Rev 1.4 20160407

Programación II

Tema 5. Composición y herencia

Contenido

1 Introducción

En este tema se tratan la composición y la herencia. Ambos conceptos son vitales para la programación mediante el paradigma de orientación a objetos. Si bien el primero ya se ha estado utilizando en ocasiones, el segundo es completamente nuevo.

2 Composición

La composición consiste en que un objeto forme parte de otro. Para ello, es necesario que una de las clases (la clase contenida), aparezca como atributo en otra (la clase contenedora). Desde un punto de vista de diseño, la relación entre ellas es del tipo “es parte de”. Por ejemplo, la clase **Punto** es parte de la clase **Línea** (o dicho de otra manera, una línea está formada por dos puntos, el de comienzo y el de final).

Ejemplo

```
public class Punto {
	private int x;
	private int y;

	public Punto(int a, int b)
	{
		x = a;
		y = b;
	}

	public int getX()
	{
		return x;
	}

	public int getY()
	{
		return y;
	}

	public String toString()
	{
		return "(" + getX() + ", " + getY() + ")";
	}
}
```

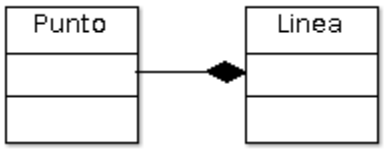
```
public class Linea {
	private Punto org;
	private Punto fin;

	public Linea(int x1, int y1, int x2, int y2)
	{
		this( new Punto( x1, y1 ), new Punto( x2, y2 ) );
	}

	public Linea(Punto i, Punto f)
	{
		org = i;
		fin = f;
	}

	public Punto getOrg()
	{
		return org;
	}

	public Punto getFin()
	{
		return fin;
	}

	public String toString()
	{
		return "(" + getOrg() + ", " + getFin() + ")";
	}

}

public class Ppal {
	public static void main(String[] args)
	{
		Linea l1 = new Linea( 1, 1, 2, 2 );
		Linea l2 = new Linea( new Punto( 4, 5 ), new Punto( 6, 7 ) );

		System.out.println( l1 );
		System.out.println( l2 );
	}
}
```

Salida

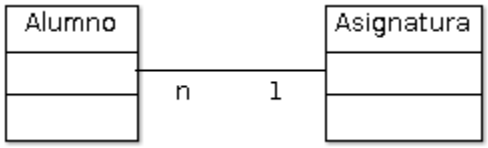
```
((1, 1), (2, 2))
((4, 5), (6, 7))
```

A la hora de presentar la relación en un diagrama de clases UML, ambas clases se unen por una línea, de tal forma que en el extremo de la clase contenedora aparece un rombo relleno. Esta es la relación más fuerte de composición, de forma que el objeto contenido se crea cuando el objeto contenedor es creado, y se destruye cuando el objeto contenedor se destruye: la existencia del objeto contenido no

tiene sentido sin el objeto contenedor.



Es posible que dos objetos colaboren entre ellos, pero que su existencia no sea dependiente el uno del otro. Por ejemplo, un objeto **Alumno** puede estar relacionado con un objeto **Asignatura**, pero el hecho de que la **Asignatura** desaparezca no implica que el alumno desaparezca a su vez, ni viceversa. Esto es composicion debil o asociacion, y se representa por una linea entre las clases sin flechas ni rombos en los extremos. En ocasiones, se indica la cardinalidad del tipo uno a uno (1, 1), uno a muchos (1, n), o muchos a muchos (n, n), cuando es significativo.



Ejemplo

```
public class Alumno {
	private String nombre;

	public Alumno(String n)
	{
		nombre = n;
	}

	public String getNombre()
	{
		return nombre;
	}

	public String toString()
	{
		return getNombre();
	}
}
```

```
public class Asignatura {
	private Alumno[] v;
	private String nombre;

	public Asignatura(String n)
	{
		nombre = n;
		v = new Alumno[ 0 ];
	}

	public void inserta(Alumno a)
	{
		final int numAlumnos = v.length;
		Alumno[] v2 = new Alumno[ numAlumnos + 1 ];

		for(int i = 0; i < numAlumnos; ++i) {
			v2[ i ] = v[ i ];
		}

		v2[ numAlumnos ] = a;
		v = v2;
	}

	public String getNombre()
	{
		return nombre;
	}

	public String toString()
	{
		StringBuilder toret = new StringBuilder();

		toret.append( getNombre() );
		toret.append( '\n' );

		for(int i = 0; i < v.length; ++i) {
			toret.append( v[ i ].toString() );
			toret.append( '\n' );
		}

		return toret.toString();
	}
}
```

```
public class Ppal {
      public static void main(String[] args)
      {
            Alumno al1 = new Alumno( “Baltasar” );
            Alumno al2 = new Alumno( “Pedro” );
            Alumno al3 = new Alumno( “Nanny” );
            Asignatura asg1 = new Asignatura( “PRO2” );
            Asignatura asg2 = new Asignatura( “ALS” );

            asg1.inserta( al1 );
            asg1.inserta( al2 );
            asg1.inserta( al3 );

            asg2.inserta( al1 );

            System.out.println( asg1 );
            System.out.println( asg2 );
      }
}
```

Salida

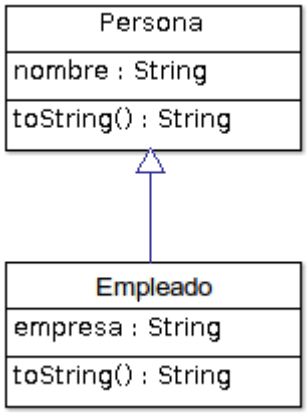
```
PRO2
Baltasar
Pedro
Nanny

ALS
Baltasar
```

3 Herencia

La herencia consiste, desde una perspectiva muy básica, en crear nuevas clases a partir de clases que ya existen previamente. Además, existe una relación “es un tipo de” entre la clase que hereda y la clase de la que se hereda. Así, la herencia tiene esa doble vertiente, como mecanismo de reutilización de código, y como método de clasificación.

