Introducción a la Programación

Tema 4. Tipos. Implementación y Reutilización

1. Detalles de Implementación: Modificadores de atributos y métodos

Veamos, en primer lugar los detalles que debemos tener en cuenta para implementar los tipos.

Como hemos visto en capítulos anteriores *Java* nos permite escribir algunas palabras reservadas que modifican las características de **acceso** o **comportamiento** del elemento al que preceden.

Hay dos tipos de modificadores:

- **De acceso:** Son los que permiten modificar la visibilidad del elemento, es decir, desde qué puntos de un programa Java son accesibles.
- **De comportamiento**: Son los que permiten modificar el funcionamiento y la manera de uso de los elementos.

Modificadores de atributos

Modificadores de **acceso**:

- **public**: El atributo es accesible desde cualquier punto del programa en el que se disponga de un objeto de la clase.
- **private:** El atributo es accesible sólo desde los métodos de la propia clase.
- **protected**: El atributo es accesible desde los métodos de la propia clase, desde los de las clases que hereden de ésta y desde los de las clases del mismo paquete (aunque no hereden de ésta).

Modificadores de **comportamiento**:

- **static:** Todas las instancias de la clase comparten el mismo valor para el atributo. Los atributos declarados *static "*pertenecen a" un objeto especial que tiene el mismo nombre que la clase. También se llaman **atributos de clase** frente a los no *static* que se denominan **atributos de instancia.**
- **final:** No se permite cambiar el valor inicial del atributo (atributo constante).

Modificadores de métodos

Modificadores de **acceso**:

- **public**: El método es visible desde cualquier punto del programa en el que se disponga de un objeto de la clase.
- **private:** El método es visible sólo desde los métodos de la propia clase.
- **protected**: El método es visible desde los métodos de la propia clase, desde los de las clases que hereden de ésta y desde los de las clases del mismo paquete (aunque no hereden de ésta).

Modificadores de **comportamiento**:

- **static:** El método puede ser invocado sin necesidad de crear una instancia de la clase. Los métodos declarados *static* deben ser invocados sobre un objeto especial que tiene el mismo nombre que la clase. Desde ellos sólo se permite el acceso a atributos o métodos declarados *static*. También se llaman **métodos de clase** frente a los no *static* que se denominan **métodos de instancia.**
- **final:** No se permite que las clases hijas redefinan este método.

Criterios para elegir los modificadores de atributos y métodos

En general, declararemos los atributos con el modificador ***private***. Los métodos a implementar definidos en la interfaz, los declararemos con el modificador ***public***.

En ocasiones, es útil definir constantes de clase: valores constantes para todos los objetos de un tipo. Para ello definiremos un atributo con los modificadores ***public, static y final***.

Usaremos el modificador **static** para los atributos que guarden información relativa a la población de objetos del tipo, y no al estado de un objeto particular. Por ejemplo, para contar el número de instancias creadas del tipo. Denominamos **población de objetos del tipo** al conjunto de objetos creados de ese tipo.

Usaremos el modificador **static** para los métodos **funcionales**: aquellos que se comportan de manera funcional, recibiendo datos por parámetro y devolviendo el resultado, sin involucrar a ningún atributo del objeto implícito. Por ejemplo, en las clases de utilidad (Colecciones, Naturales, …). Los métodos **static** pueden acceder modificar el valor de los atributos **static.** Por cada clase existe un objeto especial, con el mismo nombre de la clase, cuyos atributos y métodos son los etiquetados con **static.**

Un método declarado **static** no puede utilizar atributos ni métodos de la clase que no hayan sido declarados **static**. Un método no **static** sí puede usar un atributo o un método **static**. Un método **static** puede ser invocado (junto con el objeto especial cuyo nombre es el de clase) sobre cualquier elemento de la clase.

Si un método es **static** se puede invocar sobre el nombre de la clase:

```
Math.sqrt(2.0);
```

¿Por qué podemos acceder a los métodos estáticos de esta forma?

