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

Tema 1. Herencia y Polimorfismo

Mabel Galiano Natividad Prieto

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

Índice

Objetivos y Bibliografía

El principal objetivo de este primer tema es que el alumno aprenda cuándo y cómo utilizar en Java la **Herencia**, el mecanismo más importante de **Re-Utilización del Software** que proporciona cualquier lenguaje de Programación Orientada a objetos (POO). Para situarlo convenientemente entre los otros mecanismos, más básicos, estudiados en la asignatura Programación de primer curso, i.e. el `package` y la declaración de atributos para la Composición de clases, en el apartado de introducción de este tema se revisa el importante papel que juega cualquier mecanismo de Relación entre clases a la hora de desarrollar una aplicación bajo el

paradigma POO, esto es la ayuda que supone a la hora de reutilizar Software cuando éste ha sido diseñado manteniendo el principio de Ocultación de la Información; ello permitirá además repasar conceptos clave del diseño de clases en Java y afianzar los principios ya conocidos de la POO y su soporte Java.

Tras la introducción, los apartados 2 y 3 del tema se dedican a presentar con detalle la Herencia como expresión Java del modelo de relación entre clases **ES UN(A)** o modelo **Jerárquico** y, por tanto, a introducir paulatinamente muchos de los recursos Java necesarios para afrontar con garantías el desarrollo tanto de las Estructuras de Datos que estudia la asignatura como el de las aplicaciones con las que se ilustra su uso; se realizan en concreto los avances en la descripción del lenguaje Java que requieren **el uso de una Jerarquía** -tipos Estático y Dinámico de una variable, mecanismos de comprobación y conversión de tipo y Enlace Dinámico-, **el diseño de una Jerarquía** -cláusula `extends`, referencia `super` y Sobrescritura- y **el diseño y uso de clases** `final`, `abstract` e `interface`.

Como bibliografía básica del tema se recomienda consultar los siguientes capítulos del libro de Weiss M.A. **Estructuras de datos en Java** (Addison-Wesley, 2000): el `2` y el `3` para el apartado de introducción del tema y el capítulo `4` hasta su sección `3` para los dos restantes dedicados a la Herencia.

NOTA. Las clases Java que aparecen en este tema se han ubicado en el paquete `tema1`, incluido a su vez en el paquete `ejemplos` del proyecto BlueJ `eda`.

1. Relaciones entre Clases: Re-Utilización del Software en Java

Bajo un paradigma de Programación Orientada a objetos (POO), la **Re-Utilización del Software** es el principal criterio de diseño de aplicaciones, desde las más sencillas a las más complejas: en lugar de implementar desde cero cada nueva aplicación, utilícese siempre que sea posible el código estándar existente porque ya ha sido diseñado y depurado, es eficiente y tiene mantenimiento. Para lograr satisfacer efectivamente este criterio se deben cumplir al menos dos requisitos. En primer lugar, el lenguaje POO elegido debe ofrecer al programador diversos mecanismos que, usados como las dos caras de una misma moneda, permitan tanto el uso del Software existente como la construcción de nuevo código estándar que, a su vez, podrá ser reutilizado por otras aplicaciones; en segundo lugar, el programador debe comprometerse a desarrollar cada nueva aplicación única y exclusivamente aplicando disciplinadamente los mecanismos de Re-Utilización de Software que le ofrece el lenguaje.

Bajo estos presupuestos, y recordando que en Java cualquier cosa es un objeto o instancia de una clase, para desarrollar una aplicación en este lenguaje el primer paso es determinar las clases o tipos de los objetos que en ella intervienen y las relaciones que guardan entre sí. Tras este proceso, y en vista del organigrama de clases que origine, el diseño de la aplicación continúa según se indica a continuación:

- **si alguna Clase de la Aplicación CA es exactamente igual a otra Clase ya existente C** la reutilización es simple y se ha estudiado ya en la asignatura Programación: suponiendo que C está ubicada en su correspondiente `package`, usando la directiva de acceso `import` se obtiene el acceso a dicha clase y, con él, la posibilidad de usar (crear con `new` y manipular vía operador `.`) sus objetos mediante variables Referencia. Más en concreto, si entre las clases CA y C se da una relación de Composición TIENE UN(A), i.e. **CA TIENE UNA C**, el mecanismo de Re-Utilización de C que emplea Java para modelarla es el de definición de atributos de una clase: CA no sólo usa un objeto de la clase C sino que también lo define como uno de sus atributos.

  En cualquier caso **es la Especificación de la clase C la que dicta qué se puede hacer con sus objetos o cómo usarlos**, ignorando en todo momento los detalles del código específico que la implementa, i.e. su Implementación;
- **si alguna Clase de la Aplicación CA no es exactamente igual a otra Clase ya existente C pero sí muy parecida**, como se detallará en el apartado sobre la Herencia en Java, la reutilización es más compleja pero aún posible si existe entre las clases una relación Jerárquica **ES UN(A)**, específicamente si **CA ES UNA C**;
- **si alguna Clase de la Aplicación CA es completamente diferente a las ya existentes** se debe proceder a diseñarla reutilizando siempre que sea posible objetos de clases ya existentes, bien usándolos simplemente o componiéndolos con la nueva clase o heredándolos. Además se deben aplicar todos los mecanismos de reutilización pertinentes para que CA pueda formar parte del estándar -como mínimo su inclusión en el `package` correspondiente- y, por tanto, pueda ser reutilizada por otras aplicaciones.

Para concretar algunos aspectos de la metodología que se acaba de exponer se desarrollan a continuación dos aplicaciones de manipulación de Figuras usando los mecanismos Java de Re-Utilización del Software estudiados en Programación de primer curso; el desarrollo de la

segunda aplicación, la más compleja, dará pie a introducir la Herencia como el mecanismo de Re-Utilización de Software que modela una relación Jerárquica ES UN(A) entre dos clases, una de la aplicación a desarrollar y otra ya existente.

Ejemplo 1: la clase `TestCirculo`

Diséñese una aplicación para manipular Círculos en base a la Especificación de la clase `Circulo` que figura a continuación:

ejemplos.tema1.lasFigurasV0

Class Circulo

```
java.lang.Object
  └ejemplos.tema1.lasFigurasV0.Circulo
```

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

Constructor Summary

`Circulo()`	crea un Círculo estándar de radio 3.0, color rojo y centro en el origen
`Circulo(double radio, java.lang.String color, java.awt.geom.Point2D.Double centro)`	crea un Círculo de radio r, color c y centro en el Punto p

Method Summary

double	`area()` calcula el área de un Círculo
boolean	`equals(java.lang.Object x)` indica si un Círculo es igual a x, i.e. si tienen el mismo radio y color
java.awt.geom.Point2D.Double	`getCentro()` consulta el centro de un Círculo
java.lang.String	`getColor()` consulta el color de un Círculo
double	`getRadio()` consulta el radio de un Círculo
static Circulo	`leer(java.util.Scanner teclado)` obtiene un Circulo leyendo de teclado los valores de sus componentes
double	`perimetro()` calcula el perímetro de un Círculo
void	`setCentro(java.awt.geom.Point2D.Double nuevoCentro)` actualiza el centro de un Círculo a nuevoCentro
void	`setColor(java.lang.String nuevoColor)` actualiza el color de un Círculo a nuevoColor
void	`setRadio(double nuevoRadio)` actualiza el radio de un Círculo a nuevoRadio
java.lang.String	`toString()` obtiene el String que representa a un Círculo

Methods inherited from class java.lang.Object

`clone, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

Como se puede deducir fácilmente del enunciado, las clases principales de la aplicación a desarrollar son dos: `Circulo` y el programa Java `TestCirculo` que lanza la aplicación al ejecutar

el único método público que contiene, i.e. su `main`; nótese también que dicho `main` debe estar compuesto por instrucciones en las que se creen y manipulen, o se usen, objetos de tipo `Circulo`. Así por ejemplo, el siguiente podría ser el diseño del programa `TestCirculo`:

```
package ejemplos.tema1.gestionFigurasV0;
import ejemplos.tema1.lasFigurasV0.*;   // para usar la clase Circulo
import java.util.*;                     // para usar la clase Scanner
import java.awt.geom.*;                 // para usar la clase Point2D.Double
public class TestCirculo{
 public static void main(String args[]){
   Circulo c1 = new Circulo(); System.out.println(c1.toString());
   Circulo c2 = new Circulo(4.0, "Rojo", new Point2D.Double(0.0,3.0));
   System.out.println(c2.toString());
   Scanner teclado = new Scanner(System.in).useLocale(new Locale("es", "US"));
   Circulo c3 = Circulo.leer(teclado); System.out.println(c3.toString());
   if ( c3.equals(c2) ) c3 = null; else c2 = c3;
   double radioC1 = c1.getRadio(); if ( radioC1 > 3.0 ) c1.setRadio(3.0);
   System.out.println("Área de c1 = "+String.format(new Locale("US"), "%.2f", c1.area()));
   if ( c1.equals(new Circulo()) ) c2 = c1;
 }
}
```

Como denotan sus líneas de `import` y su `main`, `TestCirculo` usa a su vez dos clases del estándar de Java que no están incluidas en `java.lang`: `Scanner` del paquete estándar `java.util` para poder construir un Círculo tras `leer` sus componentes desde teclado y `Point2D.Double` del paquete estándar `java.awt.geom` para definir el centro del `Circulo c2`. Pero de este diseño se pueden explicitar aún otros muchos detalles estudiados en la asignatura Programación sobre diseño de clases en Java y manipulación de sus objetos mediante variables Referencia; para concluir este ejemplo repasándolos se propone la realización de los siguientes ejercicios.

Ejercicios propuestos:

1. En el desarrollo de cualquier aplicación siempre interviene un mismo tipo de clase Java ¿De qué tipo de clase se está hablando y cuál es la característica principal que la diferencia del resto de clases que componen una aplicación?
2. Explíquese brevemente el papel de la directiva `import` de Java; indíquese después que ocurriría si la clase `TestCirculo` careciera de las líneas en las que aparece.
3. Las variables `c1`, `c2` y `c3` del método `main` de `TestCirculo` son Referencias a objetos de la clase `Circulo`; trácese a mano la ejecución del método `main` para mostrar su estado y el de los objetos a los que apuntan antes y después de cada instrucción.
4. Respóndase a las siguientes cuestiones sobre el `main` de `TestCirculo`:
   - ¿Por qué la aplicación del método `leer` es diferente a la de los demás métodos de la clase `Circulo`?
   - ¿Sería correcta en él la instrucción «`double radioC1 = c1.radio;`»? ¿Por qué?
   - ¿Por qué sí es correcta en él la instrucción «`System.out.println(c2);`»? ¿Por qué no lo es «`if (c1 == new Circulo()) c2 = c1;`»?
5. Para diseñar la clase `Circulo` que usa el programa `TestCirculo` se han seguido ciertos criterios básicos de Re-Utilización de Software y de Ocultación de la Información; señálense las palabras reservadas del lenguaje Java que los expresan en el código fuente de la clase que figura a continuación.

```
package ejemplos.tema1.lasFigurasV0;
import java.util.*; import java.awt.geom.*;
public class Circulo{
  private double radio; private String color; private Point2D.Double centro;
  private static final double RADIO_POR_DEFECTO = 3.0;
  private static final String COLOR_POR_DEFECTO = "rojo";
  private static final Point2D.Double CENTRO_POR_DEFECTO = new Point2D.Double();
  /** crea un Círculo de radio r, color c y centro en el Punto p*/
  public Circulo(double radio, String color,  Point2D.Double centro){
    this.radio = radio; this.color = color; this.centro = centro;
  }
  /** crea un Círculo estándar de radio 3.0, color rojo y centro en el origen*/
  public Circulo(){ this(RADIO_POR_DEFECTO, COLOR_POR_DEFECTO, CENTRO_POR_DEFECTO);}
  /** consulta el radio de un Círculo*/
  public double getRadio(){return this.radio;}
  /** consulta el color de un Círculo*/
  public String getColor(){return this.color;}
  /** consulta el Centro de un Círculo*/
  public Point2D.Double getCentro(){return this.centro;}
  /** actualiza el radio de un Círculo a nuevoRadio*/
  public void setRadio(double nuevoRadio){ this.radio = nuevoRadio;}
  /** actualiza el color de un Círculo a nuevoColor*/
  public void setColor(String nuevoColor){ this.color = nuevoColor;}
  /** actualiza el Centro de un Círculo a nuevoCentro*/
  public void setCentro(Point2D.Double nuevoCentro){this.centro = nuevoCentro;}
  /** calcula el área de un Círculo*/
  public double area(){ return Math.PI * radio * radio;}
  /** calcula el perímetro de un Círculo*/
  public double perimetro(){return 2 * Math.PI * radio;}
  /** obtiene el String que representa a un Círculo*/
  public String toString(){
    String res = "Círculo de radio "+String.format(new Locale("US"),"%.2f", this.radio);
    return res+", color "+color+" y centro "+centro.toString();
  }
  /** indica si un Círculo es igual a x, i.e. si tiene el mismo radio y color que x*/
  public boolean equals(Object x){
    return (this.radio == ((Circulo)x).radio && this.color.equals(((Circulo)x).color));
  }
  /** obtiene un Circulo leyendo de teclado los valores de sus componentes*/
  public static Circulo leer(Scanner teclado){
    Circulo res = new Circulo();
    System.out.println("********Introduzca las componentes del Círculo*********\n");
    System.out.println("¿radio?"); res.radio = teclado.nextDouble();
    System.out.println("¿color?"); res.color = teclado.next();
    System.out.println("¿coordenada x del centro?"); double x = teclado.nextDouble();
    System.out.println("¿coordenada y del centro?"); double y = teclado.nextDouble();
    res.centro = new Point2D.Double(x,y);
    return res;
  }
}
```

6. Modificando donde convenga la Implementación de `Circulo`, y en su mismo paquete, diséñese la clase `Rectangulo`; hecho esto, y teniendo en cuenta que un Cuadrado es un Rectángulo con la misma base que altura, diséñese la clase `Cuadrado` en el mismo paquete.
7. Indíquese cómo se debe modificar la actual `TestCirculo` para obtener una clase `TestFigura` que manipule cualquier tipo de Figura existente en `ejemplos.tema1.lasFigurasV0`.