En cuanto a cuestiones de notación, la clase de la que se hereda se la conoce en Java como superclase, mientras la clase que hereda se conoce como subclase. Sin embargo, existen muchas otras notaciones: por ejemplo, clase base, y clase derivada; o clase padre y clase hija, respectivamente.

En un diagrama de clases de UML, la superclase se sitúa más arriba de la subclase, y de esta última a la primera se dibuja una línea con punta de flecha. La particularidad es que la punta de flecha está cerrada (es un triángulo) y vacía, es decir, en blanco. Se incluyen dos diagramas de ejemplo, el primero con las clases **Persona** y **Empleado**. En este primero, **Empleado** es una subclase de **Persona**, lo que permite a la primera recibir el atributo nombre, (aunque inaccesible por ser privado), y el método *getNombre()*.



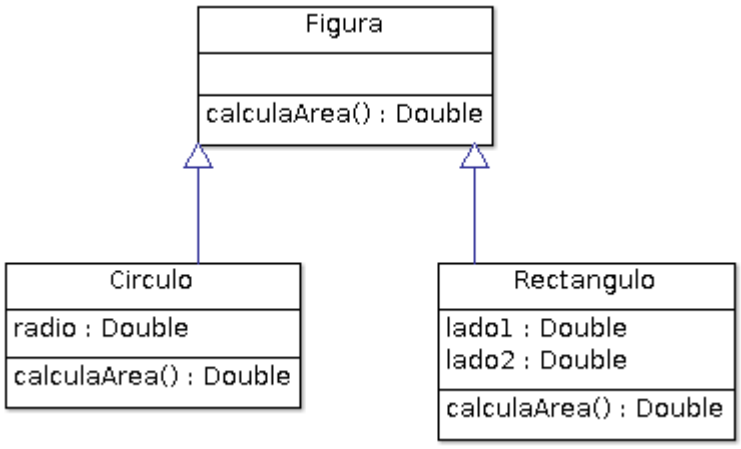
En el segundo diagrama de ejemplo, se muestran varias clases relacionadas por herencia: se trata de la clase **Figura**, que es la superclase de **Circulo** y **Rectangulo**. Nótese que el **Círculo** "es un tipo" de **Figura**, como lo es a su vez el **Rectangulo**. Esto nos permite concluir que la relación de herencia entre clases es correcta. El motivo principal de utilizar herencia es la de a) clasificar los círculos y rectángulos como figuras, y b) establecer el método *calculaArea()* como un método que deben poseer todas las subclases. Si la interfaz es el conjunto de métodos públicos de una clase, en este caso se está definiendo una interfaz mínima (el propio método *calculaArea()*) que se sabe que todos las subclases de **Figura** poseen.

Existe otro método parecido a **Figura**.*calculaArea()*, que se ha estado utilizando continuamente: se trata de *toString()*. Cada vez que se crea una clase, es muy útil crear un método *toString()* para ella, pero siguiendo el razonamiento con **Figura**, este método debe estar definido en una superclase. Efectivamente, existe una superclase común a todas las clases que se puedan crear o utilizar en Java, que es **Object**. Así, es indiferente crear la clase **Punto** indicando que hereda de **Object** o no, pues por defecto, a no ser que se indique lo contrario, todas lo hacen. Por tanto, la superclase de cualquier clase en Java siempre será **Object**, que, entre otros métodos, define *toString()*, lo que le permite aplicarlo a cualquier objeto.

Nótese que Java solo soporta la herencia simple, es decir, una clase dada solamente puede heredar de otra, o lo que es lo mismo, tras *extends* puede aparecer justamente el nombre de una clase, nunca más.

Sintaxis

```
// Forma 1
class Persona {
	// ...
}

// Forma 2 (equivalente a la forma 1, no hace falta)
class Persona extends Object {
	// ...
}
```



Si en un diagrama se colocan las superclases en la parte superior, y las subclases a continuación, más abajo, tendremos que a medida que se asciende en el diagrama se generaliza, mientras que a medida que se desciende en el diagrama se especializa. Estos diagramas están mostrando una jerarquía de herencia.

Sintaxis

```
class Superclase {
	// ...
}
class Subclase extends Superclase {
	// ...
}
```

Ejemplo

```
public class Persona {
	private String nombre;

	public Persona(String n)
	{
		nombre = n;
	}

	public String getNombre() {
		return nombre;
	}

	public String toString()
	{
		return getNombre();
	}
}
```

```
public class Empleado extends Persona {
	private String empresa;

	public Empleado(String n, String e)
	{
		super( n );
		empresa = e;
	}

	public String getEmpresa()
	{
		return empresa;
	}

	public String toString()
	{
		return getNombre() + ": " + getEmpresa();
	}
}

public class Ppal {
	public static void main(String[] args)
	{
		Empleado e1 = new Empleado( "Baltasar", "Uvigo" );

		System.out.println( e1 );
	}
}
```

Salida

```
Baltasar: Uvigo
```

Ejemplo

```
public class Figura {
	public double calculaArea()
	{
		return 0.0;
	}
}

public class Circulo extends Figura {
	private double radio;

	public Circulo(double r) {
		radio = r;
	}

	public double getRadio() {
		return radio;
	}
```

```
	public double calculaArea() {
		return radio * radio * Math.PI;
	}

	public String toString()
	{
		return String.format( "Circulo de radio %.2f", getRadio() );
	}
}

public class Rectangulo extends Figura {
	private double lado1;
	private double lado2;

	public Rectangulo(double l1, double l2)
	{
		lado1 = l1;
		lado2 = l2;
	}

	public double getLado1() {
		return lado1;
	}

	public double getLado2() {
		return lado2;
	}


	public double calculaArea()
	{
		return getLado1() * getLado2();
	}

	public String toString()
	{
		return String.format( "Rectangulo de lados %.2f y %.2f",
				getLado1(), getLado2() );
	}
}

public class Ppal {
	public static void main(String[] args)
	{
		Rectangulo r1 = new Rectangulo( 5, 6 );
		Circulo c1 = new Circulo( 1 );

		System.out.format( "%s, superficie: %.2f\n", r1, r1.calculaArea() );
		System.out.format( "%s, superficie: %.2f\n", c1, c1.calculaArea() );
	}
}
```

Salida

```
Rectangulo de lados 5.00 y 6.00, superficie: 30.00
Circulo de radio 1.00, superficie: 3.14
```

Al comienzo, el programador principiante suele abusar de la herencia en detrimento de la composición. Se han presentado, sin embargo, dos ejemplos en los que se obtienen beneficios ligeramente distintos, y en los que siempre está presente el uso de la herencia como clasificación.