El nombre de una clase en Java significa varias cosas, dependiendo del contexto:

- Es el nombre de un **tipo.**
- Es el nombre de un **constructor.**

- Es el nombre de una **clase.**
- Es el nombre de un **objeto** creado automáticamente por la máquina virtual de Java, que contiene toda la información *del tipo* (un objeto con los atributos y métodos que hayan sido definidos **static**). En el ejemplo anterior ese objeto es *Math.*

Algunas características más de los atributos y métodos *static*

- Relación con los tipos genéricos.
    - En tiempo de ejecución por cada tipo genérico sólo existe su correspondiente tipo desnudo. El nombre de ese tipo desnudo representa a un objeto (que crea el compilador) que contiene todos los atributos y métodos declarados *static*. Esto implica que los atributos *static* (igualmente los métodos) son compartidos por las poblaciones de todas las instancias del tipo genérico.
    - Una segunda consecuencia es que los métodos *static* definidos en tipos genéricos tienen que ser invocados sobre el tipo desnudo.
- Métodos *static* e interfaces

Los métodos *static* son métodos de clase. Los métodos definidos en una interfaz son métodos de instancia. Esto quiere decir que si una clase implementa una interfaz no puede tener un método *static* con el mismo nombre que otro definido en la interfaz.

2. Implementación de tipos

Para implementar la funcionalidad expresada por un contrato en Java, debemos construir una clase que será la implementación del contrato. Una implementación es una relación entre un **contrato,** que expresa el diseño de un tipo, y una **clase** concreta que lo implementa. La clase deberá incluir los métodos definidos en el contrato y estos respectar las restricciones que se hayan indicado. Para comprobar que una clase implementa un contrato se ejecutarán el conjunto de casos de prueba incluidos en el contrato.

La primera decisión a tomar es si para el tipo diseñamos un interface y posteriormente una clase que lo implementa o solamente una clase. La segunda cuestión es decidir qué clase implementará la factoría del tipo. Suponiendo diseñado un contrato para el tipo *T* tenemos varias alternativas:

- La más simple es diseñar la clase *T* (con el mismo nombre del tipo). Los métodos públicos de esta clase serán los especificados en el contrato para *T.* Además la clase T implementará la factoría de *T*. Esta opción es la más razonable cuando sólo estamos considerando una implementación para el tipo T.
- La segunda opción es diseñar el interface *T*, las clases *TImpl1*, *TImpl2*, … que lo implementan y las clase *Tes* (nombre del tipo en plural) que implementa la factoría de *T.* Esta segunda opción es la recomendable cuando estamos considerando varias implementaciones posibles.

Tomada la decisión anterior debemos implementar una o varias clases (estas pueden ser abstractas). Para cada una de ellas debemos definir los atributos, constructores y métodos (públicos y privados). Para ello hay que tomar algunas decisiones:

- ¿Cuántos atributos y de qué tipos? Una primera idea es poner un atributo por cada propiedad no derivada. Cada atributo tiene como nombre el de la propiedad pero empezando por minúscula y el tipo de la misma. Si la propiedad asociada es compartida por toda la población del tipo entonces el atributo llevará el modificador *static.*
- El estado de los objetos debe representar de forma mínima las propiedades de los objetos. Esto quiere decir que tenemos escoger una forma concreta de guardar las propiedades en los atributos de la forma más simple y eficiente posible. Es lo que denominamos **forma normal** de los valores de ese tipo. Elegir adecuadamente esta **forma normal** es muy importante, tal como hemos visto en el capítulo anterior, para implementar la igualdad, los métodos *hashCode, toString* y el orden natural. Una posibilidad es escoger la forma normal para que guarde los valores del representante canónico de cada una de las clases de equivalencia definidas por la igualdad.
- ¿Qué nuevos métodos privados son necesarios? Este paso es muy importante porque elegidos convenientemente pueden simplificar considerablemente el código. Un criterio es diseñar un método privado para cada tarea que podamos compartir en la implementación del resto de los métodos.
- ¿Cuántos constructores? Uno con todos los datos suficientes para dar valor a los atributos, otro sin parámetros y en muchos casos uno que tome un *String* como parámetro. En algunos casos puede ser interesante algún constructor más.

- Los constructores deben disparar excepciones si no pueden construir un objeto en un estado válido con los parámetros dados.
- Los constructores deben dar valor a todos los atributos.
- La implementación del método *equals, hasCode, toString* y *compareTo,* en su caso, deben hacerse de forma coherente y siguiendo las indicaciones ya explicadas en el capítulo anterior.

Al implementar los constructores:

- Debemos tener en cuenta el **invariante** del contrato (debe cumplirse al finalizar la ejecución del constructor).
- Si no puede garantizar el cumplimiento la restricción anterior ante unos parámetros determinados, se debe lanzar una **excepción**.
- Es recomendable que el constructor dé un valor inicial a cada uno de los atributos.