Ejemplo 2: la clase `ArrayGrupoDeCirculos`

Un programa como `TestCirculo` no es el más adecuado para trabajar con un Grupo de Círculos de talla variable; lo más lógico es disponer de uno que permita realizar las siguientes operaciones sobre un Grupo de Círculos:

> obtener uno nuevo Grupo leyendo de teclado, una a una, todas sus componentes; insertar en un Grupo un nuevo Círculo; obtener el `toString()` de un Grupo; obtener el `i`-ésimo Círculo de un Grupo; comprobar si un Círculo dado `c` está en un Grupo; eliminar de un Grupo un Círculo dado `c`; calcular el área total de un Grupo, i.e. la suma de las áreas de todos los Círculos que lo componen; ordenar ascendentemente (por área) un Grupo; depositar los Círculos de un Grupo en un `array` (`toArray`).

De la lectura cuidadosa de este enunciado se desprende que los elementos básicos de la aplicación son los Círculos que componen un cierto Grupo, precisamente el que se usa en el `main` del programa que lanza la aplicación. Por tanto, son tres las clases principales que forman parte de la aplicación: `Circulo`, que representa el tipo de Figuras que componen un Grupo, `ArrayGrupoDeCirculos`, que como indica su nombre representa un Grupo de Círculos mediante un `array` de tipo base `Circulo`, y el programa `TestArrayGrupoDeCirculos` que la lanza la aplicación, i.e. cuyo `main` usa un `ArrayGrupoDeCirculos`; recordando los tipos de relación que se pueden dar entre las clases de un mismo `package`, `ArrayGrupoDeCirculos` se ubicará en el mismo paquete que `Circulo` y `TestArrayGrupoDeCirculos` en el mismo que `TestCirculo`.

Efectuada la descomposición de la aplicación en sus clases, los siguientes pasos son:

1. Determinar la Especificación de la clase `ArrayGrupoDeCirculos`; atendiendo al enunciado de la aplicación no es difícil obtener la siguiente:

ejemplos.tema1.lasFigurasV0

Class ArrayGrupoDeCirculos

```
java.lang.Object
  └ejemplos.tema1.lasFigurasV0.ArrayGrupoDeCirculos
```

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

Constructor Summary

`ArrayGrupoDeCirculos()`	construye un Grupo de Círculos vacío

Method Summary

double	`area()` obtiene el área total de un Grupo de Círculos sumando las áreas de las talla() Figuras que lo componen
boolean	`eliminar(ejemplos.tema1.lasFigurasV0.Circulo c)` elimina la primera aparición de c en un Grupo de Círculos y devuelve true, o devuelve false si c no está en el Grupo
int	`indiceDe(ejemplos.tema1.lasFigurasV0.Circulo c)` devuelve la posición de la primera aparición de c en un Grupo de Círculos, o devuelve -1 si c no está en el Grupo
void	`insertar(ejemplos.tema1.lasFigurasV0.Circulo c)` inserta el Círculo c en un Grupo
ejemplos.tema1.lasFigurasV0.Circulo[]	`ordenar()` devuelve un array con los Círculos de un Grupo ordenados ascendentemente por área
ejemplos.tema1.lasFigurasV0.Circulo	`recuperar(int i)` devuelve el Círculo que ocupa la i-ésima posición del Grupo, o devuelve null si i no es válida, i<0 ó i>=talla()
int	`talla()` obtiene el número de Círculos o talla de un Grupo
ejemplos.tema1.lasFigurasV0.Circulo[]	`toArray()` obtiene un array cuyas componentes son los talla() Círculos de un Grupo
java.lang.String	`toString()` obtiene un String con los Círculos de un Grupo en orden de inserción

2. Diseñar `TestArrayGrupoDeCirculos` en base a la Especificación de `ArrayGrupoDeCirculos` establecida, por ejemplo como sigue:

```
package  ejemplos.tema1.gestionFigurasV0;
import ejemplos.tema1.lasFigurasV0.*; import java.util.*; import java.text.*;
public class TestArrayGrupoDeCirculos {
 public static void main(String args[]){
  Scanner teclado = new Scanner(System.in).useLocale(new Locale("es", "US"));
  // se insertan en un Grupo vacío talla Círculos que se leen de teclado
  System.out.println("Introduzca el número de Círculos a leer: ");
  ArrayGrupoDeCirculos g = new ArrayGrupoDeCirculos();
  int talla = teclado.nextInt();
  for ( int i = 0 ; i < talla ; i++ ){
    Circulo instancia = Circulo.leer(teclado); g.insertar(instancia);
   }
   System.out.print("El Grupo de Círculos actual es: \n"+g.toString());
   System.out.println("Buscar en el Grupo un Círculo");
   Circulo aBuscar = Circulo.leer(teclado);
   int pos1era = g.indiceDe(aBuscar);
   if ( pos1era != -1 )
    System.out.println(aBuscar+" está en la posición "+pos1era+" del Grupo");
   else System.out.println("El Círculo "+aBuscar+" no está en el Grupo");
   System.out.println("Borrar del Grupo un Círculo");
   Circulo aBorrar = Circulo.leer(teclado);
   boolean esta = g.eliminar(aBorrar);
   if ( esta ) System.out.println("Borrado del Grupo "+aBorrar);
   else System.out.println(aBorrar+" no está en el Grupo");
   System.out.println("Área del Grupo: "+String.format(new Locale("US"),"%.2f",g.area()));
   Circulo enOrden[] = g.ordenar();
   System.out.println("Círculos ordenados por área:\n"+Arrays.toString(enOrden));
 }
}
```

3. Diseñar la clase `ArrayGrupoDeCirculos` de forma que satisfaga la Especificación definida para ella en el primer paso, lo que se deja como ejercicio propuesto.

Tras concluir esta segunda aplicación cabe plantearse al menos las siguientes **cuestiones**:

- ¿Qué modificaciones debe sufrir el diseño de `ArrayGrupoDeCirculos` (su Especificación e Implementación) para obtener el de la clase `ArrayGrupoDeRectangulos`? ¿ Y para obtener el diseño de `ArrayGrupoDeCuadrados` a partir del de `ArrayGrupoDeRectangulos`?
- ¿Qué modificaciones tendrían que sufrir la clase `TestArrayGrupoDeCirculos` para obtener la clase `TestArrayGrupoDeFiguras`?
- ¿Es posible diseñar una única clase `ArrayGrupoDeFiguras` que represente un Grupo heterogéneo de Figuras, esto es un Grupo con Círculos, Rectángulos y Cuadrados? Si la respuesta es no, plantéense las ventajas que supondría diseñar una única clase `Figura` de la que serían instancias los objetos de las clases `Circulo`, `Rectangulo` y `Cuadrado` a la hora de simplificar el diseño actual de `TestArrayGrupoDeFiguras`?

Optimización de aplicaciones mediante el uso de una Jerarquía de clases

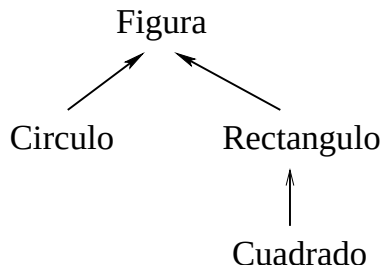
El principal inconveniente del proceso seguido para desarrollar las aplicaciones propuestas es que, para desgracia del programador y aunque se parecen mucho entre sí, cada una de las clases que representa un tipo específico de Figura ocupa un fichero `.java` distinto y se ha diseñado casi siempre mediante el socorrido `copy-paste` de otra; como consecuencia directa de ello, además, los programas de test que las usan las manipulan por separado mediante bloques de instrucciones idénticos salvo por el tipo de sus variables. Para subsanar este inconveniente la solución sería disponer de una **clase "Factor-Común"** `Figura` de `Circulo`, `Rectángulo` y `Cuadrado`. Concretando un poco más, `Figura` contendría todos los atributos y métodos que aparecen repetidos en `Circulo`, `Rectangulo` y `Cuadrado` y, lo que es más importante aún, los compartiría con todas ellas; así por ejemplo a un objeto de tipo `Circulo` se le podrían aplicar vía operador `.` no sólo los métodos propios de su clase sino también todos los de `Figura`.

Cuestión: en base a los diseños actuales de `Circulo`, `Cuadrado` y `Rectangulo`, obténgase la Especificación de la clase "Factor-Común" `Figura`. Hecho esto, ¿qué cambios se deberían realizar en las Especificaciones de `Circulo`, `Cuadrado` y `Rectangulo` para adaptarlos a la nueva situación?

Obsérvese también que utilizando variables de tipo `Figura` se podrían restructurar los dos paquetes de `ejemplos.tema1` como sigue:

- en `lasFigurasV0` se tendría una única clase `ArrayGrupoDeFiguras` para representar mediante un `array` de tipo `Figura` un Grupo heterogéneo de Figuras;
- en `gestionFigurasV0` se tendría una única clase de `TestArrayGrupoDeFiguras` para manipular cualquier Figura o Grupo de ellas.

Aunque las ventajas que esta clase `Figura` puede comportar son innegables, la pregunta obvia es cómo obtenerla. Para responderla basta observar que, en realidad, la relación que existe entre la clase Factor-Común `Figura` y las clases de las que es Factor Común, `Circulo` o `Rectangulo` o `Cuadrado`, es una relación Jerárquica **ES UN(A)**: un Círculo ES UNA Figura, un Cuadrado ES UNA Figura y un Rectángulo ES UNA Figura; pero además, como también un Cuadrado ES UN Rectángulo con la misma base que altura, por transitividad, un Cuadrado ES UN Rectángulo. La expresión gráfica de la Jerarquía de Figuras establecida sería la siguiente:



Para poder trascribirla a Java y, en base a ello, restructurar convenientemente cada uno de los paquetes que contiene `ejemplos.tema1` es necesario introducir y aprender a utilizar el mecanismo que implementa en Java este modelo de relación ES UN(A); a ello se dedica el siguiente apartado de este tema.

2. La Herencia como soporte Java del modelo ES UN(A)

La **Herencia** es el mecanismo Java que modela la relación entre clases **ES UN(A)** y, por tanto, la formación de **Jerarquías** de clases; como ya se ha mencionado, ello la convierte en el mecanismo de Re-Utilización de Software a emplear cuando alguna clase `D` de una aplicación **no** es exactamente igual a otra clase `B` ya existente **pero sí** guarda una relación ES UN(A) con ella. Específicamente, la notación que se suele emplear con las clases relacionadas por la Herencia es la siguiente: si una clase `D` ES UNA `B`, se dice que la clase Derivada `D` es una variación de la clase Base `B`; de este modo, `D` y `B` forman una Jerarquía `D` $\cdots\triangleright$ `B` en la que la clase `D` es una **sub**clase de `B` y `B` es una **super**clase de `D`. Así por ejemplo, `Circulo` y `Rectangulo` serían dos Derivadas de la clase Base `Figura`, o dos subclases de `Figura`; en cualquier caso las tres formarían parte de la Jerarquía `Figura`. Como además la relación ES UN(A) es transitiva -si `X` ES UNA `Y` e `Y` ES UNA `Z` **entonces** `X` ES UNA `Z`- la clase `Cuadrado` sería un subclase o Derivada tanto de `Figura` como de `Rectangulo` y, por tanto, también formaría parte de la Jerarquía `Figura`.

En cuanto a las características más relevantes de la relación entre las clases Base y Derivada, aquellas con las que la Herencia dota a las clases de una Jerarquía, son las siguientes:

1. la Herencia se basa en `B`, la clase Base y Raíz de la Jerarquía; de ella se derivará `D`;
2. `D` **hereda** todas las componentes (atributos y métodos) de `B` sin volver a definirlas o implementarlas, **aunque** obviamente no puede acceder a aquellas que tengan un modificador de visibilidad `private`;
3. `D` **especializa** a `B` bien definiendo nuevas componentes, como mínimo sus métodos constructores, o bien cambiando el significado de los métodos heredados de `B` mediante **Sobrescritura**. Por tanto `D` es una clase completamente nueva aunque herede de `B` y los cambios que sufra **no** afectarán a `B`;
4. `D` es de tipo compatible con `B`; sin embargo, **no** son compatibles **ni** `B` con `D` **ni** `D` con sus clases **Hermanas**.

Las ventajas que proporcionan estas cuatro características a la hora de programar son tantas que hacen de **la Herencia el mecanismo más importante de Reutilización de Código que proporciona un lenguaje POO**; para aprovecharlas al máximo los creadores del lenguaje Java organizaron todas sus clases en una Jerarquía cuya Raíz es `Object`, de donde ."[en] Java cualquier cosa es un **ES UN** Object". A pesar de ello, sin embargo, un programador novel sólo logra apreciar las más obvias o directas de la Herencia. Por ejemplo puede notar que en la Jerarquía `Figura` las propiedades 1, 2 y 3 garantizan en conjunto que `Circulo`, `Rectangulo` y `Cuadrado` heredan todas las componentes de `Figura` por ser sus Derivadas y, por tanto, es posible aplicar a cualquiera de sus objetos (vía operador `.`) no sólo sus métodos específicos sino también todos los de la clase Base `Figura`; de las propiedades 2 y 3 se desprende además que el diseño y mantenimiento de las Derivadas de `Figura`, tanto las actuales como las que en un futuro se añadan, es menor que el que se realiza cuando no forman parte de una Jerarquía porque heredan todas las componentes de `Figura` sin volver a definirlas o implementarlas y los cambios que en ellas se produzcan no afectan ni a `Figura` ni a sus Hermanas.

Ahora bien, si esto es así, le puede resultar difícil entender el sentido o la ventaja que supone la Sobrescritura de algunos métodos de `Figura` en sus Derivadas: si un método se define en `Figura` precisamente por ser independiente del tipo de Figura sobre la que se aplica, ¿para qué modificar su comportamiento en `Circulo` o `Rectangulo` o `Cuadrado`? Algo similar

le puede suceder con la última propiedad, la compatibilidad de los tipos `Circulo`, `Rectangulo` y `Cuadrado` con `Figura`: si vía `new` se pueden crear objetos de cualquiera de todas estas clases pero la Herencia no permite aplicar a uno de tipo `Figura` los métodos de sus Derivadas por muy compatibles que sean sus tipos, ¿qué ventaja puede suponer declarar variables de tipo `Figura`?