La herencia del *nombre* de la **Persona**, como ayuda para construir al **Empleado** es un ejemplo bastante típico de reutilización. La infraestructura relacionada con el *nombre* no es necesario volver a crearla, pues ya está presente en la superclase y se puede utilizar directamente desde la subclase. Sin embargo, si se pone demasiado énfasis en la reutilización, se corre el peligro en caer en el uso de la *herencia por conveniencia*, o *herencia de implementación*, en la que se favorece crear relaciones de herencia entre clases solo por el beneficio de recibir código ya existente, indiferentemente de si se puede establecer una relación “es un tipo de” entre subclase y superclase. Esto acaba resintieno la calidad del código creado.

3.1 Comportamiento de los constructores con herencia

Cuando la herencia está presente como relación entre dos clases, los constructores se ejecutan en un orden determinado: aunque evidentemente se comienza por el constructor de la subclase, este se detiene de inmediato hasta que los constructores de las superclases se han ejecutado.

Ejemplo

```
class A {
      public A()
      {
            System.out.println( “Constructor de A” );
      }
}

class B extends A {
      public B()
      {
            System.out.println( “Constructor de B” );
      }
}

public class Ppal {
      public static void main(String[] args)
      {
            B objB = new B();
      }
}
```

Salida

```
Constructor de A
Constructor de B
```

En el ejemplo anterior, al ejecutar el constructor de **B** primero se ejecuta el constructor de **A**, tal y como se puede apreciar en el ejemplo. Cuando el constructor no tiene parámetros, no es necesario especificar que esto suceda, es algo automático. Cuando sí tiene parámetros, es necesario invocar al constructor de la clase base explícitamente, utilizando como primera instrucción, la palabra clave *super* seguida de los parámetros necesarios. Retomando la clase **Empleado** anterior, se puede observar la necesidad de *super()*.

Sintaxis

```
class Superclase {
	private int x;

	public Superclase(int a)
	{
		x = a;
	}

	// ...
}

class Subclase extends Superclase {
	private int y;

	public Subclase(int a, int b)
	{
		super( a );
		y = b;
	}

	// ...
}
```

Ejemplo

```
public class Persona {
	private String nombre;

	public Persona(String n)
	{
		nombre = n;
	}

	//...
}
```

```
public class Empleado extends Persona {
	private String empresa;

	public Empleado(String n, String e)
	{
		super( n );
		empresa = e;
	}
	// ...
}
```

3.2 Reescritura de métodos

Como se puede apreciar en el ejemplo de las figuras, el método *calculaArea()* se define en **Figura** (lo que hace que esté disponible en sus subclases), pero en esta clase no tiene una funcionalidad demasiado útil. Es más, el método *calculaArea()* de **Figura** nunca llega a ser ejecutado (se ejecuta *calculaArea()* en **Círculo** o en **Rectangulo**, en su lugar). Así, los métodos de las superclases son reescritos (*override*, o anulados, en inglés), en favor de los métodos de las subclases.

Más aún, es posible utilizar una notación que ayuda al programador y al compilador a comprobar si está reescribiendo los métodos adecuadamente. Un ejemplo claro sería *calculaArea()* en **Figura**, o, de forma más genérica, *toString()* en **Object**. Esta notación es el decorador *Override*, que se precede de una arroba y se coloca antes del método en cuestión. Como se ve en el ejemplo siguiente, el mismo de las figuras, se coloca antes de *calculaArea()* y *toString()*. La ventaja de utilizar el decorador *Override* es que, en caso de confundir el tipo de retorno o los parámetros del método, el compilador avisa de que no se está reescribiendo el método en la superclase, sino creando un método nuevo. El beneficio se produce en tiempo de compilación, por lo que no varía en nada el comportamiento, el rendimiento, o la salida del programa.

Ejemplo

```
public class Figura {
	public double calculaArea()
	{
		return 0.0;
	}
}

public class Circulo extends Figura {
	private double radio;

	public Circulo(double r) {
		radio = r;
	}

	public double getRadio() {
		return radio;
	}
```

```
	@Override
	public double calculaArea() {
		return radio * radio * Math.PI;
	}

	@Override
	public String toString() {
		return String.format( "Circulo de radio %.2f", getRadio() );
	}
}

public class Rectangulo extends Figura {
	private double lado1;
	private double lado2;

	public Rectangulo(double l1, double l2) {
		lado1 = l1;
		lado2 = l2;
	}

	public double getLado1() {
		return lado1;
	}

	public double getLado2() {
		return lado2;
	}

	@Override
	public double calculaArea()
	{
		return getLado1() * getLado2();
	}

	@Override
	public String toString()
	{
		return String.format( "Rectangulo de lados %.2f y %.2f",
				getLado1(), getLado2() );
	}
}

public class Ppal {
	public static void main(String[] args)
	{
		Rectangulo r1 = new Rectangulo( 5, 6 );
		Circulo c1 = new Circulo( 1 );

		System.out.format( "%s, superficie: %.2f\n", r1, r1.calculaArea() );
		System.out.format( "%s, superficie: %.2f\n", c1, c1.calculaArea() );
	}
}
```

Salida

```
Rectangulo de lados 5.00 y 6.00, superficie: 30.00
Circulo de radio 1.00, superficie: 3.14
```

3.3 Acceso a métodos en la superclase

En ocasiones, un método reescrito en una subclase realiza una tarea que involucra lo que ya hacía el método reescrito en la subclase, lo que conllevaría repetir código. Por ejemplo, en el siguiente código **Empleado**.*toString()* repite código de **Persona**.*toString()*.