Al implementar los métodos:

- Debemos tener en cuenta la precondición, postcondición e invariante. Debe asegurarse que si se cumple la precondición, se cumplirán tras la ejecución del método tanto la postcondición como la invariante.
- Si se verifica alguna condición de disparo de excepción, entonces hay que disparar las excepciones definidas en el contrato.
- En la zona no definida en el contrato el método puede ser implementado libremente.

2.1 El tipo Racional

Veamos los detalles de implementación del tipo racional diseñado en el capítulo anterior y que volvemos a recordar aquí:

Racional

Descripción: Racional. Inmutable

Propiedades:

- *Numerador: Integer*, consultable.
- *Denominador: Integer*, consultable
- *ValorReal: Double,* consultable.

Propiedades relacionadas con la igualdad y órdenes

- *Criterio de igualdad*: dos racionales $\frac{a}{b}, \frac{c}{d}$ de este tipo son iguales si $ad = bc$
- *Representación como cadena*: de la forma a/b si b es distinto de 1 en otro caso a.
- *Orden natural*: dados dos racionales, $\frac{a}{b}, \frac{c}{d}$, el primero es menor que el segundo si $ad < bc$
- Orden según el numerador
- Orden según el denominador

Operaciones:

- *Racional suma(Racional r).*
- *Racional resta(Racional r)*
- *Racional multiplica(Racional r)*
- *Racional divide(Racional r)*
- *Racional invierte().*

Factoría:

- *Racional getCero().*
- *Racional getUno().*
- *Racional create(Integer a, Integer b).*
- *Racional create(Integer a).*
- *Racional create(String s).*
- *Racional create(Racional r).*

Un conjunto de casos de prueba es:

Método	this	p	r	Excepción
create		(4,6)	(2,3)	
create		(-2,-3)	(2,3)	
create		(1,0)		*IllegalArgumentException*
create		(“-8/4”)	(-2,1)	
create		(“2 3”)		*IllegalArgumentException*
getNumerador	(-5,9)		-5	

getDenominador	(6,3)		1	
suma	(1,4)	(3,4)	(1,1)	
resta	(3,2)	(4,2)	(-1,2)	
multiplica	(2,3)	(3,2)	(1,1)	
divide	(1,2)	(1,4)	(2,1)	
divide	(1,2)	(0,1)		*ArithmeticException*
invierte	(4,2)		(1,2)	
invierte	(0,7)			*ArithmeticException*
getValorReal	(3,2)		1.5	
equals	(3,4)	(6,8)	true	
equals	(1,2)	(1,4)	false	
toString	(-6,-9)		“2/3”	
toString	(8,-4)		“-2”	
compareTo	(1,3)	(5,6)	-1	

Decidimos seguir la primera de las estrategias. Es decir implementamos únicamente la clase *Racional*. Esta clase implementará los métodos del tipo y la factoría. El tipo se diseñó como inmutable. Los métodos de la factoría decidimos que sean *static*.

La clase Racional tiene la forma:

```
public class Racional implements Comparable<T> {

   public static Racional getCero(){..}
   public static Racional getUno(){..}
   public static Racional create(Integer a, Integer b) {..}
   public static Racional create(Integer a) {..}
   public static Racional create(String s) {..}
   public static Racional create(Racional r){..}

      //Atributos
      //Constructores
      //Métodos privados

   public Integer getNumerador(){..}
   public Integer getDenominador(){..}
   public Double getValorReal(){..}
   public Racional suma(Racional r) {..}
```

```
    public Racional resta(Racional r);
    public Racional multiplica(Racional r) {..}
    public Racional divide(Racional r) {..}
    public Racional invierte(){..}
    public String toString(){..}
    public boolean equals(Object p) {..}
    public int hashCode(){..}
    public int compareTo(Racional r) {..}
    public int compareByNumerador(Racional r){...}
    public int compareByDenominador{...}
}
```

La implementación que hagamos debe funcionar de acuerdo con los casos de prueba. Esto puede comprobarse diseñando, a partir de los casos de prueba, una clase *TestRacional*. Alternativamente podemos usar la herramienta *jUnit* para comprobar si la implementación responde como se espera al invocar a los diferentes métodos o llamar a los constructores.