Obviamente conviene aclarar estas dos últimas cuestiones antes de plantear el proceso de diseño de una Jerarquía, la pieza clave que se buscaba para simplificar el desarrollo de cualquier aplicación en la que se manipulen Grupos hetereogéneos de objetos. Para ello los contenidos de este apartado se han organizado en dos secciones. En la primera se plantea la (re)utilización de una Jerarquía en el diseño de nuevas clases con el fin de introducir el concepto de **Polimorfismo** como la mayor ventaja que aporta la Herencia y sin la cual no se pueden entender ni el papel de la Sobrescritura ni las ventajas que aporta la Herencia a la hora de ampliar y mantener con ciertas garantías Software ya existente; tras haber comprobado que el objetivo último que se persigue al diseñar una Jerarquía es poder declarar más tarde variables polimórficas de su clase Base, en la segunda sección del apartado se presenta el proceso de diseño (pasos a dar y la sintaxis asociada) de una Jerarquía de clases Java de usuario.

2.1. Uso de una Jerarquía: el Polimorfismo y otras ventajas de la Herencia

La principal cuestión a la que se pretende responder en esta sección ya ha sido formulada: "por muy compatibles que sean sus tipos, ¿qué sentido o ventaja puede suponer declarar variables del tipo Base de una Jerarquía?. Para contestarla se impone recordar primero qué supone en ausencia de Herencia el que un tipo dado sea o no compatible con otro en el momento de usar variables de dicho tipo; para ello supóngase diseñado el siguiente programa Java:

```
package ejemplos.tema1.gestionFigurasV0;
import ejemplos.tema1.lasFigurasV0; import java.awt.geom.*;
public class TestCompatibilidad {
 public static void main (String args[]){
    /*****se trabaja con variables de tipos primitivos y sus valores*****/
    int i = 17; double d = i; System.out.println("d contiene el valor "+d); boolean b = i;
   // Mostrar por pantalla las componentes y la media de un Grupo de Enteros
   int grupoInt[] = {4, 7, 10, 12, 15}; double media = 0;
   System.out.print("Componentes del Grupo de Enteros: ");
   for ( int indice = 0; indice < grupoInt.length; indice++ ){
        media += grupoInt[indice]; System.out.print(grupoInt[indice]+" ");
   }
   System.out.println("con media "+media/grupoInt.length);
   /*****se pasa a trabajar con variables Referencia a objetos*****/
   Circulo c = new Rectangulo();  Rectangulo r = new Figura();
   // Crear un Grupo de Círculos y mostrar por pantalla sus componentes
   Point2D.Double p = new Point2D.Double();
   Circulo grupoC[] = {new Circulo(), new Circulo(3,"verde",p), new Circulo(5.4,"azul",p)};
   System.out.println("Componentes del Grupo de Círculos: ");
   for ( int indice = 0; indice < grupoC.length; indice++ )
        System.out.println(" "+grupoC[indice].toString());
   // Crear un Grupo de Figuras hetereogeneo y mostrar por pantalla sus componentes
   Figura grupoF[] = {new Circulo(), new Rectangulo(), new Cuadrado()};
   System.out.println("Componentes del Grupo de Figuras: ");
   for ( int indice = 0; indice < grupoF.length; indice++ )
        System.out.println(" "+grupoF[indice].toString());
 }
}
```

Recordando que un lenguaje fuertemente tipado como Java penaliza con errores de compilación y ejecución la evaluación de una expresión diseñada sin garantizar la igualdad o compatibilidad de tipos entre sus componentes, la compilación de `TestCompatibilidad` provocaría los siguientes errores:

- en relación a variables de tipos primitivos, el único error se produce en la evaluación de «`boolean b = i;`» es «`incompatible types-found int but expected boolean`», pues como ya se sabe los tipos de `b` e `i` no son ni compatibles ni idénticos. No hay problema sin embargo en asignar a un `double d` un `int i` pues el tipo `int` es compatible con `double`, aunque no viceversa por una posible pérdida de precisión;
- en relación a variables Referencia a objetos de una clase, si `Figura`, `Circulo` y `Rectangulo` **no** forman parte de una Jerarquía los errores de incompatibilidad se producen al intentar apuntar a un objeto con una variable Referencia de tipo diferente al suyo; así por ejemplo las instrucciones «`Circulo c = new Rectangulo();`», «`Rectangulo r = new Figura();`» y «`Figura grupoF[]={new Circulo(),new Rectangulo(),new Cuadrado()};`» provocan el mismo tipo de error, como delata la idéntica estructura de los tres mensajes que los acompañan: «`incompatible types-found Rectangulo but expected Circulo`» para la primera, «`incompatible types-found Figura but expected Rectangulo`» para la segunda e «`incompatible types-found Circulo but expected Figura`» para la tercera. El resto de expresiones no da problemas porque los tipos de las variables involucradas en ellas siempre referencian un objeto de tipo idéntico al suyo y, por tanto, compatible.

Más allá de los errores que su compilación puede provocar, lo que realmente interesa señalar del diseño de `TestCompatibilidad` es la ventaja que obtiene un programador al declarar y manipular variables de tipo compatible aunque distinto a la hora de calcular la media de un Grupo de Enteros como `grupoInt`: por ser compatible `int` con `double`, al ejecutar la instrucción «`media += grupoInt[indice];`» los valores de tipo `int` que componen `grupoInt` serán tratados automáticamente como si fueran de tipo `double` y, por tanto, se irán sumando al valor `double` que contiene la variable `media`. En otras palabras, lo que garantiza la compatibilidad de `int` con `double` es que se puede obtener la media de un Grupo de `int` de la misma forma que si fuera de `double` pero sin modificar el tipo y valor de sus componentes, tal como ilustra la segunda línea de la siguiente traza de `TestCompatibilidad`:

```
d contiene el valor 17.0
Componentes del Grupo de Enteros: 4 7 10 12 15 con media 9.6
Componentes del Grupo de Círculos:
 Círculo de radio 3.00, color rojo y centro Point2D.Double[0.0, 0.0]
 Círculo de radio 3.00, color verde y centro Point2D.Double[0.0, 0.0]
 Círculo de radio 5.40, color azul y centro Point2D.Double[0.0, 0.0]
```

Entonces, usando como símil la compatibilidad de `int` con `double`, para poder compilar y ejecutar las cuatro últimas instrucciones de `TestCompatibilidad` que afectan a `grupoF` basta con que `Figura`, `Circulo`, `Rectangulo` y `Cuadrado` **sí** formen una Jerarquía: al ejecutar la instrucción «`Figura grupoF[] = {new Circulo(), new Rectangulo(), new Cuadrado()};`» las Referencias a los objetos `new Circulo()`, `new Rectangulo()` y `new Cuadrado()`, **que no éstos**, se tratarán automáticamente como Referencias de tipo `Figura` y así `grupoF` se podrá inicializar con tres objetos de tipo distinto a `Figura`. La Herencia garantiza así poder trabajar con el Grupo heterogéneo de Figuras `grupoF` de la misma forma que con el Grupo homogéneo de Círculos `grupoC`; para comprobarlo basta diseñar la Jerarquía `Figura` en el

paquete `ejemplos.tema1.lasFigurasV1`, ejecutar `TestCompatibilidad` sobre ella y observar qué resultado se obtiene a partir de la séptima línea:

```
d contiene el valor 17.0
Componentes del Grupo de Enteros: 4 7 10 12 15 con media 9.6
Componentes del Grupo de Círculos:
 Círculo de radio 3.00, color rojo y centro Point2D.Double[0.0, 0.0]
 Círculo de radio 3.00, color verde y centro Point2D.Double[0.0, 0.0]
 Círculo de radio 5.40, color azul y centro Point2D.Double[0.0, 0.0]
Componentes del Grupo de Figuras:
 Círculo de radio 3.00, color rojo y centro Point2D.Double[0.0, 0.0]
 Rectángulo con base 3.00 y altura 4,00, color rojo y centro Point2D.Double[0.0, 0.0]
 Cuadrado de lado 3.00, color rojo y centro Point2D.Double[0.0, 0.0]
```

Generalizando a partir del ejemplo ya se puede dar una primera respuesta a la pregunta con la que se abría la sección:

> interesa declarar variables de la clase Base de una Jerarquía porque **con un único tipo de Referencia se pueden crear vía `new` y manipular vía operador . tantos tipos distintos de objetos como clases Derivadas tenga Base**, lo que supone tratarlos (casi) a todos los efectos como si pertenecieran a una misma clase. Gracias a ello la reutilización de la Jerarquía resulta mucho más sencilla y efectiva que la de las clases que la componen por separado, o en ausencia de Herencia.

Esta primera respuesta se puede enunciar con mayor propiedad relacionando el concepto de Polimorfismo y el de variable de la clase Base de una Jerarquía. En efecto, sea la siguiente la **definición de Polimorfismo**:

> una variable Referencia es polimórfica si su tipo de declaración o **tipo Estático no coincide** con el del objeto al que referencia o **tipo Dinámico** de la variable.

Por tanto, **una variable de la clase Base de una Jerarquía es polimórfica si referencia a cualquier objeto de tipo compatible con el suyo**; por ejemplo en el método `main` de `TestCompatibilidad` son polimórficas las variables Referencia `grupoF[0]`, `grupoF[1]` y `grupoF[2]` pues su tipo de declaración o tipo Estático es `Figura`, el tipo del `array grupoF`, mientras que su tipo Dinámico o tipo del objeto al que referencia tras la inicialización del `grupoF` es `Circulo` para `grupoF[0]`, `Rectangulo` para `grupoF[1]` y `Cuadrado` para `grupo[2]`. Dicho esto conviene subrayar que el Polimorfismo de per-se no garantiza **ni** la compatibilidad de los tipos Dinámico y Estático de una variable **ni** la aplicación a dicha variable de cualquier método de cualquiera de estos tipos, sólo lo hace en presencia de Herencia y bajo las condiciones establecidas por ésta; volviendo al ejemplo de `TestCompatibilidad`, recuérdese que únicamente se puede inicializar `grupoF` con tres Figuras de tipo heterogéneo y obtener su `toString()` sin problemas de compilación cuando se garantiza que la clase `Figura` es la Base de una Jerarquía, que `Circulo`, `Rectangulo` y `Cuadrado` son sus Derivadas y, finalmente, que el programador usa de forma debida las variables de tipo Estático `Figura`.

En conclusión, el Polimorfismo es la mayor ventaja que obtiene el programador usando una Jerarquía **pero** siempre y cuando respete las normas que establece la Herencia. En los siguientes puntos de esta sección se repasan en detalle estas normas estudiando precisamente el procedimiento y los mecanismos de conversión de tipos y selección de métodos que emplea el intérprete Java para garantizarlas; para ilustrar este estudio convenientemente se utilizará como

ejemplo el código de `TestJerarquiaFigura`, un programa que ha sido diseñado, tal como aparece comentado en el código adjunto, para comprobar la funcionalidad básica de un Grupo heterogéneo de Figuras de la Jerarquía ubicada en el paquete `ejemplos.tema1.lasFigurasV1`.

```
package ejemplos.tema1.gestionFigurasV1;
import ejemplos.tema1.lasFigurasV1; import java.text.*; import java.util.*;
public class TestJerarquiaFigura {
 public static void main(String args[]){
     // inicializar un Grupo de Figuras: crear un Grupo de 3 Figuras e
     // inicializarlo con un Círculo, un Rectángulo y un Cuadrado estándar
     int talla = 3; Figura grupo[] = new Figura[talla];
     grupo[0] = new Circulo(); grupo[1] = new Rectangulo(); grupo[2] = new Cuadrado();
     // toString de un Grupo de Figuras: obtener el String que representa
     // las componentes del Grupo y mostrarlo por pantalla
     System.out.println("Inicializado un Grupo con las siguientes "+talla+" Figuras:");
     String res = "";
     for ( int i = 0 ; i < talla ; i++ ) res += " "+grupo[i].toString()+"/n";
     System.out.println(res);
     // buscar una Figura dada en un Grupo de Figuras: obtener la primera posición
     // de una Figura aBuscar en el Grupo, pero si aBuscar no está en el Grupo advertirlo
     System.out.println("Buscar en el Grupo una Figura");
     Scanner teclado = new Scanner(System.in).useLocale(new Locale("es", "US"));
     Figura aBuscar = Figura.leer(teclado); int i = 0;
     while ( i < talla && !grupo[i].equals(aBuscar) ) i++;
     if ( i != talla )
     System.out.println("Primera aparición de "+aBuscar+" en la posición "+i+" del Grupo");
     else System.out.println("La Figura "+aBuscar+" NO está en el Grupo");
     // área de un Grupo de Figuras: obtener el área del Grupo sumando las áreas
     // de las Figuras que lo componen y mostrarla por pantalla
     double areaDelGrupo = 0.0;
     ....
 }
}
```

2.1.1. Reglas de compatibilidad y mecanismos de conversión de tipos en presencia de Polimorfismo

Para evaluar una expresión en la que aparecen variables polimórficas el intérprete Java procede en primer lugar, en tiempo de compilación, a comprobar si sus tipos Dinámicos son compatibles con su tipos Estáticos; si lo son convierte el tipo Dinámico de cada variable polimórfica en su tipo Estático o **realiza una Conversión de Ampliación** (***Upcasting***) y si no lo son produce un error con mensaje «`incompatible types - found ... but expected ...`» para advertir del problema al programador y acaba la evaluación de la expresión. Para ilustrar el proceso descrito considérese la expresión «`grupo[0] = new Circulo();`», la primera instrucción del programa ejemplo `TestJerarquiaFigura` donde aparecen variables polimórficas. Como la variable `grupo[0]` tiene un tipo Dinámico `Circulo` (el de la Referencia al objeto `new Circulo()`) compatible por Herencia con su tipo Estático `Figura`, el intérprete realiza una Conversión de Ampliación y convierte la Referencia a `new Circulo()` en una referencia de tipo `Figura`; gracias a ello, y como pretendía el programador, en tiempo de ejecución el

intérprete asigna a una componente de un `array` de `Figura` un objeto de tipo compatible con el suyo aunque distinto, y por extensión inicializa un `array` con distintos tipos de Figuras que, por ende, son incompatibles entre sí. Nótese también que si `Circulo` no es Derivada de `Figura` la compilación de la misma instrucción provoca un error que hace imposible su ejecución y al que acompaña el mensaje «`incompatible types - found Circulo expected Figura`». Este ejemplo permite señalar ya una primera norma de uso del Polimorfismo: la Herencia implica Polimorfismo pero el Polimorfismo de per-se, sin Herencia, se penaliza con los mismos errores de compilación que provoca la evaluación de una expresión diseñada sin garantizar la igualdad de tipos entre sus componentes.