Ejemplo

```
public class Persona {
	private String nombre;

	public Persona(String n)
	{
		nombre = n;
	}

	public String getNombre() {
		return nombre;
	}

	@Override
	public String toString()
	{
		return getNombre();
	}
}

public class Empleado extends Persona {
	private String empresa;

	public Empleado(String n, String e)
	{
		super( n );
		empresa = e;
	}

	public String getEmpresa() {
		return empresa;
	}

	@Override
	public String toString()
	{
		return getNombre() + ": " + getEmpresa();
	}
}
```

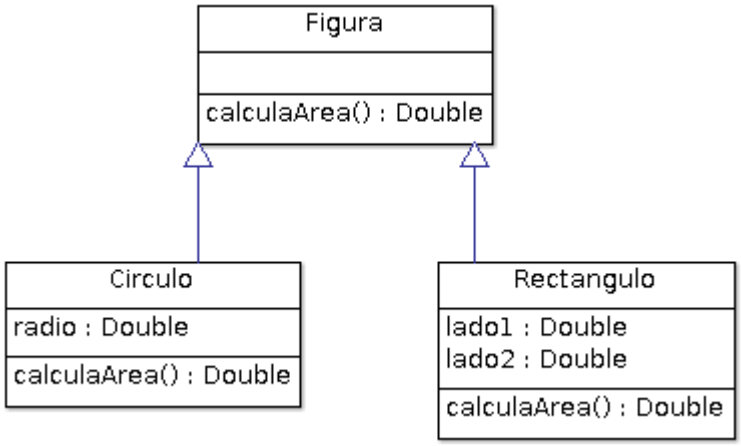
De una forma más o menos similar a como se utiliza *super* en los constructores, es posible llamar a métodos en la superclase. En esta ocasión, *super* es una referencia, que apunta a una versión del objeto referenciado por *this*, que es del tipo de la superclase.

Ejemplo

```
public class Persona {
	private String nombre;

	public Persona(String n)
	{
		nombre = n;
	}

	public String getNombre() {
		return nombre;
	}

	@Override
	public String toString()
	{
		return getNombre();
	}
}

public class Empleado extends Persona {
	private String empresa;

	public Empleado(String n, String e)
	{
		super( n );
		empresa = e;
	}

	public String getEmpresa() {
		return empresa;
	}

	@Override
	public String toString()
	{
		return super.toString() + ": " + getEmpresa();
	}
}
```

3.4 Conversión a la superclase o *upcasting*

Retomando el ejemplo de las figuras, se puede observar que siempre es seguro tratar un objeto de una subclase como si fuera de una superclase. Así, como se observa en el diagrama, un **Rectangulo** o un **Circulo** puede ser tratado sin problema como una **Figura**.



Así, por ejemplo, normalmente se asigna un objeto a una referencia de la misma clase de ese objeto, pero no tiene por qué ser necesariamente así. Es posible, en el ejemplo de las figuras, asignar un **Rectángulo** o un **Círculo** a una referencia de la clase **Figura**. Así, el método *main()* de la clase principal podría reescribirse como aparece a continuación.

Ejemplo

```
// ...

public class Ppal {
      public static void main(String[] args)
      {
            Figura r1 = new Rectangulo( 5, 6 );
            Figura c1 = new Circulo( 1 );

            System.out.format( "%s, superficie: %.2f\n", r1, r1.calculaArea() );
            System.out.format( "%s, superficie: %.2f\n", c1, c1.calculaArea() );
      }
}
```

Salida

```
Rectangulo de lados 5.00 y 6.00, superficie: 30.00
Circulo de radio 1.00, superficie: 3.14
```

Sería lógico asumir que la salida de este ejemplo, en lugar de ser la correcta (la que se muestra más arriba) fuese algo parecida a la que aparece más abajo. Y sin embargo, el programa sigue comportándose como lo hacía anteriormente.

Salida[1]

```
Rectangulo, superficie: 0.00
Circulo, superficie: 0.00
```

Sería razonable, como se explicitaba más arriba, asumir que una referencia a **Figura** solamente tenga accesos a la parte de dicha clase dentro de un **Rectangulo** o un **Circulo** (en concreto, a un *calculaArea()* que solo es capaz de devolver 0).

La razón de que el programa siga funcionando se debe a que Java siempre elige la llamada a los métodos adecuados, sean de la subclase que sean, incluso cuando son llamados desde una referencia a la superclase. Esta característica es llamada enlace tardío, que se discutirá en profundidad más adelante.

Una de las ventajas es que se puede pasar a un método que acepte referencias a **Figura** objetos de **Círculo** o de **Rectángulo**, o que puedan crearse vectores primitivos de **Figura**, con elementos que en realidad son objetos de sus subclases. Se muestra un ejemplo a continuación.

Ejemplo

```
// ...

public class Ppal {
      public static void main(String[] args)
      {
            Figura[] figuras = new Figura[ 2 ];
            figura[ 0 ] = new Rectangulo( 5, 6 );
            figura[ 1 ] = new Circulo( 1 );

            for(int i = 0; i < figuras.length; ++i) {
                Figura f = figuras[ i ];
                System.out.format( "%s, superficie: %.2f\n", f, f.calculaArea() );
            }

            return;
      }
}
```

Salida

```
Rectangulo de lados 5.00 y 6.00, superficie: 30.00
Circulo de radio 1.00, superficie: 3.14
```

1 Esta salida es totalmente inventada, con el propósito de clarificar la explicación.

3.5 Conversión a la subclase o *downcasting*

Si bien siempre es seguro tratar de apuntar a una subclase con una referencia a la superclase, lo contrario no lo es. Puede ser útil en ciertas situaciones comprobar a qué clase pertenece el objeto al que se apunta con una referencia a la superclase, aunque hacer esto continuamente es sin duda un error de diseño en el programa.

El operador *instanceof* devuelve *true* si, al ser aplicado a un objeto y una clase, el objeto pertenece a dicha clase. La comprobación se hace en tiempo de ejecución.

