar un método genérico para ordenar por área un `GrupoDeFiguras` antes deberá definirse la clase `Figura` como una Implementación de `Comparable` que instancia su tipo genérico `T` a `Figura` y, por tanto, sobrescribe obligatoriamente su método `compareTo` para establecer el criterio de comparación por área de cualesquiera dos Figuras; traduciendo a Java se tiene entonces el siguiente diseño de la clase `Figura`:

```
package ejemplos.tema3.lasFigurasV4;
import java.util.*; import java.awt.geom.*; import librerias.util.entradaNoEstandar.*;
public abstract class Figura implements Comparable<Figura> {
   ...
   // Sobrescritura del método compareTo de Comparable: dos Figuras se comparan por área
   final public int compareTo(Figura f){
     double areaF = f.area(), areaThis = this.area();
     if ( areaThis < areaF ) return -1;
     if ( areaThis > areaF ) return  +1;
     return  0;
   // Alternativamente, todo el código anterior se puede substituir por una instrucción:
   // return (int) Math.signum(this.area() - f.area());
   }
   ...
}
```

Recordando ahora que la clase `Figura` ya sobrescribía el método `equals` de `Object` como sigue

```
/** comprueba si una Figura es igual a otra dada x,
  * i.e. comprueba si ambas tienen el mismo tipo, color y área **/
final public boolean equals(Object x){
   Figura fX = (Figura)x;
   return ( tipo.equals(fX.tipo) && color.equals(fX.color) && area() == fX.area() );
}
```

se puede plantear uno de los puntos más importantes que menciona la documentación del API de Java sobre el método `compareTo` de `Comparable`; se dice allí, literalmente,

> *It is strongly recommended, but not strictly required that* `(x.compareTo(y)==0)==(x.equals(y))`. *Generally speaking, any class that implements the* `Comparable` *interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals"*

Dicho en Castellano, se recomienda que los métodos `compareTo` y `equals` implementen una definición de igualdad consistente, i.e. que `x.compareTo(y)` valga `0` cuando `x.equals(y)` tome el valor `true` y viceversa. Es más, se aconseja también que cuando en el diseño de una clase no se siga esta recomendación su documentación incluya un mensaje como "Nota: esta clase tiene un orden natural que es inconsistente con `equals`". Nótese que éste sería el caso de la clase `Figura`, pues mientras que el criterio de ordenación "natural"de las Figuras de un Grupo es su área para `buscar` o `borrar` una `Figura fX` en él parece lógico exigir que contenga una con igual área, tipo y color que `fX`; observar también que si se quisiera seguir la recomendación de consistencia en Figura, entonces `compareTo` y `equals` se deberían diseñar en ella como sigue:

```
final public int compareTo(Figura f){
  double areaF = f.area(), areaThis = this.area();
  if ( areaThis < areaF ) return -1;
  if ( tipo.equals(f.tipo) && color.equals(f.color) && areaThis == areaF ) return  0;
  return  +1;
}
final public boolean equals(Object x){
  return ( this.compareTo(x) == 0 );
}
```

2.3. Diseño y uso de un método genérico

Al igual que sucedía en una clase Java, para que un método Java sea genérico debe incluir en su cabecera de definición una variable o parámetro formal de Tipo `U`; ahora bien, y a diferencia de lo que sucede para la variable de Tipo `T` de una clase, `U` debe declararse en la cabecera del método justo antes del tipo de su resultado y su ámbito queda restringido al perfil y cuerpo del método desde su punto de declaración. Así por ejemplo, la siguiente sería la definición de un método genérico que almacena un `dato` de Tipo `D` en cada una de las componentes de un `ArrayList a` de tipo `Caja<D>`:

```
public static <D> void llenarCajas(D dato, ArrayList<Caja<D>> a){
  for ( Caja<D> caja: a ) caja.setDato(dato);
}
```

Entonces, por ejemplo, reutilizando `llenarCajas` como se muestra a continuación se consigue almacenar el `Integer` de valor `10` en cada `Caja` de `Integer` de un `ArrayList` de talla `20`:

```
ArrayList<Caja<Integer>> cajas = new ArrayList<Caja<Integer>>(20);
llenarCajas(new Integer(10), cajas);
```

Nótese que en el código anterior se invoca al método `llenarCajas` sin explicitar el nombre del tipo concreto al que se instancia `D`; simplemente el intérprete Java lo infiere en tiempo de compilación a partir de los tipos de los parámetros actuales y formales del método: **si** el tipo del primer parámetro actual de `llenarCajas`, el que substituye al `dato` de Tipo `D`, es `Integer` **y si** el tipo del segundo parámetro actual de `llenarCajas`, el que substituye al `ArrayListCaja<D>` `a`, es `ArrayList<Caja<Integer>>` **entonces** el nombre del tipo al que se instancia `D` no puede ser otro más que `Integer`; de hecho, se denomina **Inferencia de Tipo** al mecanismo estático que usa el intérprete Java para inferir el nombre del tipo al que se instancia `D` y que le permite evaluar la invocación a un método genérico de la misma forma que si éste no lo fuera.