Siguiendo con la compilación de `TestJerarquiaFigura` la siguiente expresión en la que cabe notar el uso de la Conversión de Ampliación es «`!grupo[i].equals(aBuscar)`», pues a primera vista no resulta evidente cuál es la variable polimórfica que la sufre. Recordando que el perfil del método a aplicar sobre `grupo[i]` es «`public boolean equals(Object x)`» queda claro que la variable en cuestión es el parámetro formal de entrada `x` de dicho método: su tipo Estático es `Object`, el de su declaración en el perfil de `equals`, mientras que su tipo Dinámico es `Figura`, el de la variable `aBuscar` que se le pasa a `equals` como parámetro actual; cuando el intérprete lo detecta, y puesto que `Object` es la Raíz de la Jerarquía de clases Java, convierte la Referencia `aBuscar` en una Referencia de tipo `Object` y con ello, tal como pretendía el programador, permite que ya en tiempo de ejecución se pueda comprobar si las Figuras `grupo[i]` y `aBuscar` son iguales, y por extensión buscar una Figura en un Grupo. El análisis realizado permite entender también el motivo por el que no provoca error de compilación alguno que el método `equals` reciba como parámetro actual cualquier tipo de objeto Java distinto de `Figura`, como por ejemplo `"Este String"`, o que se inicialice un `Object grupo[]` con objetos de tipos distintos entre sí y también de `Object`: si el tipo Estático de una variable polimórfica es `Object` entonces cualquiera que sea su tipo Dinámico es compatible con él y el intérprete Java puede realizar la correspondiente Conversión de Ampliación.

Tras haber planteado situaciones en las que el programador usa el Polimorfismo sin ignorar las condiciones de compatibilidad que impone la Herencia a las clases de una Jerarquía, en lo que sigue se presentan algunos casos de interés en los que **no** lo hace y, en consecuencia, obliga al intérprete a advertírselo mediante los correspondientes errores de compilación y, en algunos casos, de ejecución. Supóngase por ejemplo que en `TestJerarquiaFigura` el programador escribe la expresión «`Circulo aBuscar = Figura.leer(teclado)`» en lugar de la actual «`Figura aBuscar = Figura.leer(teclado)`»; al compilarla, aún en presencia de Herencia y Polimorfismo, el mensaje «`incompatible types - found Figura but expected Circulo`» aparece asociado a ella; nótese que sucede algo muy parecido si el tipo Estático de `aBuscar` es `Rectangulo` o `Cuadrado` en lugar de `Circulo`: aunque el intérprete Java siempre encuentra (*found*) `Figura` espera (*expected*) `Rectangulo` o `Cuadrado` respectivamente. El origen de cualquiera de estos errores de compilación se entiende de inmediato cuando se recuerda que la Herencia sólo hace compatibles con `Figura` los tipos de sus Derivadas y nunca viceversa: como el tipo Dinámico de la variable `aBuscar` es `Figura` **no** es compatible con su tipo Estático el intérprete se lo advierte al programador con el correspondiente mensaje de error. Para solventar el problema la solución sería, simplemente, substituir el tipo Estático de `aBuscar` por `Figura`, esto es igualar sus tipos Estático y Dinámico, y con ello garantizar la compilación correcta de la instrucción y, más importante todavía, mantener el objetivo con el que se diseñó la Jerarquía.

Existe sin embargo otra solución menos general pero que interesa plantear porque ya se ha utilizado con los tipos primitivos y porque además es la única que le resta al programador en determinadas circunstancias, como se verá un poco más adelante al calcular el área de `grupo` en `TestJerarquiaFigura`. A saber: el programador puede pensar que al igual que escribe «`(int)vD`» para convertir el valor de una variable `double vD` en un `int` sin modificar su tipo, para resolver el error que provoca «`Circulo aBuscar = Figura.leer(teclado)`» puede introducir delante de `Figura.leer(teclado)` un ***Casting*** a `Circulo` para igualar los tipos Dinámico (`Figura`) y Estático (`Circulo`) de la variable `aBuscar`. Y tiene razón: la expresión «`Circulo aBuscar = (Circulo)Figura.leer(teclado)`» no contiene ningún error de compatibilidad, pues al encontrar el *Casting* a `Circulo` el intérprete Java convierte la Referencia de tipo `Figura` resultado de `Figura.leer(teclado)` en una Referencia de tipo `Circulo` o **realiza una Conversión de Restricción** de `Figura` a `Circulo`. Pero por desgracia el programador está ignorando en su solución los efectos secundarios que el *Casting* impuesto puede ocasionar: el intérprete Java convierte la Referencia de tipo `Figura` obtenida al ejecutar `Figura.leer(teclado)` en una Referencia de tipo `Circulo` obligado por el `Casting` y, por tanto, si el objeto leido por teclado es de tipo `Rectangulo` o `Cuadrado` producirá un error de ejecución `ClassCastException` al intentar apuntarlo con una Referencia de tipo `Circulo`. Nótese que el programador es penalizado esta vez en tiempo de ejecución por haber ignorado que en una Jerarquía las clases Derivadas son incompatibles entre sí y que la Herencia sólo hace compatibles los tipos de las Derivadas con el de la clase Base.

Llegados a este punto la pregunta es cómo resolver este error de ejecución, pues mientras se desconozca el tipo Dinámico de la variable `aBuscar`, el tipo específico de Figura que introducirá el usuario al ejecutarse `Figura.leer(teclado)`, no se sabrá cuál es el *Casting* adecuado a realizar. Como ilustra el código que figura a continuación la respuesta es usar el operador binario Java `instanceof`, que comprueba en tiempo de ejecución si el objeto referenciado por su primer operando es una instancia de la clase Java que tiene como segundo operando:

```
...
Figura aBuscar = Figura.leer(teclado);
// aBuscar puede referenciar un Círculo o un Rectángulo o un Cuadrado sq
Circulo c = null; Rectangulo r = null; Cuadrado sq = null;
// uso de instanceof para prevenir ClassCastException:
// si  aBuscar referencia un Círculo,
if (   aBuscar instanceof  Circulo )              c = (Circulo)aBuscar;
else // si   aBuscar referencia un Cuadrado,
        if ( aBuscar instanceof  Cuadrado )       sq = (Cuadrado)aBuscar;
        else /*aBuscar referencia un Rectángulo*/ r = (Rectangulo)aBuscar;
...
```

En este código en particular `instanceof` permite comprobar en tiempo de ejecución el tipo Dinámico de la variable polimórfica `aBuscar`, su primer operando: si al `leer` de `teclado` se obtiene un objeto de tipo `Circulo` entonces `c` será la variable que lo referenciará previo y apropiado *Casting* a `Circulo`; sino, si se ha leído un `Cuadrado` entonces `sq` será la variable que lo referenciará .... Gracias a ello se consigue que independientemente del tipo de Figura que se lea se ejecute una Conversión de Restricción correcta y, por tanto, se eliminen ya en la fase de diseño los posibles errores de ejecución debidos a un *Casting* inadecuado.

Pero como siempre ocurre en presencia de Herencia y Polimorfismo un uso descuidado del operador `instanceof`, más específicamente una mala elección de su segundo operando, puede ocasionar errores de ejecución que en algunos casos resultan difíciles de subsanar. En efecto,

nótese que si la clase a la que pertenece el primer operando de `instanceof` ES UNA Derivada de su segundo operando entonces también es una instancia de él por compatibilidad de tipos y, por tanto, la evaluación de `instanceof` con el primer y segundo operando de tipos compatibles pero no idénticos resulta ser también `true`; así por ejemplo las siguientes expresiones se evaluarán a `true` independientemente de si el tipo Dinámico de `aBuscar` es `Circulo` o `Rectangulo` o `Cuadrado`: «`aBuscar instanceof Figura`» y «`aBuscar instanceof Object`». Por tanto, si en lugar del código propuesto más arriba para introducir el uso de `instanceof` se tuviese

```
...
Figura aBuscar = Figura.leer(teclado);
Circulo c = null; Rectangulo r = null; Cuadrado sq = null;
if      ( aBuscar instanceof  Figura   )  c  = (Circulo)aBuscar;
else if ( aBuscar instanceof  Cuadrado )  sq = (Cuadrado)aBuscar;
     else                                 r =  (Rectangulo)aBuscar;
...
```

, al ejecutarlo y leer una Figura que no fuese un Círculo se produciría en la instrucción que contiene el primer *Casting* un error `ClassCastException` por intentar apuntar con una Referencia de tipo `Circulo` a un objeto de tipo `Cuadrado` o `Rectangulo`. Si bien este error es fácil de subsanar no ocurre lo mismo si el código que se escribe es el adjunto a la siguiente **cuestión:** recordando que un Cuadrado ES UN Rectángulo de base igual a su altura y suponiendo que la Figura leída de teclado es un Cuadrado, trácese la ejecución del siguiente código y en función del resultado obtenido coméntese su corrección.

```
...
Figura aBuscar = Figura.leer(teclado);
Circulo c = null; Rectangulo r = null; Cuadrado sq = null;
if      ( aBuscar instanceof    Circulo  )  c  = (Circulo)aBuscar;
else if ( aBuscar instanceof  Rectangulo )  r =  (Rectangulo)aBuscar;
     else                                   sq = (Cuadrado)aBuscar;
...
System.out.println("Figura aBuscar: "+aBuscar);
System.out.println("Círculo c: "+c+" Rectangulo r: "+r+" Cuadrado sq: "+sq);
...
```

2.1.2. Reglas de aplicación y mecanismos de selección de métodos en presencia de Polimorfismo: Enlace Dinámico o la utilidad de la Sobrescritura

Una vez establecida con éxito la compatibilidad de los tipos de las variables de una expresión y aún en tiempo de compilación, si el intérprete Java detecta el identificador de un método pasa a comprobar si tiene acceso a su código en la clase del **tipo Estático** de la variable a la que se le aplica; si lo tiene, y ya en tiempo de ejecución, el intérprete aplica a dicha variable el método definido en la clase de su **tipo Dinámico** y si no lo tiene produce un error de compilación con mensaje «`cannot resolve symbol - method ...`» para advertir del problema al programador y, con ello, finalizar la evaluación de la expresión. El procedimiento que se acaba de enunciar recibe el nombre de **Enlace Dinámico** o **Selección de Método Dinámica**, pues establece que el método a ejecutar sobre una variable sea el definido por su tipo Dinámico, y gracias a él se obtienen la gran mayoría de las ventajas que el uso de variables polimórficas de la clase Base de una Jerarquía brinda a un programador; piénsese sino, por ejemplo, en los dos primeros bucles del `main` de `TestJerarquiaFigura` que se reproducen a continuación:

```
      ...
      // toString de un Grupo de Figuras: obtener el String que representa
      // las componentes del Grupo y mostrarlo por pantalla
      System.out.println("Inicializado un Grupo con las siguientes "+talla+" Figuras:");
      String res = "";
      for ( int i = 0 ; i < talla ; i++ ) res += " "+grupo[i].toString()+"/n";
      System.out.println(res);
      // buscar una Figura dada en un Grupo de Figuras: obtener la primera aparición
      // de una Figura aBuscar en el Grupo, pero si aBuscar no está en el Grupo advertirlo
      System.out.println("Buscar en el Grupo una Figura");
      Scanner teclado = new Scanner(System.in).useLocale(new Locale("es", "US"));
      Figura aBuscar = Figura.leer(teclado); int i = 0;
      while ( i < talla && !grupo[i].equals(aBuscar) ) i++;
      if ( i != talla )
      System.out.println("Primera aparición de "+aBuscar+" en la posición "+i+" del Grupo");
      else System.out.println("La Figura "+aBuscar+" NO está en el Grupo");
      ....
  }
}
```

Tras haber inicializado el `array grupo` con objetos de tipo heterogéneo gracias al Polimorfismo de las variables de la clase `Figura`, la existencia del Enlace Dinámico hace posible que el resultado de la ejecución de cada uno de estos bucles sea el que el programador pretendía siguiendo la Especificación de la Jerarquía: para el primer bucle mostrar las distintas Figuras que componen `grupo` vía aplicación del método `toString()` a cada uno de los objetos que lo conforman y con el segundo determinar si una Figura dada `aBuscar` está en `grupo` comparándola vía aplicación del método `equals(aBuscar)` con cada uno de ellos. En efecto, cuando el intérprete Java detecta el identificador `toString` en la expresión «`grupo[i].toString()`» comprueba si tiene acceso a su código en la clase `Figura` o clase del tipo Estático de `grupo[i]`; como sí lo tiene, **y por Herencia cualquier Derivada de `Figura` también**, en tiempo de ejecución aplicará efectivamente a `grupo[i]` el método definido en la clase de su tipo Dinámico o tipo del objeto al que `grupo[i]` referencia, i.e. el `toString()` de la clase `Circulo` al ejecutar `grupo[0].toString()`, el de la clase `Rectangulo` al ejecutar `grupo[1].toString()` y, finalmente, el de la clase `Cuadrado` al ejecutar `grupo[2].toString()`. De igual manera el Enlace Dinámico explica la ejecución del método `equals` en la expresión «`grupo[i].equals(aBuscar)`»: una vez comprobado que el método `equals` está definido en la clase `Figura`, la clase del tipo Estático de `grupo[i]`, por Herencia el intérprete podrá aplicar sobre esta componente el método `equals` de la clase del objeto al que apunta. Ahora bien, **y como resultado de que `toString` se sobrescribe en las Derivadas de `Figura` pero `equals` no**, el intérprete Java aplica a cada componente de `grupo` un método `toString` **distinto**, el `toString` definido en la correspondiente Derivada cambiando sólo el código del heredado de `Figura`, **pero** el **mismo** método `equals`, el definido en la clase Base `Figura` y que heredan sus Derivadas.