Sintaxis

```
boolean resultado = <objeto> instanceof <clase>;
```

Ejemplo[2]

```
public class Figura {
}

public class Circulo extends Figura {
	private double radio;

	public Circulo(double r) {
		radio = r;
	}

	public double getRadio() {
		return radio;
	}

	@Override
	public String toString()
	{
		return String.format( "Circulo de radio %.2f", getRadio() );
	}
}

public class Rectangulo extends Figura {
	private double lado1;
	private double lado2;

	public Rectangulo(double l1, double l2)
	{
		lado1 = l1;
		lado2 = l2;
	}
```

2 Sería incorrecto diseñar el programa de esta forma; el ejemplo solamente se añade para clarificar la explicación.

```
	public double getLado1() {
		return lado1;
	}

	public double getLado2() {
		return lado2;
	}

	@Override
	public String toString()
	{
		return String.format( "Rectangulo de lados %.2f y %.2f",
		                      getLado1(), getLado2() );
	}
}

public class Ppal {
	public static double calculaArea(Figura f)
	{
		double toret = 0;

		if ( f instanceof Rectangulo ) {
			Rectangulo r = (Rectangulo) f;
			toret = r.getLado1() * r.getLado2();
		}
		else
		if ( f instanceof Circulo ) ) {
			Circulo c = (Circulo) f;
			toret = c.getRadio() * c.getRadio() * Math.PI;
		}

		return toret;
	}

	public static void main(String[] args)
	{
		Figura[] figuras = new Figura[ 2 ];
		figura[ 0 ] = new Rectangulo( 5, 6 );
		figura[ 1 ] = new Circulo( 1 );

		for(int i = 0; i < figuras.length; ++i) {
		    Figura f = figuras[ i ];
		    System.out.format( "%s, superficie: %.2f\n", f, calculaArea( f ) );
		}

		return;
	}
}
```

Salida

```
Rectangulo de lados 5.00 y 6.00, superficie: 30.00
Circulo de radio 1.00, superficie: 3.14
```

Nótese que el código anterior está basado en el ejemplo que contiene la superclase **Figura**, pero en el que no aparece el método *calculaArea()* en sus subclases, sino que existe en cambio un método estático en la clase **Ppal** que realiza la misma funcionalidad. De hecho, la salida del programa es la misma.

Este enfoque es completamente erróneo, debido a que resulta en un código poco mantenible: cada vez que se cree una nueva subclase de **Figura**, será necesario modificar el método estático *calculaArea()*, de forma que, de llegar a existir veinte figuras, albergará veinte estructuras *if* en su interior. Esto es inmanejable, poco legible, y extremadamente poco mantenible, especialmente frente a la alternativa de tener un método en cada subclase que anule el método de la superclase.

Un ejemplo correcto puede ser el que aparece a continuación, en el que **instanceof** se utiliza para un detalle concreto. El código vuelve a estar tomado del código de las figuras.

Ejemplo

```
// ...

public class Ppal {
      public static void listaRectangulos(Figura[] figuras)
      {
            for(int i = 0; i < figuras.length; ++i) {
                  Figura f = figuras[ i ];

                  if ( f instanceof Rectangulo ) {
                        System.out.format( "%s, superficie: %.2f\n",
                                          f, f.calculaArea() );
                  }
            }

            return;
      }

      public static void main (String[] args)
      {
            Figura r1 = new Rectangulo( 5, 6 );
            Figura c1 = new Circulo( 1 );

            listaRectangulos( new Figura[]{ r1, c1 } );
      }
}
```

Salida

```
Rectangulo de lados 5.00 y 6.00, superficie: 30.00
```

3.6 Clases abstractas

Retomando el ejemplo de las figuras, la clase **Figura** especialmente era contenedora de un método realmente extraño. El método *calculaArea()*, en la clase **Figura**, devuelve 0, como se ve a continuación.

Ejemplo

```
public class Figura {
	public double calculaArea()
	{
		return 0.0;
	}
}
```

Sin embargo, este valor de retorno no se llegaba a utilizar nunca, puesto que el método de **Figura** no se llegaba a ejecutar al estar reescrito en las subclases **Círculo** y **Rectángulo**. Es decir, lo realmente importante de este método es que, al estar definido en la superclase **Figura**, es seguro que todas sus subclases también lo tienen, proporcionando en todas estas clases una interfaz mínima.

En estos casos en los que el cuerpo del método no importa realmente, se puede declarar el método como *abstract*, abstracto lo que permite precisamente no especificar un cuerpo para ese método, y a la vez obliga a que las subclases creen su propia versión. En cuanto una clase declara un método abstracto, esa clase tiene que ser también declarada como *abstract*, con lo que automáticamente se vuelve imposible crear objetos de la misma.

Retomando el ejemplo de las figuras, la nueva clase **Figura** se lista a continuación. Esta clase se puede sustituir en el código global del ejemplo de las figuras sin hacer ningún otro cambio, ya que a) nunca se crearon objetos de la clase **Figura**, y b) las subclases especifican su propia versión de *calculaArea()*. Así, el código seguirá generando la misma salida que antes.

Ejemplo

```
public abstract class Figura {
	public abstract double calculaArea();
}
```

En el caso específico en el que, como ocurre con la clase **Figura**, todos sus miembros son abstractos, entonces se la puede llamar clase abstracta pura.

Nótese que no todas las subclases tienen que definir su propia versión de *calculaArea()*. Por ejemplo, la clase **Cuadrado** podría ser una subclase de **Rectángulo**, y al estar *calculaArea()* ya definido en la superclase, no sería necesario volver a escribirlo.

Ejemplo

```
public class Cuadrado extends Rectangulo {
	public Cuadrado(double l)
	{
		super( l, l );
	}
}
```

3.7 Evitando que una clase se convierta en superclase

En ocasiones, puede ser útil indicar que no se desea que una clase se convierta en superclase, al heredar de la misma. Para ello, se utiliza el modificador *final*, delante de la clase.

Sintaxis

```
final class <nombre_de_la_clase> {
	// ...
}
```

Ejemplo

```
public final class Cuadrado extends Rectangulo {
	public Cuadrado(double l)
	{
		super( l, l );
	}
}
```

Si en a partir del codigo anterior, añadido al codigo de las figuras, se intentase crear una subclase de Cuadrado, se produciria un error de compilacion.