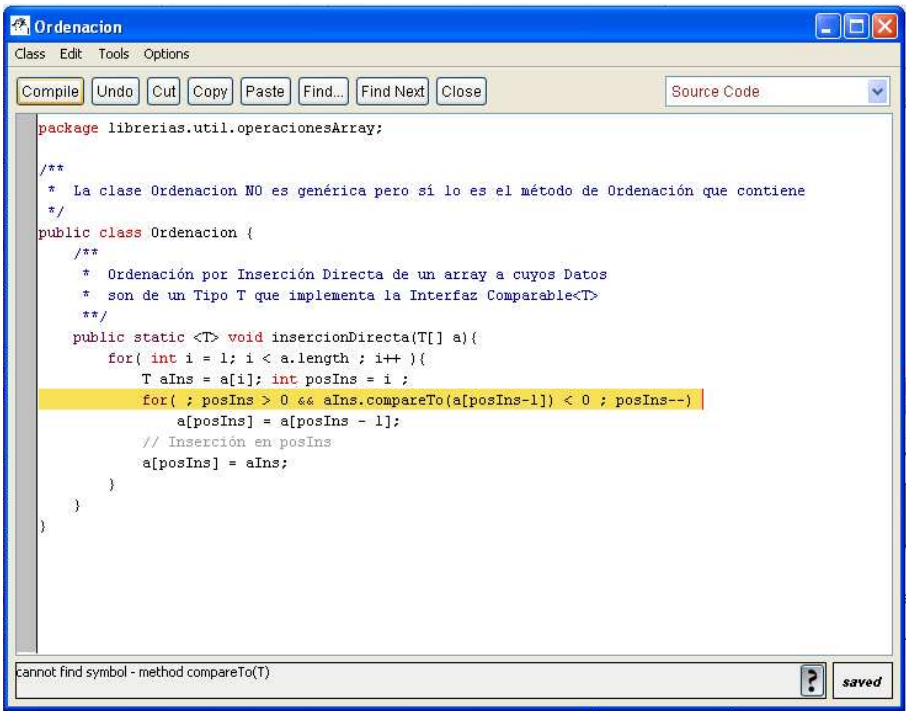
Cuestión: el método `llenarCajas` se podría seguir invocando igual que en el ejemplo anterior aún en el caso en el que la variable `cajas` se definiera como un `ArrayList` de `Caja<Number>` ¿Cuál sería en ese caso el tipo inferido por el intérprete Java? ¿Por qué?

3. Restricción de un Tipo genérico: declaración y uso de variables de Tipo Restringido en clases y métodos Java

Para introducir el papel que juega la Restricción de un Tipo genérico se plantea el siguiente caso: poniendo en práctica lo estudiado sobre Genericidad, se ha diseñado el método que figura a continuación para ordenar por Inserción las componentes de un `array a` de Tipo genérico `T`:

```
public static <T> void insercionDirecta(T a[]){
  for( int i = 1; i < a.length ; i++ ){
    T aIns = a[i]; int posIns = i;
    for (;posIns>0 && aIns.compareTo(a[posIns-1]) < 0;posIns--) a[posIns]=a[posIns-1];
    a[posIns] = aIns;
  }
}
```

Pero cuando se compila este método, definido en la clase `Ordenacion` del paquete `operacionesArray` de `librerias.util`, y como se observa en la siguiente imagen, surge un problema: el intérprete Java lanza un error con mensaje asociado «`cannot find symbol — method compareTo(T)`» . Pero ... ¿por qué?



Pensando un poco sobre lo sucedido no es difícil deducir el motivo de tal error: no todos los nombres de tipos concretos Java a los que se puede instanciar la variable de Tipo `T` son de la Jerarquía `Comparable` y, por tanto, no todos tienen definido el método `compareTo`. En conclusión, el método genérico `insercionDirecta` diseñado sólo se puede aplicar al subconjunto de clases o tipos Java `Comparable` o, equivalentemente, es un método cuyo tipo `T` está Restringido o Acotado superiormente por `Comparable`.

Para expresar esta restricción del tipo genérico `T` de un método el lenguaje Java proporciona la siguiente sintaxis: situar tras el identificador de la variable de Tipo la palabra reservada `extends` seguida por el nombre de la clase Java que la acota superiormente; por tanto, para expresar que el método `insercionDirecta` sólo es aplicable a objetos de tipo `Comparable` su variable de Tipo debe ser `<T extends Comparable<T>>`:

```
package librerias.util.operacionesArray;
public class Ordenacion {
  public static <T extends Comparable<T>> void insercionDirecta(T a[]){...}
}
```