Nótese entonces que el Enlace Dinámico permite resolver las situaciones de sobrecarga de métodos que la Sobrescritura provoca dentro de una Jerarquía; es más, como dependiendo de que un método de la clase Base de una Jerarquía se sobrescriba o no en sus Derivadas el Enlace Dinámico aplicará a una variable polimórfica un método que bien especializa o bien mantiene invariante un comportamiento definido en la clase Base, el Enlace Dinámico hace de la Sobrescritura un verdadero regulador de la Herencia con el que el programador puede explotar al máximo todas las ventajas que le proporciona el uso de variables polimórficas.

La propia Jerarquía de clases Java proporciona un ejemplo claro de ello: los métodos que se definen en su clase Base `Object` se pueden sobrescribir en cualquier clase Java con el propósito de especializar o adecuar su comportamiento al que conviene a esa clase, típicamente sus métodos `toString()` y `equals(Object x)`; ahora bien, si el programador olvida sobrescribirlos en una clase el intérprete Java **no** se encargará de recordárselo y seguirá aplicando a sus objetos los métodos definidos en `Object`, lo que en general suele producir bien un resultado que no es el pretendido o bien un resultado erróneo. Por ejemplo, si en la Jerarquía de Figuras no se sobrescriben ni `toString` ni `equals` entonces el resultado de la ejecución de los dos primeros bucles de `TestJerarquiaFigura` sería el siguiente:

```
...
Inicializado un Grupo con las siguientes 3 Figuras:
ejemplos.tema1.lasFigurasV1.Circulo@1f2cea2
ejemplos.tema1.lasFigurasV1.Rectangulo@1dc0e7a
ejemplos.tema1.lasFigurasV1.Cuadrado@3a9bba

Buscar en el Grupo una Figura
Para leer una Figura desde teclado PULSE
1 para leer  Círculo
2 para leer  Rectángulo
3 para leer  Cuadrado
1
********Introduzca las componentes del Círculo *********
¿radio?: 3
¿color?: rojo
¿coordenada x del centro?: 0
¿coordenada y del centro?: 0
La Figura ejemplos.tema1.lasFigurasV1.Circulo@163f7a1 NO está en el Grupo
...
```

Nótese que el comportamiento que por Enlace Dinámico reproducen los métodos `toString` y `equals` que se aplican a las componentes de `grupo` son los mismos que se aplicarían a cualquier otro `Object`, demasiado generales por tanto como para que su resultado sea el esperado en el caso de `toString` y el correcto en el caso de `equals`. La solución a este tipo de errores es sencilla: sobrescribir tanto el método `toString()` de `Object` en la clase `Figura` como el método `toString()` de `Figura` en sus Derivadas y, además, sobrescribir el método `equals` de `Object` en `Figura`, pues al servir como criterio de comparación de distintos tipos de Figuras no debiera depender de las características particulares de los objetos a comparar.

Ya para finalizar este punto queda por analizar qué ocurre cuando un programador ignora en presencia de Polimorfismo el hecho de que los únicos métodos que comparten por Herencia la clase Base de una Jerarquía y sus Derivadas son los definidos en Base, y no los métodos nuevos que puedan definir las Derivadas para especializar a Base. Así por ejemplo supóngase que siguiendo el patrón de Recorrido establecido hasta el momento en el `main` de `TestJerarquiaFigura` el programador escribe las líneas de código que figuran a continuación para calcular el área del Grupo heterogéneo de Figuras `grupo`:

```
....
double areaDelGrupo = 0.0;
for ( int i = 0 ; i < talla ; i++ ) areaDelGrupo += grupo[i].area();
....
```

Cuando el intérprete Java comprueba que el identificador de método `area` que aparece en la expresión «`areaDelGrupo += grupo[i].area()`» no corresponde al de un método definido en `Figura`, el tipo Estático de la variable sobre la que se aplica, produce un error de compilación con mensaje «`cannot find symbol - method area()`» y finaliza la evaluación de la expresión; con ello le recuerda al programador que sólo se puede aplicar al objeto referenciado por una componente de `grupo` un método heredado y, por tanto, definido en `Figura`. Pero si un método público como `area` está definido en una clase dada ¿por qué no se puede aplicar a uno de sus objetos? Pues porque a bajo nivel una variable polimórfica de tipo Estático `Base` -léase `Figura`- y tipo Dinámico `Derivada` -léase `Circulo` o `Rectangulo` o `Cuadrado`- es una Referencia de tipo `Base` a un objeto de tipo `Derivada` y, por tanto, al objeto al que referencia se le pueden aplicar única y exclusivamente los métodos definidos en `Base`, esto es los que `Derivada` hereda de `Base`; para dejar la cuestión más clara basta recordar el motivo por el que el intérprete Java provoca un error con mensaje «`cannot find symbol - method toUpperCase()`» al evaluar la expresión «`areaDelGrupo += grupo[i].toUpperCase()`»: para señalar que se está aplicando a una variable Referencia de la clase `Figura` un método `toUpperCase()` no definido en ella, aunque `toUpperCase()` sí sea un método de la clase `String`.

La mejor manera posible de solucionar este problema sería primero definir el método `area()` en la clase `Figura` y luego especializar su comportamiento en sus `Derivadas` mediante Sobrescritura, pero no será hasta el último apartado del tema cuando se conozca la forma de llevarla a la práctica. Una solución menos eficaz pero que ya se puede proponer es combinar el uso del operador `instanceof` con el del *Casting*; en concreto, consiste en establecer primero con `instanceof` el tipo del objeto apuntado por cada Referencia de `grupo`, convertir después el tipo de cada Referencia al del objeto al que apunta mediante *Casting* y, finalmente, aplicarle el método `area()`. Se tendría así el siguiente código:

```
...
double areaDelGrupo = 0.0;
for ( i = 0 ; i < talla ; i++ )
 if ( grupo[i] instanceof Circulo )       areaDelGrupo += ((Circulo)grupo[i]).area();
 else if ( grupo[i] instanceof Cuadrado ) areaDelGrupo += ((Cuadrado)grupo[i]).area();
      else                                areaDelGrupo += ((Rectangulo)grupo[i]).area();
 ...
```

Para el intérprete Java tal solución es aceptable porque al deshacer vía *Casting* el Polimorfismo de cada componente de `grupo` ya tiene acceso al código del método `area()` de la `Derivada` en cuestión; por ejemplo, al evaluar «`areaDelGrupo += ((Circulo)grupo[i]).area()`» puede acceder al código del método `area()` de `Circulo`, la clase del tipo Estático y Dinámico de `(Circulo)grupo[i]`.

Ejercicios propuestos: para cerrar por el momento el tema del Polimorfismo -en la siguiente sección de este apartado se presentarán nuevos ejemplos sobre el uso de variables polimórficas al diseñar la clase `Base` de la Jerarquía- y suponiendo ya diseñada la Jerarquía de Figuras del paquete `ejemplos.tema1.lasFigurasV1`, se propone la realización de los siguientes ejercicios:

1. Supóngase inicializado el `grupo` de Figuras de `TestJerarquiaFigura` como sigue:

```
...
Scanner teclado = new Scanner(System.in).useLocale(new Locale("es", "US"));
System.out.println("Introduzca el número de Figuras del Grupo: ");
int talla = teclado.nextInt(); Figura grupo[] = new Figura[talla];
for ( int i = 0 ; i < talla ; i++ ) grupo[i] = Figura.leer(teclado);
....
```

Modifíquese el `main` de este programa para que tras finalizar la inicialización de un Grupo con `talla` Figuras se muestre por pantalla el número de Círculos, Cuadrados y Rectángulos que lo componen.

2. Se quiere comprobar el funcionamiento de la clase `Figura` con el siguiente test:

```
package ejemplos.tema1.gestionFigurasV1;
import ejemplos.tema1.lasFigurasV1.*;
public class TestPolimorfismo {
  public static void main(String args[]){
    // creación e inicialización del grupo
    Figura grupo[] = {new Cuadrado(), new Rectangulo(), new Circulo()};
    // gestión del grupo: mostrar por pantalla el área de sus componentes
    Figura estandar = grupo[0];
    System.out.println("Área del Cuadrado del Grupo: "+((Circulo)estandar).area());
    Rectangulo rEstandar = grupo[1];
    System.out.println("Área del Rectángulo del Grupo: "+rEstandar.area());
    estandar = grupo[2];
    System.out.println("Área del Círculo del Grupo: "+estandar.area());
  }
}
```

Si la compilación de este programa diera algún error indíquese el motivo por el que se produce, solvéntese y luego trácese la ejecución del programa; si se produce un error entonces indíquese la manera de evitarlo.

3. Diséñese en el paquete `ejemplos.tema1.lasFigurasV1` una clase `ArrayGrupoDeFiguras` que represente a un Grupo heterogéneo de Figuras con la siguiente Especificación:

---

ejemplos.tema1.lasFigurasV1

Class ArrayGrupoDeFiguras

java.lang.Object
  └ **ejemplos.tema1.lasFigurasV1.ArrayGrupoDeFiguras**

---

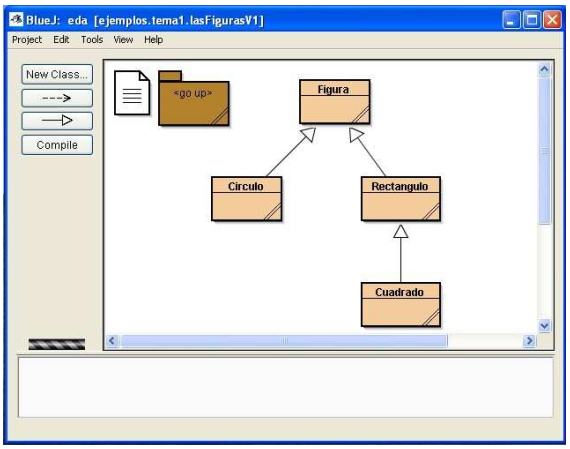
public class **ArrayGrupoDeFiguras**
extends java.lang.Object

Constructor Summary	
ArrayGrupoDeFiguras()	construye un Grupo de Figuras vacío

Method Summary	
double	**area**() obtiene el área total de un Grupo de Figuras sumando las áreas de las talla() Figuras que lo componen
boolean	**eliminar**(ejemplos.tema1.lasFigurasV1.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
int	**indiceDe**(ejemplos.tema1.lasFigurasV1.Figura f) devuelve la posición de la primera aparición de f en un Grupo de Figuras, o devuelve -1 si c no está en el Grupo
void	**insertar**(ejemplos.tema1.lasFigurasV1.Figura f) inserta la Figura f en un Grupo, duplicando la talla actual de éste si fuese necesario
void	**ordenar**() devuelve un array con las Figuras de un Grupo ordenados ascendentemente por área; se ordena por Inserción Directa
ejemplos.tema1.lasFigurasV1.Figura	**recuperar**(int i) devuelve la Figura que ocupa la i-ésima posición del Grupo, o devuelve null si i no es válida, i<0 ó i>=talla()
int	**talla**() consulta la talla de un Grupo de Figuras
ejemplos.tema1.lasFigurasV1.Figura[]	**toArray**() obtiene un array cuyas componentes son las talla() Figuras de un Grupo
java.lang.String	**toString**() obtiene un String con las Figuras de un Grupo en orden de inserción

2.2. Diseño de una Jerarquía de clases

Una vez conocidas las ventajas que supone el uso de una Jerarquía de clases Java y la funcionalidad que interesa exigirle ya se puede presentar su proceso de diseño, esto es los pasos a dar y la sintaxis que se precisa para expresar la Herencia. Para secuenciar convenientemente esta presentación cada nuevo elemento que aparezca en ella se aplicará al diseño de la Jerarquía de Figuras reutilizada en la sección anterior y que, como recuerda la siguiente imagen, se encuentra ubicada en el paquete `ejemplos.tema1.lasFigurasV1`:



2.2.1. Diseño de la clase `Base` de una Jerarquía: uso del modificador de visibilidad `protected` y variables polimórficas

Como indica la primera característica de la Herencia -recuérdese «la Herencia se basa en `B`, la clase `Base` y Raíz de la Jerarquía; de ella se derivará `D` »- el primer paso a dar en el diseño de una Jerarquía de clases Java es definir (especificar e implementar) la clase que representa a la Raíz de la Jerarquía o clase `Base`, pues de ella se derivarán las restantes; así, por ejemplo, el desarrollo en Java de una Jerarquía de Figuras comienza con el diseño de la clase `Figura`, la que representa a su Raíz y a partir de la cual se derivarán las **sub**clases `Circulo` y `Rectangulo` (de la que hereda `Cuadrado`).

En cuanto a funcionalidad o Especificación se refiere, la clase `Figura` necesita definir aquellos métodos que permitan manejar cualquiera de los tipos de Figuras que representa, y no los específicos de cada tipo concreto; al ya disponer de ellas, comparar entre sí las Especificaciones de las clases `Circulo`, `Rectangulo` y `Cuadrado` en ausencia de Herencia permite establecer la siguiente Especificación Factor-Común para `Figura`: la consulta y modificación en su caso del `tipo`, `color` y `centro` específicos que TIENE UNA Figura, tres Datos que se deberán proporcionar a la hora de crear una nueva Figura, bien como parámetros de los constructores de la clase o bien leyéndolos uno a uno del teclado (método `leer`); la comprobación de la igualdad de dos Figuras en base a su tipo (método `equals`); finalmente, la obtención del `String` que representa a una Figura (método `toString`), método que además se incluirá en sus Derivadas para mostrar un ejemplo de Sobrescritura.