4 Revisitando la visibilidad de miembros

En ocasiones, puede resultar útil el poder emplear atributos de la superclase en la subclase. Tal y como se han manejado hasta ahora, los atributos son privados en la superclase, mientras que en la subclase solamente se puede acceder a ellos a través de métodos marcados como públicos. Sin embargo, además de *public* y *private*, existe un tercer modificador de visibilidad: *protected*, y que, como los dos anteriores, puede aplicarse a cualquier miembro, atributo o método.

Un miembto *protected* sigue siendo privado a todos los efectos, excepto justamente para su acceso desde las subclases de la clase donde se define. El uso de *protected* no es preferible a limitarse al uso de *private* y *public*, pero en ocasiones puede simplificar mucho el código y el diseño, al no requerir crear un método público de acceso a un atributo cuando en realidad solamente se desea acceder a él desde subclases.

En el siguiente ejemplo, se crea la clase **CandadoMinTresPosiciones**, de forma que cuando se le pasa al constructor una clave menor de 100 (menos de tres dígitos), le suma precisamente 100, de manera que una clave como 5 quedaría convertida en 105, mientras una de 49 se convertiría en 149. Para poder ofrecer esta funcionalidad, la clase **CandadoMinTresPosiciones** necesita tener acceso al atributo *clave*, pues va a leerlo y a modificarlo. La mejor solución será marcar el atributo como *protected*, con lo que, a pessar de estar definido en Candado, será también accesible en **CandadoMinTresPosiciones**, sin ser necesario crear un método público *getClave()* en **Candado**, lo cual sería catastrófico para la funcionalidad de la clase.

Ejemplo

```
/** Representa un candado con clave */
public class Candado {
	protected int clave;
	private boolean abierto;

	/** Crea un candado, asignando una clave.
	  * @param k La clave, como entero */
	public Candado(int k)
	{
		clave = k;
		abierto = false;
	}

	/** Devuelve el estado del candado
	  * @return true si abierto, false cerrado. */
	public boolean estaAbierto()
	{
		return abierto;
	}
```

```
	/** Abre el candado
	  * @param k La clave a probar, como entero */
	public void abre(int k)
	{
		if ( clave == k ) {
			abierto = true;
		}
	}

	/** Cierra el candado */
	public void cierra()
	{
		abierto = false;
	}

	@Override
	public String toString()
	{
		return "Candado abierto: " + estaAbierto();
	}
}

public class CandadoMinTresPosiciones extends Candado {
	public CandadoMinTresPosiciones(int k)
	{
		super( k );

		// Clave de menos de tres posiciones?
		if ( clave < 100 ) {
			clave += 100;
		}

		return;
	}
}

public class Ppal {
	public static void main(String[] args)
	{
		CandadoMinTresPosiciones c1 = new CandadoMinTresPosiciones( 2 );

		System.out.println( c1 );
		c1.abre( 102 );
		System.out.println( c1 );
	}
}
```

Salida

```
Candado abierto: false
Candado abierto: true
```

5 Revisitando las excepciones

En documentos anteriores, se ha empleado la clase **Exception** como clase a utilizar, en general, para todas las posibles excepciones que se pudieran producir. En realidad, esto no es demasiado práctico, pues no permite distinguir entre varias posibles situaciones de error. Así, es posible crear nuestra propia jerarquía de herencia de excepciones, solo con crear nuevas clases que hereden, directa o indirectamente, de **Exception**.

Sintaxis

```
class <Excepción_propia> extends Exception {
      public <Excepción_propia>(String msg) {
            super( msg );
      }
}
```

En el siguiente ejemplo, se crea una clase **ListaCompra** que utiliza un vector para almacenar un número de elementos máximo. La clase lleva un control del número de elementos realmente utilizados del máximo del vector.

Pueden ocurrir dos excepciones durante la ejecución: que se intente acceder a un elemento inexistente, o que que se intente insertar un elemento de más cuando el vector ya está lleno. A continuación aparece la clase **ListaCompra** empleando **Exception** para tratar estas situaciones de error.

Ejemplo

```
/** Representa una serie de elementos a comprar */
class ListaCompra {
      private String[] elementos;
      private int num;

      /** Crea una nueva lista de la compra
        * @param max El num. max. de elementos */
      public ListaCompra(int max)
      {
            elementos = new String[ max ];
            num = 0;
      }

      /** @return el max. de elementos que caben */
      public int getMaxElementos()
      {
            return elementos.length;
      }
```

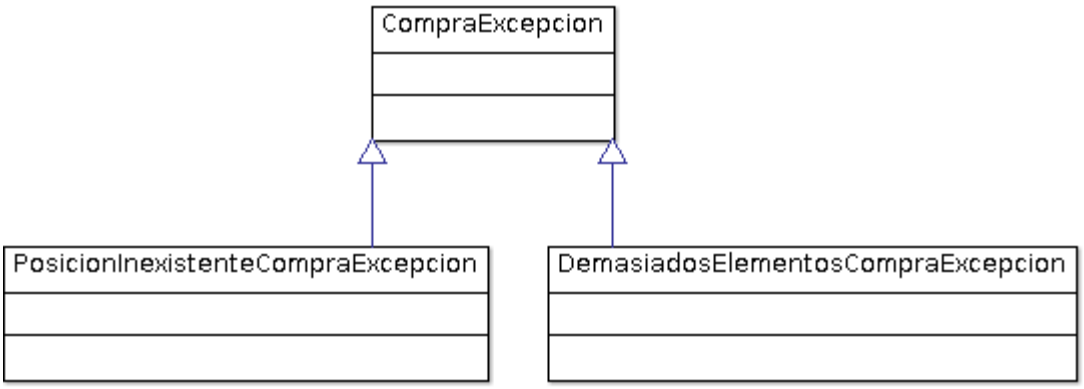
```
	/** @return el num. de elementos existentes */
	public int getNumElementos()
	{
		return num;
	}

	/** Anota un nuevo elemento en la lista
	  * @param elemento El nuevo elemento */
	public void inserta(String elemento) throws Exception
	{
		if ( getNumElementos() >= getMaxElementos() ) {
			throw new Exception( "max. alcanzado con: " + elemento );
		}

		elementos[ num ] = elemento;
		++num;
	}

	/** Devuelve un elemento de la lista
	  * @param i La pos del elemento */
	public String get(int i) throws Exception
	{
		if ( i >= getNumElementos() ) {
			throw new Exception( "pos. no existe: " + i );
		}

		return elementos[ i ];
	}

	@Override
	public String toString()
	{
		StringBuilder toret = new StringBuilder();

		try {
			for(int i = 0; i < getNumElementos(); ++i) {
				toret.append( " - " );
				toret.append( get( i ) );
				toret.append( '\n' );
			}
		} catch(Exception exc)
		{
			toret.append( "..." );
		}


		return toret.toString();
	}
}
```

```
public class Ppal {
      public static void main (String[] args)
      {
            ListaCompra lc = new ListaCompra( 2 );

            try {
                  lc.inserta( "lechuga" );
                  System.out.println( "Lista al insertar 'lechuga':\n" + lc );

                  lc.inserta( "tomate" );
                  System.out.println( "Lista al insertar 'tomate':\n" + lc );

                  lc.inserta( "cebolla" );
                  System.out.println( "Lista al insertar 'cebolla':\n" + lc );
            }
            catch(Exception exc)
            {
                  System.err.println( "\nERROR: " + exc.getMessage() );
            }
      }
}
```