Por ejemplo el caso de prueba:

Método	this	p	r	Excepción
create		(“8/-4”)	(-2,1)	

Se concreta en el test:

```
public class TestRacionalJUnit {
    Racional r;
    @Test
       public void testRacionalString() {
              r = Racional.create("8/-4");
              int n = r.getNumerador();
              int d = r.getDenominador();
              assertEquals(n,-2);
              assertEquals(d,1);
       }
       ...
```

El siguiente paso es decidir los atributos. Decidimos guardar las propiedades básicas del representante canónico de cada una de las clases de equivalencia definidas por la igualdad. Es

decir para cada racional su representante canónico es el racional simplificado con denominador positivo.

Escogemos, por lo tanto como atributos *numerador* y *denominador* y diseñamos un método privado que encuentre la representación normal. Es decir el representante canónico correspondiente. Esto se consigue dividiendo numerador y denominador por el máximo común divisor de ambos cambiado de signo si el denominador es negativo.

```
private Integer numerador;
private Integer denominador;

private void normaliza(){
       Integer m = Math2.mcd(numerador,denominador);
       if(denominador<=0) m =-m;
       numerador = numerador/m;
       denominador = denominador/m;
}
```

El siguiente paso es la implementación de los constructores necesarios. Diseñamos dos constructores: uno que toma dos parámetros de tipo entero y otro que toma un String que representa el racional. El segundo tipo de constructores lo veremos más adelante.

```
private Racional(Integer a, Integer b){
    if(b==0) throw new IllegalArgumentException(
                  "El denominador no puede ser cero");
    this.numerador=a;
    this.denominador=b;
          normaliza();
    }
}
private Racional (String s) {..}
```

Vemos como el constructor dispara una excepción si el denominador es igual a cero y posteriormente normaliza los valores guardados del numerador y denominador. A partir de ellos es cómodo implementar los métodos de la factoría.

```
public static Racional getCero() { return new Racional(0,1); }
public static Racional getUno() { return new Racional(1,1);}
```

```
 public static Racional create(Integer a, Integer b) {
       return new Racional(a, b);
}
public static Racional create(Integer a) { return new Racional(a,1);}
public static Racional create(String s) { return new Racional(s); }
public static Racional create(Racional r) {
           return new Racional(r.getNumerador(), r.getDenominador());
}
```

A partir de lo anterior es fácil diseñar los métodos *get*.

```
    . . .
Integer getNumerador(){
    return numerador;
}
Integer getDenominador(){
    return denominador;
}
Double getValorReal(){
     return ((double)getNumerador())/getDenominador();
}


    . . .
```

Los métodos asociados a las operaciones pueden tener un código más o menos complejo dependiendo de la funcionalidad ofrecida. En general estos métodos toman unos parámetros, llevan a cabo las operaciones correspondientes y actualizan el estado del objeto dejándolo en la forma normal o construyen un objeto nuevo. El objeto de la clase que estamos diseñando lo denominamos *this*. El estado de *this* son los atributos de la clase. Así el método *this.suma(r)* hace el trabajo equivalente a *this +r*. Es decir, suma *r* a *this* y, en este caso por ser un objeto inmutable, construye un objeto nuevo. Claramente la expresión *s = o+r,* o la anterior con *this,* no es correcta en Java aunque *o* sea un objeto racional porque el operador *+* no está disponible para ese tipo. La expresión correcta es *s = o.suma(r).*

La implementación de estos métodos es:

```
  . . .
  Racional suma(Racional r) {
    Integer a = numerador*r.getDenominador()+denominador*r.getNumerador();
    Integer b = denominador*r.getDenominador();
```

```
    return create(a,b);
  }

  Racional resta(Racional r){
    Integer a = numerador*r.getDenominador()-denominador*r.getNumerador();
    Integer b = denominador*r.getDenominador();
    return create(a,b);
  }

  Racional multiplica(Racional r){
    Integer a = numerador*r.getNumerador();
    Integer b = denominador*r.getDenominador();
    return create(a,b);
  }

  Racional divide(Racional r){
    if(r.getNumerador() == 0) throw new ArithmeticException
                        ("No se puede dividir entre cero");
    Integer a = numerador*r.getDenominador();
    Integer b = denominador*r.getNumerador();
    return create(a,b);
  }

  Racional invierte(){
    if(numerador == 0) throw new ArithmeticException
                        ("No se puede invertir si el numerador es cero");
    Integer a = numerador;
    Integer b = denominador;
    return create(a,b);
  }
  . . .
```

En general, como podemos observar, es recomendable hacer los cálculos necesarios en variables locales adecuadas y posteriormente asignar el valor calculado a los atributos. Si el tipo fuera mutable entonces, posiblemente, el método actualizaría el valor de los atributos en lugar de crear un objeto nuevo.