Obsérvese que la Especificación realizada no contiene referencia alguna al `radio` de un Círculo o la `base` y `altura` de un Rectángulo, pues una Figura **ni** ES UN Círculo **ni** ES UN Rectángulo. Estas referencias sólo se introducirán al derivar de `Figura` las subclases `Circulo` y `Rectangulo` respectivamente, precisamente para hacer factible la implementación de aquellos

métodos que dan cuenta de la ideosincrasia específica de un Círculo o un Rectángulo; nótese que, a falta de saber definir métodos abstractos, ello obliga a que `area()` y `perimetro()` no puedan ser implementados más que en las Derivadas de `Figura`, si bien en teoría debieran formar parte de la Especificación de `Figura`.

A partir de la Especificación de `Figura` realizada, si sus atributos son UN `String tipo`, UN `String color` y UN `Point2D.Double centro`, la Implementación de la clase sería la siguiente:

```
package ejemplos.tema1.lasFigurasV1;
import java.util.*; import java.awt.geom.*;
public class Figura {
  protected String tipo, color; protected Point2D.Double centro;
  protected static final String TIPO_POR_DEFECTO = "Círculo", COLOR_POR_DEFECTO = "rojo";
  protected static final Point2D.Double CENTRO_POR_DEFECTO = new Point2D.Double();
  public Figura(String tipo, String color, Point2D.Double centro){
    this.tipo = tipo; this.color = color; this.centro = centro;
  }
  public Figura(){ this(TIPO_POR_DEFECTO, COLOR_POR_DEFECTO, CENTRO_POR_DEFECTO); }
  public String getTipo(){ return this.tipo; }
  public String getColor(){ return this.color; }
  public Point2D.Double getCentro(){ return this.centro; }
  public void setColor(String nuevoColor){ this.color = nuevoColor; }
  public void setCentro(Point2D.Double nuevoCentro){ this.centro = nuevoCentro; }
  public String toString(){
    return "Figura de tipo "+tipo+", color "+color+" y centro "+centro.toString();
  }
  public boolean equals(Object x){
    Figura fX = (Figura)x;
    return ( this.tipo.equals(fX.tipo) && this.color.equals(fX.color) );
  }
  public static Figura leer(Scanner teclado){
    // Si se produce un error en la lectura el resultado es un Círculo estándar,
    // la Figura por defecto de la clase
    Figura res = new Circulo();
    System.out.println("Para leer una Figura desde teclado PULSE");
    System.out.println("1 para leer  Círculo");
    System.out.println("2 para leer  Rectángulo");
    System.out.println("3 para leer  Cuadrado");
    int opcion = teclado.nextInt();
    if ( opcion == 1 )      res = Circulo.leer(teclado);
    else if ( opcion == 2 ) res = Rectangulo.leer(teclado);
    else if (opcion == 3)   res = Cuadrado.leer(teclado);
    return res;
  }
}
```

De esta Implementación de `Figura` hay que destacar los dos rasgos básicos de diseño que la hacen Raíz de la Jerarquía y, por tanto, afectan no sólo a las clases Derivadas sino también a las clases que más tarde usen variables de tipo `Figura`: el modificador de visibilidad `protected` que acompaña a sus atributos y la posibilidad de usar en el diseño de sus métodos variables polimórficas, como por ejemplo se hace en `leer` y equals. En lo que sigue se analizan estas características, así como los errores y efectos laterales que provoca su ausencia o uso indebido a la hora de compilar la clase Raíz de una Jerarquía y a la de usar sus variables.

El modificador de visibilidad `protected`

Como regla general, el modificador de visibilidad que acompaña a los atributos de cualquier clase de una Jerarquía (Base o Derivada) es `protected`, privado para cualquier clase que no sea de la Jerarquía ni del mismo paquete. El objetivo que se pretende conseguir introduciendo este modificador de visibilidad para la Herencia es doble: mantener el Principio de Ocultación de la Información al mismo tiempo que se evita la definición y uso de métodos consultores y modificadores que exige `private` para cualquier clase Java de ese y de cualquier otro paquete. Piénsese que, por definición de Herencia, no tendría mucho sentido que las clases Derivadas de una dada Base, como por ejemplo `Circulo` y `Rectangulo` de `Figura`, no vieran los atributos que usan en su propia definición, tanto si se ubican en el mismo paquete que Base como fuera de él; lo mismo ocurre con clases ajenas a la Jerarquía pero del mismo paquete pues guardan una relación más o menos directa con Base, como por ejemplo `ArrayGrupoDeFiguras` que TIENE UN `Figura elArray[]` como atributo. Sin embargo, cualquier clase ajena a una Jerarquía y su paquete que use variables de éstos, como es el caso de `TestArrayGrupoDeFiguras`, no debe ni conocer ni poder manipular la Implementación de las clases de una Jerarquía, sólo su Especificación.

Indicar finalmente que el modificador de visibilidad `protected` también puede acompañar a aquellos métodos de las clases de una Jerarquía que no siendo públicos se reutilizan sistemáticamente dentro de ella.

Cuestión: arguméntense los inconvenientes que desaconsejan en caso de Herencia la definición de modificadores de visibilidad públicos o *friendly*.

Ejemplos de uso de variables polimórficas en el diseño de los métodos de Base

En este punto se proponen una serie de ejercicios y cuestiones sobre el uso de variables polimórficas en los métodos `leer` y `equals` de la clase `Figura` con el fin de repasar la relación entre Polimorfismo y Herencia y los mecanismos de conversión de tipos y selección de métodos que se deben aplicar para garantizar una correcta compilación y ejecución de una clase donde se definen o usan variables polimórficas.

1. Indíquense los identificadores de las variables polimórficas que se usan en los métodos `leer` y `equals` de `Figura`.
2. Se ha diseñado el programa Java que figura a continuación para comprobar que los métodos `leer` y `equals` de `Figura` funcionan sin mayores problemas.

```
package ejemplos.tema1.gestionFigurasV1;
import ejemplos.tema1.lasFigurasV1.*; import java.util.*;
public class TestLeerYEqualsDeFigura {
  public static void main(String args[]){
    Scanner teclado = new Scanner(System.in).useLocale(new Locale("es", "US"));
    Circulo c = Figura.leer(teclado);
    Rectangulo r = Figura.leer(teclado);
    Cuadrado sq = Figura.leer(teclado);
    String s = "UnString"; boolean esIgual = c.equals(s);
    esIgual = c.equals(r); esIgual = c.equals(sq); esIgual = r.equals(sq);
  }
}
```

Señálense las instrucciones de este programa que pueden provocar un error de compilación o ejecución; tras modificarlas para que compile y ejecute correctamente, o eliminarlas si no es posible solventar el error que provocan, trácese la ejecución del programa resultante.

3. Indíquese si sería posible que el método `leer` devolviera un resultado de tipo `Object` y que el parámetro formal del método `equals` fuese por ejemplo de tipo `Figura` en lugar de `Object`; si en algún caso la respuesta es que sí, indíquese si valdría la pena o no hacer el cambio.
4. Tal como ha sido diseñado el método `leer` de `Figura`, ¿es cierto que dicha clase no sufriría modificación alguna al añadir a la Jerarquía una nueva Derivada o modificar alguna de las existentes? ¿Por qué?
5. Suponiendo que `Figura`, `Circulo`, `Rectangulo` y `Cuadrado` **no** forman parte de una Jerarquía sino que simplemente son clases de un mismo paquete, ¿los métodos `leer` y `equals` se podrían mantener en la clase `Figura` tal y como se acaban de presentar?

2.2.2. Diseño de una clase Derivada de Base: fórmulas para la expresión de la Herencia y la Especialización

Una vez concluido el diseño de la clase Base de una Jerarquía ya se pueden derivar de ella las restantes clases que la conforman. En lo que sigue se introducen los pasos y sintaxis necesaria para expresar que una Derivada hereda de la clase Base -uso de la cláusula `extends` y de la referencia `super`- y, además, la especializa -definición de nuevas componentes, obligatoriamente sus métodos constructores, y la Sobrescritura.

Expresión de la Herencia en el diseño de Derivada: cláusula `extends` y referencia `super`

La práctica habitual a la hora de definir Derivada es establecer antes que nada su parte de Herencia. Así, primero se debe indicar en su Implementación que **hereda de Base** utilizando como sigue la cláusula `extends`: `public class D extends B{...}`. Por ejemplo, las siguientes líneas de definición de clase indican de quién heredan `Circulo`, `Rectangulo` y `Cuadrado`:

```
public class Circulo    extends Figura {...}
public class Rectangulo extends Figura {...}
public class Cuadrado   extends Rectangulo {...}
```

A partir de este momento se puede utilizar en Derivada la referencia `super`, que no `this`, para vía operador `.` acceder a cualquier componente no privada de Base; se implementa así el hecho de que Derivada hereda todas las componentes de Base sin volver a definirlas o implementarlas.

Expresión de la Especialización en el diseño de Derivada: constructores y Sobrescritura

Además de las que hereda, en la Implementación de Derivada intervienen forzosamente las componentes que se utilizan para especializar a Base y que, recuérdese, pueden ser de dos tipos: las nuevas componentes que defina Derivada y los métodos heredados de Base que se redefinan por Sobrescritura. Por ejemplo, la siguiente Implementación de `Circulo` especializa a `Figura` utilizando ambos tipos de componentes, como se hace notar en su código mediante comentarios:

```
package ejemplos.tema1.lasFigurasV1;
import java.util.*; import java.awt.geom.*;
public class Circulo extends Figura {
  // Además de tipo, color y centro, un Círculo ES UNA Figura que TIENE UN
  protected double radio; protected static final double RADIO_POR_DEFECTO = 3.0;
  // Sobrescribe toString() de Figura para incluir el radio de un Círculo
  public String toString(){...}
  // Además de los que hereda y sobrescribe de Figura,
  // un Círculo define como métodos propios:
  public Circulo(double radio, String color, Point2D.Double centro){...}
  public Circulo(){...}
  public double getRadio(){ ... }
  public void setRadio(double nuevoRadio){ ... }
  public double area(){ ... }
  public double perimetro(){ ... }
  public static Circulo leer(Scanner teclado){...}
}
```

Comparando esta Implementación de `Circulo` con la presentada en la página 6 de este tema se observan las diferencias que marca la presencia de Herencia en el diseño de una clase Derivada. En primer lugar, en la actual clase `Circulo` han desaparecido aquellas componentes que se heredan de `Figura` y sólo permanecen las que aporta la clase para especializar a `Figura`. En segundo lugar y con respecto a las componentes comunes a ambas Implementaciones, la presencia de Herencia provoca dos modificaciones: los atributos pasan a ser `protected` por si alguna otra clase, del mismo u otro paquete, extendiera más tarde a `Circulo` y los métodos constructores y `toString` de la clase actual modifican su cuerpo, que no su perfil, por los motivos y en la forma que se describe en adelante. En cuanto al resto de métodos mantienen el mismo cuerpo y perfil que en la versión sin Herencia, por lo que no se ha reproducido en la versión actual; a pesar de ello es necesario señalar que los métodos `area`, `perimetro` y `leer` serían firmes candidatos a la Sobrescritura **si** en `Figura` se pudieran definir métodos lo suficientemente abstractos como para que su código, que no su perfil, se pudiese reformular para adaptarlo al que tienen en los métodos homónimos de cada una de sus Derivadas.

Métodos constructores de `D`erivada

Como regla general una clase Derivada debe definir sus propios constructores, como mínimo tantos como tenga la clase Base de la que hereda; para entender la obligatoriedad de esta regla en lo que sigue se explica lo que sucedería si no se cumpliera y los efectos laterales perjudiciales que ello conllevaría.

Si una Derivada no implementa ningún método constructor tiene por defecto, sin necesidad de definirlo, el constructor sin parámetros que Java proporciona a cualquier clase, i.e. un método sin resultado, con el mismo nombre que la clase y que inicializa sus atributos a los valores por defecto del tipo al que pertenecen; específicamente, «`public D(){ super(); }`» es el constructor por defecto de una clase Derivada, donde `super()` es la invocación al constructor sin parámetros de Base. Así, la invocación «`D();`» al constructor por defecto de Derivada inicializará los atributos que ésta hereda de Base y, tras ello, aquí reside el problema, inicializará los atributos propios de

Derivada a sus valores por defecto. Como muestra el siguiente ejemplo, este efecto lateral puede resultar indeseable en ocasiones: cuando no se implementa ningún método constructor en la clase `Circulo` de la Jerarquía de Figuras su constructor por defecto, sin parámetros, es el método «`public Circulo(){ super(); }`», donde `super()` es una invocación al constructor de `Figura` `public Figura(){this(TIPO_POR_DEFECTO, COLOR_POR_DEFECTO, CENTRO_POR_DEFECTO);}`. Por tanto, todos los objetos de tipo `Circulo` que se creen al invocarlo tendrán `Círculo` como `TIPO_POR_DEFECTO`, `rojo` como `COLOR_POR_DEFECTO`, el origen de coordenadas `new Double.Point2D()` como `CENTRO_POR_DEFECTO` y, lo que es peor, el valor por defecto `0.0` de una variable de tipo `double` como `radio`. La gravedad de esta situación se hace aún más patente cuando en lugar de Círculos se están creando Rectángulos o Cuadrados que siempre tienen un `TIPO_POR_DEFECTO` igual a `Círculo`.

Para evitar heredar por despiste el constructor por defecto de la clase `Base` y los efectos laterales que ello puede suponer, como regla general Derivada **siempre** deberá

1. definir explícitamente sus propios constructores, al menos tantos como tenga Base;
2. invocar **obligatoriamente** en la primera línea de cada método constructor al correspondiente de Base mediante el método `super` con los parámetros apropiados, ya que sino se generará automáticamente la llamada `super()`;
3. inicializar cada uno de los atributos propios de la clase como corresponda.