Nótese que si se quiere usar este método para, por ejemplo, ordenar un `ArrayGrupoDeFiguras` entonces basta modificar el código de la clase como se muestra a continuación:

```
package ejemplos.tema3.lasFigurasV4;
import librerias.util.operacionesArray.*; import librerias.excepciones.*;
public class ArrayGrupoDeFiguras implements GrupoDeFiguras {
  ...
  public Figura[] ordenar(){
      Figura res[] = this.toArray();
      Ordenacion.insercionDirecta(res);
     return res;
  }
  ...
}
```

La sintaxis de declaración de una variable de Tipo Restringido **sirve igualmente para una clase genérica Java**; así por ejemplo, para diseñar una clase genérica que represente una Caja en la que únicamente se pueden almacenar Números, y no cualquier tipo de Datos como ocurre en `Caja<T>`, su variable de Tipo debe ser `<T extends Number>`:

```
package ejemplos.tema3.lasCajas;
public class CajaNumber<T extends Number> {
   private T dato;
   public CajaNumber(){ super(); }
   public T getDato(){ return dato; }
   public void setDato(T d){ dato = d; }
   public int getParteEntera(){ return dato.intValue(); }
}
```

Cuestiones y Ejercicios propuestos:

1. ¿Qué sucede al compilar la clase `CajaNumber` si, por error, se omite la restricción de su variable de Tipo? ¿Por qué?
2. ¿Sería correcto el uso que se realiza de la clase `CajaNumber` en la siguiente instrucción: «`CajaNumber<String> cajaString=new Caja<String>()`»? ¿Por qué?
3. En el paquete `lasCajas` de `ejemplos.tema3`, diséñese una clase `CajaConPeso` para representar una Caja que almacena un `dato` de Tipo genérico `T` y el valor `double` que indica su `peso`; sus métodos deben permitir, además de modificar y consultar el `dato` y `peso` de una Caja, la comparación por peso de dos Cajas cualesquiera. Hecho esto, en el paquete `gestionDeCajas` de `ejemplos.tema3`, diséñese un programa `TestCajaConPeso` que cree un `array` de 10 Cajas de Animales con pesos aleatorios y lo ordene utilizando el método genérico `insercionDirecta` de la clase `Ordenación`.
4. En el paquete `operacionesArray` de `librerias.util`, diséñese una clase `Operaciones` con la siguiente Especificación:

librerias.util.operacionesArray

Class Operaciones

```
java.lang.Object
  └librerias.util.operacionesArray.Operaciones
```

```
public class Operaciones
extends java.lang.Object
```

Constructor Summary
Operaciones()

Method Summary	
static <T extends java.lang.Comparable<T>> boolean	estaOrdenado(T[] unArray, boolean ascendente) SII unArray.length > 1: comprueba si unArray de Datos de Tipo T compatible con Comparable está o no ordenado ascendente o descendentemente, según el valor del parámetro ascendente sea true o false
static <T> int	indiceDe(T[] unArray, T aBuscar) devuelve la posición de la primera aparición de aBuscar en unArray de Datos de tipo T, o devuelve -1 si aBuscar no está en unArray
static <T extends java.lang.Comparable<T>> T	maximo(T[] unArray) SII unArray.length > 0: obtiene el máximo de unArray de Datos de tipo T compatible con Comparable
static <T extends java.lang.Comparable<T>> T	minimo(T[] unArray) SII unArray.length > 0: obtiene el mínimo de unArray de Datos de tipo T compatible con Comparable
static <T extends java.lang.Number> int[]	toArrayInt(T[] unArray) obtiene un array de int con las unArray.length partes Enteras de unArray de Datos de Tipo T compatible con Number
static <T> java.lang.String	toString(T[] unArray) obtiene un String que representa a unArray de Datos de tipo T

Diséñese también en el mismo paquete un programa que use los métodos de la clase `Operaciones` para manipular un `array` de `Integer`.

5. Indíquese si sería posible implementar el método `indiceDe` de la clase `Operaciones` invocando en su código a `compareTo` en lugar de a `equals`; en caso de que sí lo fuera, ¿sería correcto?
6. Diséñese en la clase `Ordenacion` un método `seleccionDirecta` que ordene un `array` genérico por Selección.
7. Indíquese por qué el diseño de la siguiente clase genérica no es correcto.

```
public class Caja<String> {
   private String dato;
   public Caja(){ super(); }
   public String getDato(){ return dato; }
   public void setDato(String d){ dato = "Hola"; }
}
```

8. Explíquese por qué el diseño de la siguiente clase genérica no es correcto, a pesar de que al compilar y ejecutar el programa `TestCaja` no se observe problema alguno.

```
public class Caja<Number> {
    private Number dato;
    public Caja(){ super(); }
    public Number getDato(){ return dato; }
    public void setDato(Number d){ dato = d; }
}
public class TestCaja {
   public static void main(String args[]){
     Caja<Integer>caja = new Caja<Integer>();
     caja.setDato(new Integer(46));
     Integer x = caja.getDato(); System.out.println("la Caja contiene un: "+x);
   }
}
```