2.2 Métodos relacionados con la igualdad

A partir de esa expresión lógica quedan definidas las clases de equivalencia. A partir de la relación de equivalencia podemos definir las funciones relacionadas con la igualdad. Podemos seguir los siguientes pasos:

- Definir la expresión lógica que define la equivalencia entre dos objetos. Esta expresión lógica debe devolver *true* si los dos objetos son *null*. Es importante conocer si las propiedades del tipo pueden tomar el valor *null* o no para impedir que se disparen excepciones no deseadas. Si las propiedades pueden tomar el valor *null* entonces debemos comprobar si ambos son *null* o no antes de pasar a invocar la expresión que define la relación de equivalencia.
- Por cada clase de equivalencia escogemos un representante canónico de todos los objetos de clase. Implementamos el método *hash(e)* para que devuelva un entero calculado sobre el representante canónico de la clase a la que pertenece el parámetro *e*. El método *hash* devolverá *0* sobre le objeto *null*.
- El método *hash* sobre un objeto dado se debe calcular combinando los códigos *hash* de las propiedades del representante canónico definido a partir de la relación de equivalencia escogida. Una forma de hacerlo es s*umar* los *hash* de las propiedades del representante canónico multiplicadas por mú*ltip*los de un número primo dado. Una elección de este número primo puede s*er 31.*
- *E*l resultado del método *toString,* coherente con la relación de equivalencia anterior, s*e debe calcular s*olamente a partir de las representaciones como *String* d*e las pr*opiedades involucradas en la relación de equivalencia o d*e otras* propiedades que sean derivadas de ellas pero calculadas sobre el representante canónico.

Veamos algunos ejemplos:

1. Dos objetos de tipo *Punto2D* son equivalentes si tienen iguales su *X* y su *Y*.
2. Dos racionales $\frac{a}{b}$ y $\frac{c}{d}$ son iguales si se cumple $ad == bc$.

De las definiciones anteriores podemos ver que en el caso 1 cada clase de equivalencia está formada por un solo objeto con valores definidos para las propiedades *X* e *Y*. Sin embargo en el segundo caso una clase de equivalencia está formada por una infinitud de racionales. En cada caso podemos escoger un representante canónico de cada una de las clases. En el caso 1 podemos escogemos el único miembro de cada clase. En el segundo la elección es arbitraria. En concreto escogemos como representante canónico el número racional simplificado y con denominador positivo. Así el racional $\frac{3}{-6}$ tiene como representante canónico $\frac{-1}{2}$.

Un tipo puede extender otros tipos y por cada uno de ellos podemos definir relaciones de equivalencia. En general sobre un tipo podemos definir distintas relaciones de equivalencia. Pero cada objeto (de una clase dada) tiene un único método *equals(Object o).* A su vez los métodos *hashCode* y *toString* deben ser coherentes con el mismo. A partir de la relación de equivalencia escogida debemos diseñar ese único método *equals*. Para ello debemos tomar algunas decisiones y posteriormente seguir algunas recomendaciones. Estas pueden ser las siguientes:

- Escoger unos de los tipos ofrecidos por el objeto para definir la igualdad a partir de él. Es decir asumir que el objeto actual será igual a otro si este otro ofrece el tipo que hayamos escogido. Con lo que sabemos hasta ahora si tenemos una clase que implementa una interfaz el tipo definido por la interfaz y el definido por la clase son tipos posibles. Cada uno de ellos tiene sus ventajas e inconvenientes. Debemos tener en cuenta, por su importancia para la posterior implementación, si el tipo escogido permite que algunas de sus propiedades puede tomar como valor *null* o no. Si no tenemos información al respecto asumimos que las propiedades del tipo pueden tomar *null* como valor.
- Escoger una relación de equivalencia sobre los objetos del tipo anterior y cuyos detalles hemos decidido previamente.

Una vez tomadas las decisiones anteriores podemos pasar a implementar los métodos *equals, hashCode y toString.* Para ello podemos seguir las siguientes indicaciones:

- Si dos objetos son idénticos entonces son iguales. Esto se comprueba con el operador ==.
- Los dos objetos deben ser de un tipo dado: el tipo escogido previamente. Esto se puede comprobar con el operador *instanceof* y si el tipo es una clase también con el método *getClass().* Suponemos que el objeto *this* ofrecerá ese tipo. Por lo tanto si el objeto que se recibe como parámetro en el método *equals* no es de ese tipo el resultado es *false*. Como caso particular si el objeto que se recibe como parámetro es null el resultado es false porque *null* no es instancia de ningún tipo aunque puede ser asignado como valor a cualquier objeto.