Aplicando esta regla los métodos constructores de la clase `Circulo` deben ser los siguientes:

```
public Circulo(double radio, String color, Point2D.Double centro){
  super("Círculo", color, centro); this.radio = r;
}
public Circulo(){ super(); this.radio = RADIO_POR_DEFECTO; }
```

Ejercicios propuestos:

1. Indíquese lo que ocurriría si se elimina la primera línea de la constructora con parámetros de la clase `Circulo`, i.e. la llamada `super("Círculo", color, centro)`; razónese también si tal llamada se puede sustituir por las tres asignaciones que efectúa.
2. Explíquense los motivos por los que los constructores de la clase `Circulo` ya no pueden ser los que figuran a continuación, los de la versión sin Herencia de la misma clase:

   ```
   public Circulo(double radio, String color,  Point2D.Double centro){
     this.radio = radio; this.color = color; this.centro = centro;
   }
   public Circulo(){ this(RADIO_POR_DEFECTO, COLOR_POR_DEFECTO, CENTRO_POR_DEFECTO); }
   ```

3. Sea `Cuadrado` una Derivada de `Rectangulo` y sean los constructores de `Rectangulo` los que figuran a continuación. Indíquese si la clase `Cuadrado` debe definir como atributos propios `lado` y `LADO_POR_DEFECTO` y, hecho esto, defínanse los constructores, consultores y modificadores de sus atributos.

   ```
   public Rectangulo(double base, double altura, String color, Point2D.Double centro){
     super("Rectangulo", color, centro); this.base = base; this.altura = altura;
   }
   public Rectangulo(){
     this(BASE_POR_DEFECTO,ALTURA_POR_DEFECTO,COLOR_POR_DEFECTO,CENTRO_POR_DEFECTO);
   }
   ```

4. Dada la siguiente declaración de clases:

```
package ejemplos.tema1.lasFigurasV1;
public class Figura {
   public String tipo; protected String color; private Point2D.double centro;
   ...
}
package ejemplos.tema1.lasFigurasV1;
public class Triangulo extends Figura {
   public double base; private double altura;
   ...
}
package ejemplos.tema1.gestionFigurasV1;
 public class Test {
   public static void main(String args[]){
      Figura f = new Figura(); Triangulo t = new Triangulo();
      System.out.println( f.tipo+"\t"+f.color+"\t"+f.centro);
      System.out.println( t.base+"\t"+t.altura);
   }
}
```

Indíquense los accesos incorrectos que se producen desde el `main` de la clase `Test`. ¿Cuáles serían los accesos incorrectos si dicho `main` se definiese dentro de la clase `Figura`? ¿Y si se definiera dentro de la clase `Triangulo`?

Sobrescritura de un método de la clase `Base`

Como ya se ha dicho, **se sobrescribe** cualquier método de `Base` que se defina de nuevo en `Derivada`. Para ello hay que definir en `Derivada` un método que tenga

1. el mismo nombre y lista de parámetros que el método de `Base`, i.e. con su misma signatura; como se estudiará en el próximo tema, para preservar dicha signatura el método de `Derivada` no puede añadir Excepciones a la lista `throws` del método de `Base`.
2. el mismo tipo de resultado que el método de `Base`;
3. un cuerpo distinto total o parcialmente del definido para el método de `Base` según se quiera cambiar completa o parcialmente su significado en `Derivada`; en el caso de **Sobrescritura Parcial** se debe invocar vía `super` como mínimo a uno de los métodos de `Base`.

Por ejemplo, el siguiente método `toString` de `Circulo` sobrescribe totalmente al de `Figura`:

```
public String toString(){
    String res = "Círculo de radio "+String.format(new Locale("US"), "%.2f", this.radio);
    res += ", color "+color+" y centro "+centro.toString();
    return res;
}
```

Nótese que de esta forma el método `toString` resultante reproduce al que se presentó en la versión sin Herencia de la clase `Circulo`. Pero si en lugar de una Sobrescritura Total se realiza una Parcial, i.e. se reutiliza el `toString` de `Figura`, se tendría:

```
public String toString(){
    return super.toString()+" y radio "+String.format(new Locale("US"), "%.2f", this.radio);
}
```

Advertir que en este último caso es obligatorio usar la referencia `super` al invocar al `toString()` de `Figura`, pues así se rompe la ambigüedad generada por la sobrecarga del nombre del método dentro de la clase y, con ello, se evita la definición recursiva e infinita de `toString()` en

`Circulo`. Al hilo de esta advertencia cabe señalar también que dentro de una Jerarquía la única diferencia entre la Sobrescritura y la Sobrecarga del nombre de un método de Base en sus Derivadas, como la de `toString` y la de `leer` respectivamente, es el tipo del resultado del método en cuestión, el mismo en la clase Base que en sus Derivadas si se sobrescribe y distinto del de Base en cada una de ellas si se sobrecarga su nombre. Para ver la repercusión que esta pequeña diferencia puede tener a la hora de diseñar una Jerarquía nótese que mientras la Sobrecarga de `leer` en una Derivada, actual o futura, de `Figura` supone recompilar `Figura` tras cada cambio que se produzca en dicha Derivada, la Sobrescritura de `toString` no. Por tanto, la Sobrecarga de nombres de métodos en una Jerarquía rompe con una de las características que la Herencia imprime a las clases que forman parte de ella, la de que Derivada es una clase completamente nueva cuyos cambios no afectan a Base, y con ello anula las ventajas de mantenimiento y expansión del Software que de ella se desprenden. **En conclusión**, siempre que sea posible no conviene sobrecargar el nombre de un método en un Jerarquía; si se ha hecho en el ejemplo de la Jerarquía de Figuras con el método `leer` ha sido por mostrar que gracias al Polimorfismo de una variable es posible definir ya en la clase Base un método que, al igual que `area` y `perimetro`, siendo común a todas sus Derivadas no podría figurar en ella y al que, por tanto, no se le podría aplicar el Enlace Dinámico.

Ejercicios propuestos:

1. Impleméntense las clases `Rectangulo` y `Cuadrado` de la Jerarquía `Figura` siguiendo el ejemplo de `Circulo`.
2. Las siguientes clases, ubicadas en el mismo paquete, deben ser completadas y mejoradas:

```
public class Circulo {
  private String tipo; double radio;
  public Circulo(double r){ this.radio = r; this.tipo = "Círculo"; }
  public Circulo(double r, String t){ this.radio = r; this.tipo = t; }
  public double area(){ return Math.PI * radio * radio; }
  public double perimetro(){ return 2 * Math.PI * radio; }
  public String toString(){ return "Círculo de radio "+radio; }
}
public class Cilindro extends Circulo {
  private double altura;
  public Cilindro(double radioBase, double altura){...}
  public double area(){ return 2*Math.PI*radio*radio + 2*Math.PI*radio*altura; }
  public double volumen(){ return Math.PI*radio*radio*altura; }
}
```

   Para ello se pide:

   - indicar el modificador de visibilidad que se le debe asignar al atributo `radio` de la clase `Circulo` para conseguir que resulte accesible desde la clase `Cilindro` y, además, favorecer el Principio de Ocultación de la Información;
   - indicar cuál de las siguientes implementaciones del método constructor de la clase `Cilindro` es la **in**correcta y por qué:

     ```
     super(radioBase, "Cilindro"; this.altura = altura;
     super.radio = radioBase; super.tipo = "Cilindro"; this.altura = altura;
     ```

   - las actuales implementaciones de los métodos `area` y `volumen` de la clase `Cilindro` **no** favorecen la Reutilización de Software; substituirlas por las que sí lo hagan;
   - la actual clase `Cilindro` ES UN `Circulo`; redefinirla para que TENGA UN `Circulo` como base.

3. Sean las clases del paquete `elAlmacen` de `ejemplosDelTema1` las siguientes:

```
public class Almacen1 {
 public Almacen1(){ System.out.println("constructor de Almacen1"); }
 public void insertar(Object o){ System.out.println("inserto en Almacen1 "+o); }
 public Almacen2 pasar(){ Almacen2 alm = new Almacen2(); return alm; }
}
public class Almacen2 {
 public  Almacen2(){ System.out.println("constructor de Almacen2"); }
 public void insertar(Object o){ System.out.println("inserto en Almacen2 "+o); }
 public Almacen1 pasar(){ Almacen1 alm = new Almacen1(); return alm; }
}
public class Almacen3 extends Almacen1 {
 public Almacen3(){ System.out.println("constructor de Almacen3"); }
 public void insertar(Object o){ System.out.println("inserto en Almacen3"+o); }
}
public class Almacen4 extends Almacen2 {
 public Almacen4(){ System.out.println("constructor de Almacen4"); }
 public void insertar(Object o){ System.out.println("inserto en Almacen4"+o); }
}
```

Suponiendo ubicado el siguiente programa en el paquete `gestionAlmacen`, compruébese si hay errores de compilación en su método `main`; en caso afirmativo indíquese cuáles son, por qué se producen y corríjanse.

```
public class TestAlmacen {
 public static void main(String args[]){
   Almacen1 alm1 = new Almacen1();
   Almacen2 alm2 = new Almacen4();
   Almacen3 alm3 = new Almacen1();
   Almacen2 alm4 = new Almacen4();
   System.out.print("1.-"); alm1.insertar("objeto1");
   System.out.print("2.-"); alm2.insertar("objeto2");
   System.out.print("3.-"); alm3.insertar("objeto3");
   System.out.print("4.-"); alm4.insertar("objeto4");
   System.out.print("5.-"); alm2.pasar().insertar("objeto5");
   System.out.print("6.-"); ((Almacen1)alm1).pasar().insertar("objeto6");
 }
}
```

4. Sean las siguientes clases del paquete `ejemplos.tema1.losAnimales`:

```
public class Animal {
   public void emitirSonido(){ System.out.println("Grunt!"); }
}
public class Muflon extends Animal {
   public void emitirSonido(){ System.out.println("MOOOO!"); }
   public void alimentarCon(){ System.out.println("Hierba!"); }
}
public class Armadillo extends Animal {}
public class Guepardo extends Animal {
   public void emitirSonido(){ System.out.println("Groar!"); }
}
```

Si el siguiente programa Java se ubica también en el paquete, indíquense las instrucciones de su `main` que provocan error y las que no y explíquese brevemente el motivo.

```
public class Test1Animal {
 public static void main(String[] args){
   adoptar(new Armadillo());
   Object o=new Armadillo(); Armadillo a1=new Animal(); Armadillo a2=new Muflon();
 }
 private static void adoptar(Animal a){ System.out.println("Ven, cachorrito!");}
}
```

5. Trácese el resultado de la ejecución del siguiente programa Java:

```
package ejemplos.tema1.losAnimales;
public class Test2Animal {
 public static void main(String[] args){
   Animal a = new Armadillo(); a.sonido();
   a = new Muflon(); a.emitirSonido();
   a = new Guepardo(); a.emitirSonido();
 }
}
```

   En función del resultado obtenido y siguiendo las reglas de la Herencia ¿qué modificaciones se deberían realizar en la Jerarquía para que todos los Animales emitieran el sonido `Grunt`? ¿Cómo se podría conseguir saber el tipo de alimentación de cada Animal?

6. Se dispone de las siguientes clases en `ejemplos.tema1.losAnimales`:

```
public class Milpies {
   protected int numeroDePies;
   public Milpies(){
      numeroDePies = 1000;
      escribirPies();
   }
   public void escribirPies(){
     System.out.println("Un Milpiés o Cochinilla tiene "+numeroDePies+" pies");
   }
}

public class MilpiesEsquiador extends Milpies {
   protected int numeroDePiesRotos;
   public MilpiesEsquiador(){
      numeroDePiesRotos = 100;
   }
   public void escribirPies(){
      System.out.println("A un Milpiés esquiador le quedan "+
                         (numeroDePies - numeroDePiesRotos)+" pies");
   }
}
public class TestMilpies {
   public static void main(String[] args){
      MilpiesEsquiador m = new MilpiesEsquiador();
   }
}
```

   Indíquese el motivo por el que el resultado de la ejecución del `main` de `TestMilPies` es «`A un Milpiés esquiador le quedan 1000 pies`».

3. Más Herencia en Java: control de la Sobrescritura y regulación de la Herencia

Como se ha observado en el apartado anterior, el diseñador de una Jerarquía puede emplear la Sobrescritura para regular la Herencia al mismo tiempo que potencia todas y cada una de sus ventajas; recuérdese que sólo cuando las Derivadas de la clase Base de una Jerarquía especializan su comportamiento sobrescribiendo métodos heredados de ella, en lugar de definir otros nuevos como propios, se pueden aprovechar al máximo todas las ventajas que el uso de las variables polimórficas comporta. Ahora bien, con los mecanismos estudiados hasta el momento para implementar la Herencia el diseñador no posee más que un control parcial de la Sobrescritura, pues **ni puede impedir que se produzca ni puede forzar u obligar a realizarla**. No puede impedir por ejemplo que métodos de una Jerarquía de Figuras cuyo comportamiento debiera permanecer invariante, como `equals` de `Figura` o `area` de sus Derivadas, sean sobrescritos de forma espuria más tarde; tampoco puede forzar u obligar a sobrescribir aquellos métodos que como `area` y `perimetro` forman parte de la Especificación natural de `Figura` pero ni siquiera es capaz de implementar en ella, sólo en sus Derivadas. Otro tanto sucede con la Jerarquía de Animales propuesta en la última tanda de ejercicios del apartado anterior, como se observa al realizar el número 5: el diseñador no puede garantizar que todos los Animales de un Grupo emitan un mismo sonido porque no puede impedir que el método `emitirSonido` de la clase `Animal` sea sobrescrito al modificar sus actuales Derivadas o al añadir otras nuevas; asimismo tampoco dispone de un mecanismo que le permita implementar en `Animal` un método tan abstracto como `alimentarCon` y, además, imponer a cada una de sus actuales o futuras Derivadas la obligación de sobrescribirlo, por lo que para mostrar el tipo de alimentación de cada uno de los Animales que componen un cierto Grupo no puede más que emplear el siguiente código:

```
public class Test3Animal {
 public static void main(String[] args){
    Animal bestias[] = {new Muflon(), new Guepardo(), new Armadillo()};
    System.out.println("Alimentando a las bestias del grupo:");
    for ( int i = 0 ; i < bestias.length ; i++ )
      if ( bestias[i] instanceof Muflon )        ((Muflon)bestias[i]).alimentarCon();
      else if ( bestias[i] instanceof Guepardo ) ((Guepardo)bestias[i]).alimentarCon();
           else                                  ((Armadillo)bestias[i]).alimentarCon();
 }
}
```

Nótese que si el método `alimentarCon` se pudiera definir en `Animal` y, además, se pudiese forzar su Sobrescritura en sus Derivadas el `main` de `Test3Animal` pasaría a ser el siguiente:

```
Animal bestias[] = {new Muflon(), new Guepardo(), new Armadillo()};
System.out.println("Alimentando a las bestias del grupo:");
for ( int i = 0 ; i < bestias.length ; i++ ) bestias[i].alimentarCon();
```

Pues bien, para conseguir prohibir o imponer la Sobrescritura, y así convertir a la Herencia en un verdadero instrumento de Reutilización del Software, se requieren respectivamente los modificadores Java `final` y `abstract`; a su presentación y estudio están dedicadas las siguientes secciones del apartado.