Salida

```
Lista al insertar 'lechuga':
 - lechuga

Lista al insertar 'tomate':
 - lechuga
 - tomate

ERROR: max. alcanzado, con: cebolla
```

Los métodos que pueden generar excepciones son *get()* e *inserta()*. De hecho, la razón de que exista un *try... catch* en *toString()* es debido al uso de *get()*. Si bien es posible especificar "*throws Exception*" en la cabecera del método *toString()*, esto lo convertiría en un método distinto al **Object**.*toString()* que se desea reescribir (no es lo mismo un método que lanza una excepción, a uno que no lanza ninguna).



La nueva jerarquía de excepciones aparece en el siguiente diagrama. **CompraExcepcion** es directamente una subclase de **Exception**, mientras el resto son subclases de la primera.

Ejemplo

```
public class CompraExcepcion extends Exception {
      public CompraExcepcion(String msg)
      {
            super( msg );
      }
}

public class DemasiadosElementosCompraExcepcion extends CompraExcepcion {
      public DemasiadosElementosCompraExcepcion(String msg)
      {
            super( msg );
      }
}

public class PosicionInexistenteCompraExcepcion extends CompraExcepcion {
      public PosicionInexistenteCompraExcepcion(String msg)
      {
            super( msg );
      }
}
```

A continuación aparecen los métodos *get()* e *inserta()* de la misma clase **ListaCompra**, utilizando las nuevas clases de excepciones.

Ejemplo

```
/** Representa una serie de elementos a comprar */
public class ListaCompra {
      private String[] elementos;
      private int num;

      // ...
      /** Anota un nuevo elemento en la lista
        * @param elemento El nuevo elemento */
      public void inserta(String elemento)
            throws DemasiadosElementosCompraExcepcion
      {
            if ( getNumElementos() >= getMaxElementos() ) {
                  throw new DemasiadosElementosCompraExcepcion(
                                    "max. alcanzado, con: " + elemento );
            }

            elementos[ num ] = elemento;
            ++num;
      }
```

```
	/** Devuelve un elemento de la lista
	  * @param i La pos del elemento */
	public String get(int i) throws PosicionInexistenteCompraExcepcion
	{
		if ( i >= getNumElementos() ) {
			throw new PosicionInexistenteCompraExcepcion(
					"pos. no existe: " + i );
		}

		return elementos[ i ];
	}

	// ...
}
```

El hecho de emplear la herencia permite clasificar las excepciones, de manera que se puede distinguir entre ellas cómodamente, al contrario que cuando se utiliza directamente la clase **Exception**. A continuación se muestra la clase principal utilizando las nuevas excepciones, lo que permite ofrecer comportamientos más específicos.

Ejemplo

```
public class Ppal {
	public static void main(String[] args)
	{
		ListaCompra lc = new ListaCompra( 2 );

		try {
			lc.inserta( "lechuga" );
			System.out.println( "Lista al insertar 'lechuga':\n" + lc );

			lc.inserta( "tomate" );
			System.out.println( "Lista al insertar 'tomate':\n" + lc );

			lc.inserta( "cebolla" );
			System.out.println( "Lista al insertar 'cebolla':\n" + lc );
		}
		catch(CompraExcepcion exc)
		{
			System.err.println( "\nERROR con la lista de la compra: "
					+ exc.getMessage() );
		}
		catch(Exception exc)
		{
			System.err.println( "\nERROR inesperado: " + exc.getMessage() );
		}
	}
}
```

Salida

```
Lista al insertar 'lechuga':
 - lechuga

Lista al insertar 'tomate':
 - lechuga
 - tomate

ERROR con la lista de la compra: max. alcanzado, con: cebolla
```

5.1 Clases anidadas

Existen ocasiones, como la anterior, en la que ciertas clases solamente se van a emplear en conjunción con otra clase, que podríamos decir, engloba a las primeras. En el caso de las excepciones propias que aparecen más arriba, estas solo se usan cuando se llama a algún método de la clase **ListaCompra** (concretamente, *get()* e *inserta()*). Así, tiene sentido crear las clases de excepciones propias como clases anidadas. Solo hay dos cambios a tener en cuenta. El primero es que, desde fuera de la clase **ListaCompra**, será necesario prefijar las excepciones con "ListaCompra.", pues estas existen en su interior. A continuación, aparece el código fuente completo. En cuanto al segundo, hay distintos tipos de clases anidadas. Las más apropiadas para este caso son las *static*, que indican una clase anidada en la que no se tiene acceso a los miembros de la clase que la engloba. Las clases que no llevan el modificador *static*, serán normalmente creadas y manejadas solo desde dentro de la clase que las engloba.