3.1. Métodos y clases `final`

Si se quiere que un método `f` de la clase Base de una Jerarquía **permanezca invariante** dentro de ella para impedir su redefinición accidental o su uso espurio, i.e. se quiere prohibir su Sobrescritura, dicho método se debe definir como **Final** añadiendo a su cabecera el modificador Java `final`; a partir de ese momento cualquier intento de sobrescribir `f` en una Derivada de Base será penalizado por el intérprete Java con un error de compilación tipo «`f in Derivada cannot override f in Base; overridden method is final`». Sería entonces conveniente, por ejemplo, añadir `final` a la cabecera del método `equals` de `Figura` para conseguir un criterio invariante de comparación entre Figuras de distinto tipo; igualmente deberían ser `final`, como mínimo, los métodos `area` y `perimetro` de `Rectangulo`, pues recuérdese que las fórmulas de cálculo del área y perímetro de un Rectángulo de base y altura iguales son las de un `Cuadrado`. También en el caso de la Jerarquía de Animales el uso de `final` permitiría impedir que el método `emitirSonido` de la clase `Animal` fuese sobrescrito en sus Derivadas; como se muestra a continuación, bastaría con añadir a su cabecera el modificador `final` y eliminar los métodos que con el mismo perfil se hubieran definido en sus Derivadas:

```
public class Animal {
  public final void emitirSonido(){ System.out.println("Grunt!"); }
}
public class Muflon extends Animal {
  public void alimentarCon(){ System.out.println("Hierba!"); }
}
public class Armadillo extends Animal {
  public void alimentarCon(){ System.out.println("Insectos!");}
}
public class Guepardo extends Animal {
  public void alimentarCon(){ System.out.println("Carne Roja!");}
}
```

Si se añadiese a esta Jerarquía una clase `Milpies` que incluyera su propio método `emitirSonido()` el intérprete Java lo advertiría y provocaría un error de compilación «`emitirSonido() in Milpies cannot override emitirSonido( ) in Animal; overridden method is final`»; como ilustra el siguiente programa, se podría emplear el mismo sistema para obligar a que todos los Animales de un Grupo emitan un mismo sonido `Grunt!` durante un (hipotético) concierto:

```
public class Test4Animal {
 public static void main(String[] args){
    Animal bestias[] = {new Muflon(), new Guepardo(), new Armadillo()};
    System.out.println("Concierto de las bestias del grupo:");
    for ( int i = 0 ; i < bestias.length ; i++ ) bestias[i].emitirSonido();
 }
}
```

Cuestión: tras eliminar `emitirSonido()` de la clase `Milpies`, la Base de `MilpiesEsquiador`, indíquese si el uso de `final` conseguiría cambiar el actual resultado de `TestMilpies`.

Además de métodos cuya Sobrescritura se prohíba, deben ser `final` los atributos que representen constantes, como sucede con `PI` en la clase `Math`, y también las clases Java cuyas componentes sean todas `final` por representar valores y operaciones que no se puede modificar vía Herencia, como sucede con cualquiera de las clases Envoltorio de los tipos primitivos de Java (`Integer`, `Boolean`, `Character`, `Double`, etc.) ubicadas en `java.lang` - obsérvese en la documentación de cualquiera de ellas que una clase `final` se ubica como una Hoja del Árbol

de Herencia Java para expresar que no puede ser extendida o reutilizada vía Herencia y que, ya a nivel puramente sintáctico, basta poner el modificador `final` en su cabecera de definición para que automáticamente todas sus componentes sean definidas como finales.

Ya para concluir la sección resta comentar que el uso de componentes `final` no sólo evita su redefinición accidental sino que también permite generar código más eficiente puesto que el intérprete Java resuelve su función asignada en tiempo de compilación (estáticamente), al igual que con una componente `static`, y no en tiempo de ejecución (dinámicamente).

3.2. Métodos y clases `abstract`

Si se quiere **imponer la especialización** de un método `f` de la clase Base de una Jerarquía vía Herencia para garantizar su aplicación *ad-hoc* mediante Enlace Dinámico, i.e. se quiere forzar su Sobrescritura, dicho método se debe definir **Abstracto** tal y como se indica ahora:

1. **En la implementación de la clase Base** se define tan sólo el perfil de `f` entre el modificador `abstract` y un simple punto y coma, en lugar del habitual bloque de instrucciones delimitado por llaves; por ejemplo «`public abstract double area();`» sería la definición de `area()` como método abstracto de `Figura`.
2. **En la cabecera de definición de la clase Base** se añade obligatoriamente el modificador `abstract`, pues una clase que define al menos un método abstracto es también abstracta; por ejemplo «`public abstract class Figura {...}`» sería la cabecera de la clase `Figura` abstracta.
3. **En cada una de las clases Derivadas de Base** se sobrescribe `f`, salvo si ésta es `abstract` también; es más, no sobrescribir `f` en una Derivada no abstracta de Base provoca un error de compilación. Por ejemplo, si en la clase `Circulo` no se sobrescribe `area()` el intérprete Java lo advierte con el error «`Circulo is not abstract and does not override abstract method area() in Figura`».

Para presentar las distintas situaciones en las que resulta beneficioso el uso de `abstract` conviene retomar el ejemplo de la Jerarquía de Figuras y explicar los motivos por los que conviene definir como abstractos en su clase Base `Figura`, que por ello pasa a ser abstracta, los métodos `leer`, `area` y `perimetro`. A saber:

- definir como abstracto el método `leer` de `Figura` permite separar por completo su especificación de su implementación y, por tanto, permite solventar los problemas que causa la Sobrecarga de su nombre dentro de la Jerarquía;
- definir como abstractos en `Figura` los métodos `area` y `perimetro` es la única vía para permitir que estos métodos se apliquen mediante Enlace Dinámico a las variables polimórficas de tipo `Figura`. Así por ejemplo, la combinación de Polimorfismo y Enlace Dinámico hace posible que el método abstracto `area` se pueda usar para implementar el método `equals` de la clase `Figura` como sigue:

```
/** indica si una Figura es igual a x, i.e. si tiene su mismo tipo, color y área */
public final boolean equals(Object x){
  Figura fX = (Figura)x;
  return ( tipo.equals(fX.tipo) && color.equals(fX.color) && area() == fX.area() );
}
```

- definir como abstractos en `Figura` los métodos `leer`, `area` y `perimetro` siempre proporciona la ventaja adicional de su Sobrescritura forzosa porque, como ya se ha comentado e ilustrado antes, el intérprete Java recuerda con un error de compilación al programador que lo olvida cuáles son los métodos que se deben sobrescribir en una Jerarquía.

Ejercicios propuestos:

1. Suponiendo que el método `leer` se declara como abstracto en `Figura`, ¿por qué no puede seguir siendo estático? ¿Podría entonces definirse en `Figura` el siguiente un método `leer`?

   ```
   /** modifica las componentes de una Figura leyendo las nuevas desde teclado **/
   public abstract void leer(Scanner teclado);
   ```

   Utilizando este nuevo perfil, ¿que modificaciones sufrirían los métodos `leer` de `Circulo`, `Rectangulo` y `Cuadrado`.

2. Indíquense las líneas del programa `TestJerararquiaFigura` que se simplifican como consecuencia directa de declarar `abstract` el método `area()` en `Figura`.

3. Explicítense los motivos por los que el Polimorfismo y el Enlace Dinámico hacen posible usar el método abstracto `area` en la implementación del método `equals` de `Figura`. Para ello puede resultar de ayuda trazar cual intérprete Java la ejecución del siguiente código:

   ```
   Figura f1 = new Circulo(); Figura f2 = new Cuadrado();
   boolean sonIguales = f1.equals(f2);
   ```

4. Créese el paquete `ejemplos.tema1.lasFigurasV2` y cópiense en él las clases de `lasFigurasV1`. Actualícese entonces su clase `Figura` para que contenga los métodos abstractos `area`, `perimetro` y `leer`, la definición del método `equals` que usa `area` y añádanse también todos los modificadores `final` que sean oportunos; al concluir con `Figura`, actualícense convenientemente las restantes clases del paquete. Compruébese entonces si se puede `insertar` el objeto `new Figura()` en un `ArrayGrupoDeFiguras`; si no fuese posible indíquese la utilidad que tiene definir los métodos constructores de la clase `Figura`. Ya para concluir, complétese y ejecútese el siguiente programa:

```
package ejemplos.tema1.gestionFigurasV2;
import ejemplos.tema1.lasFigurasV2.*; import java.util.*; import java.text.*;
public class TestArrayGrupoDeFiguras {
 private static Scanner teclado = new Scanner(System.in).useLocale(new Locale("es","US"));
 public static void main(String args[]){
        ArrayGrupoDeFiguras g = new ArrayGrupoDeFiguras();
        int opcion;
        do { opcion = menu(); if ( opcion != 0 ) procesar(opcion, g);
           } while ( opcion != 0 );
 }
 private static int menu(){
        int res = 0; System.out.println();
        System.out.println("    ***  MENÚ  ***");
        System.out.println("1.- Insertar una nueva Figura en el Grupo");
        System.out.println("2.- Buscar una Figura en el Grupo");
        System.out.println("3.- Eliminar una Figura del Grupo");
        System.out.println("4.- Mostrar las Figuras que componen el Grupo");
        System.out.println("5.- Calcular el área del Grupo");
        System.out.println("6.- Ordenar el Grupo por área");
        System.out.println("0 - Salir");
        System.out.print("Seleccione opción: "); res = teclado.nextInt();
        return res;
 }
```

```
private static void procesar(int opcion, ArrayGrupoDeFiguras g){
       switch (opcion){
           case 1: Figura f = getFiguraALeer();
                   f.leer(teclado); g.insertar(f); break;
           /** COMPLETAR **/
}
private static Figura getFiguraALeer(){
       Figura res = new Circulo();
       System.out.println("Para leer una Figura desde teclado PULSE:");
       System.out.println("1 para leer  Círculo");
       /** COMPLETAR **/
}
}
```

5. Modifíquese la Jerarquía de Animales actual para garantizar que cada Derivada de `Animal` defina el método `alimentarCon()`, y con ello conocer el tipo de alimentación de cada Animal. Añádase después a la Jerarquía la clase `Milpies` que se presentó en la última tanda de ejercicios del apartado anterior y si su compilación presenta algún problema, soluciónese.

3.3. Clases `interface` y Herencia Múltiple

Una (super)clase Java cuyos métodos son todos abstractos recibe el nombre específico de `interface` pues, por definición de `abstract`, únicamente especifica el comportamiento o funcionalidad de una clase pero no lo implementa; en otras palabras, actúa como una **Interfaz** (*Interface* en Inglés) al describir qué hace una clase (Especificación) pero no cómo lo hace (Implementación). Además del nombre que recibe, la definición de `abstract` imprime tres características más a una clase de este tipo: sus atributos sólo pueden ser públicos y finales, no posee métodos constructores al carecer de estructura y, finalmente, sus métodos son públicos y han de ser implementados obligatoriamente en cualquiera de sus Derivadas -por lo que se dice que éstas implementan la `interface` en lugar de extenderla.

Vistas sus características, los pasos para definir una clase `interface I` en Java, y para indicar que otra clase `D` la implementa, son los siguientes:

1. En la cabecera de `I` se debe substituir la palabra `class` por la palabra `interface`.
2. En la cabecera de cada método de `I` **no** se deben escribir los modificadores `public` y `abstract` pues lo son por definición de `interface`.

   Así por ejemplo, suponiendo que en la `interface I` se define un único método con perfil `TipoRes f(TipoArg1 arg1, ..., TipoArgN argN) throws ExcepcionCheked` su definición en Java sería como sigue:

   ```
   public interface I {
      TipoRes f(TipoArg1 arg1, ... , TipoArgN argN) throws ExcepcionCheked;
   }
   ```

3. En la cabecera de la clase `D` que implementa `I` se debe incluir `implements I` antes de la llave abierta con la que empieza su cuerpo; además, en el caso en el que la clase `D` extienda a `B` la palabra `implements` irá detrás de `B` (separada de él por un blanco) y en el caso en el que `D` implemente varias `interface` sus nombres aparecen separados por comas. Por ejemplo, para expresar que la clase `Circulo` implementa las `interface I`, `H` y `K`, su cabecera de definición sería la siguiente:

```
public class Circulo extends Figura implements I,H, K { ... }
```

Es muy importante señalar ahora que gracias a que una clase puede implementar tantas interfaces como sea necesario, la `interface` es el mecanismo con el que Java implementa la **Herencia Múltiple**, esto es el hecho de que una clase `Derivada` herede de más de una clase `Base` distinta de `Object`.

4. En el cuerpo de la clase `D` que implementa `I` se deben sobrescribir obligatoriamente todos los métodos definidos en `I`, salvo si `D` es una clase `abstract` o `interface`, pues de lo contrario el intérprete Java lo advierte al ser abstractos los métodos de `I`.

Aunque a lo largo del curso las clases `interface` se utilizarán de forma sistemática cuando sea necesario que una clase herede de más de una clase distinta de `Object` y/o cuando sea necesario imponer a una clase una funcionalidad que no posee `Object`, para poder ofrecer ejemplos verdaderamente ilustrativos de estas situaciones habrá que esperar a introducir la Genericidad y la Representación Enlazada de un Grupo genérico de Datos.