Ejemplo

```
/** Representa una serie de elementos a comprar */
public class ListaCompra {

	/** Excepciones propias de la Lista de la compra */
	public static class CompraExcepcion extends Exception {
		public CompraExcepcion(String msg) {
			super( msg );
		}
	}

	/** Acceso a un elemento de la lista fuera de rango */
	public static class RangoCompraExcepcion extends CompraExcepcion {
		public RangoCompraExcepcion(String msg) {
			super( msg );
		}
	}

	/** Intento de superar el max. de elementos en la lista */
	public static class LimiteCompraExcepcion extends CompraExcepcion {
		public LimiteCompraExcepcion(String msg) {
			super( msg );
		}
	}
```

```
	private String[] elementos;
	private int num;

	/** Crea una nueva lista de la compra
	  * @param max El num. max. de elementos */
	public ListaCompra(int max)
	{
		elementos = new String[ max ];
		num = 0;
	}

	/** @return el max. de elementos que caben */
	public int getMaxElementos()
	{
		return elementos.length;
	}

	/** @return el num. de elementos existentes */
	public int getNumElementos()
	{
		return num;
	}

	/** Anota un nuevo elemento en la lista
	  * @param elemento El nuevo elemento */
	public void inserta(String elemento) throws LimiteCompraExcepcion
	{
		if ( getNumElementos() >= getMaxElementos() ) {
			throw new LimiteCompraExcepcion( "max. alcanzado, con: "
                                                + elemento );
		}

		elementos[ num ] = elemento;
		++num;
	}

	/** Devuelve un elemento de la lista
	  * @param i La pos del elemento */
	public String get(int i) throws RangoCompraExcepcion
	{
		if ( i >= getNumElementos() ) {
			throw new RangoCompraExcepcion( "pos. no existe: " + i );
		}

		return elementos[ i ];
	}
```

```
	public String toString()
	{
		StringBuilder toret = new StringBuilder();

		try {
			for(int i = 0; i < getNumElementos(); ++i) {
				toret.append( " - " );
				toret.append( get( i ) );
				toret.append( '\n' );
			}
		} catch(Exception exc)
		{
			toret.append( "..." );
		}


		return toret.toString();
	}
}

public class Ppal {
	public static void main(String[] args)
	{
		ListaCompra lc = new ListaCompra( 2 );

		try {
			lc.inserta( "lechuga" );
			System.out.println( "Lista al insertar 'lechuga':\n" + lc );

			lc.inserta( "tomate" );
			System.out.println( "Lista al insertar 'tomate':\n" + lc );

			lc.inserta( "cebolla" );
			System.out.println( "Lista al insertar 'cebolla':\n" + lc );
		}
		catch(ListaCompra.CompraExcepcion exc) {
			System.err.println( "\nERROR de compras: " + exc.getMessage() );
		}
		catch(Exception exc) {
			System.err.println( "\nERROR inesperado: " + exc.getMessage() );
		}
	}
}
```

Salida

```
Lista al insertar 'lechuga':
 - lechuga

Lista al insertar 'tomate':
 - lechuga
 - tomate
ERROR con la lista de la compra: max. alcanzado, con: cebolla
```

6 Interfaces

Desde un punto de vista pragmático, de funcionalidad, las interfaces pueden ser entendidas como clases abstractas puras, con una sintaxis ligeramente distinta. En cuanto a la notación, los identificadores de las interfaces llevan una 'I' mayúscula como prefijo.

Sintaxis

```
interface <nombre_de_interface> {
      <cabecera de método1>;
      <cabecera de método2>;
      ...
      <cabecera de métodon>;

}
```

Una diferencia con respecto a las clases abstractas puras, es que en cuanto a interfaces se dice que la clase las implementa, no que las hereda. Además, una clase dada puede implementar varias interfaces.

Sintaxis

```
class <nombre_de_clase> extends superclase implements interfaz1, interfaz2,... {
      // ...
}
```

Dado que Java no permite la herencia múltiple (es decir, que una subclase esté relacionada con más de una superclase), lo más parecido es que una clase utilice varias interfaces.

Se pueden crear referencias de una interfaz que apunten a una clase que implementa dicha interfaz, así como el *downcasting* y el *upcasting* funcionan exactamente igual.

En el siguiente ejemplo, se utiliza el código en el que la clase **Persona** es la superclase de **Empleado**. Además, la interfaz **IGeneradorHTML** aplicada a **Persona** obliga a que ambas también tengan el método *toHTML()*[3] (no es estrictamente obligatorio que **Empresa** lo defina, al estar también definido en su superclase, pero la reescritura en este caso es recomendable para ofrecer toda la información posible en el método). Finalmente la interfaz **IAsalariado** se aplica exclusivamente a **Empleado**, de forma que le obliga a definir el método *getSalario()*.

3 HTML es un lenguaje de marcado utilizado principalmente en páginas web. Véase http://es.wikipedia.org/wiki/HTML

Ejemplo

```
public interface IGeneradorHTML {
	public String toHTML();
}

public interface IAsalariado {
	public double getSalario();
}

public class Persona implements IGeneradorHTML {
	private String nombre;

	public Persona(String n)
	{
		nombre = n;
	}

	public String getNombre()
	{
		return nombre;
	}

	@Override
	public String toString()
	{
		return getNombre();
	}

	@Override
	public String toHTML()
	{
		return "<b>" + getNombre() + "</b>";
	}
}
```

```
public class Empleado extends Persona implements IAsalariado {
	private String empresa;
	private double salario;

	public Empleado(String n, String e, double s)
	{
		super( n );
		empresa = e;
		salario = s;
	}

	public String getEmpresa()
	{
		return empresa;
	}

	@Override
	public String toString()
	{
		return super.toString() + ": " + getEmpresa()
				+ " (" + getSalario()  + ')';
	}

	@Override
	public String toHTML()
	{
		return super.toHTML() + ": <i>" + getEmpresa()
				+ " (" + getSalario()  + ")</i>";
	}

	@Override
	public double getSalario()
	{
		return salario;
	}
}

public class Ppal {
	public static void main(String[] args)
	{
		IGeneradorHTML e1 = new Empleado( "Carolina Alguacil", "Bar", 1000 );

		System.out.println( e1 );
		System.out.println( e1.toHTML() );
	}
}
```

Salida

```
Carolina Alguacil: Bar (1000.0)
<b>Carolina Alguacil</b>: <i>Bar (1000.0)</i>
```