Como ejemplos veremos el tipo *Punto2D* y como segundo ejemplo el tipo *Racional*. Este tiene, entre otras que veremos, dos propiedades consultables: *Numerador* y *Denominador*. Ambas propiedades son de tipo entero. Este tipo nos servirá para manipular fracciones de la forma: *3/4, -5/2, …*

El tipo *Racional* será de la forma que aparece abajo. A lo largo del tema iremos completándolo.

```
public class Racional .. {
    ...
    Integer getNumerador();
    Integer getDenominador();
    . . .
}
```

Seguiremos las estrategia explicitada arriba: definir los detalles de la relación de equivalencia y pasar a implementar directamente los métodos relacionados con la igualdad: *equals, hashCode, toString*.

Supongamos que queremos diseñar el método *equals* en la clase *Racional* y que dos racionales $\frac{a}{b}$ y $\frac{c}{d}$ son iguales si se cumple $ad == bc$. Escogemos con representante canónico el racional simplificado y con denominador positivo. Esta misma estrategia se sigue en el tipo *Punto2D* pero en este es más fácil. Con la definición dada de igualdad entre puntos cada clase de equivalencia tiene un solo objeto que a su vez es el representante canónico. La implementación de *equals* para *Punto2D* y *Racional,* según es generada directamente por Eclipse, será:

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
            return true;
    if (obj == null)
            return false;
    if (!(obj instanceof Punto2D))
            return false;
    Punto2D other = (Punto2D) obj;
    if (x == null) {
            if (other.x != null)
                    return false;
```

```
    } else if (!x.equals(other.x))
          return false;
    if (y == null) {
          if (other.y != null)
                return false;
    } else if (!y.equals(other.y))
          return false;
    return true;
}
```

Y para el caso de Racional

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
          return true;
    if (obj == null)
          return false;
    if (!(obj instanceof Racional))
          return false;
    Racional other = (Racional) obj;
    if (denominador == null) {
          if (other.denominador != null)
                return false;
    } else if (!denominador.equals(other.denominador))
          return false;
    if (numerador == null) {
          if (other.numerador != null)
                return false;
    } else if (!numerador.equals(other.numerador))
          return false;
    return true;
}
```

Algunos comentarios al código anterior:

- En ambos casos se devuelve true si ambos objetos son idénticos y false si el que se recibe como parámetro es null.
- Si el objeto que recibe como parámetro no ofrece el tipo sobre el que hemos definido la relación de equivalencia el resultado es false.
- Podemos estar seguros que el casting a *Racional* y a *Punto2D* no disparará excepciones puesto que se hace después de comprobar que objeto ofrece ese tipo.

- Como las propiedades*, X e Y de Punto2D* son de tipo *Double* podemos usar el método *equals* para preguntar por su igualdad. Si su tipo hubiera sido *int* entonces este método no está disponible y tendríamos que usar el operador *==.*
- En el caso de Racional se ha usado la estrategia de consistente en que dos objetos son equivalentes si sus representantes canónicos tienen las mismas propiedades. Recordamos, para que todo cuadre, que cuando creamos un racional siempre guardamos únicamente las propiedades de su representante canónico: el racional simplificado.
- En el caso del tipo *Punto2D* y el tipo *Racional* tenemos que comprobar si sus propiedades son *null* o no y tomar la decisión adecuada.

Veamos ahora los otros dos métodos. Tal como hemos comentado arriba, por la implementación escogida, los atributos asociados a las propiedades *Numerador* y *Denominador* de *Racional* guardan los valores del representante canónico. Con esta decisión de implementación los métodos *getNumerador()* y *getDenominador()* devolverán el numerador y denominador del representante canónico en el caso del *Racional.* La implementación, en este caso, para *Punto2D* y *Racional* sería:

```
@Override
public String toString(){
          String s;
          s = "("+getX()+","+getY()+")";
          return s;
}
```

```
. . .
 @Override
public String toString(){
   String s;

   if(numerador!=null && denominador!=null && denominador==1){
         s = numerador.toString();
   } else {
         s = numerador+"/"+denominador;
   }
   return s;
}
. . .
```

Algunos comentarios:

- La cadena resultante en el método *toString* se debe calcular a partir de los resultados devueltos por los correspondientes *toString* de las propiedades involucradas en la igualdad y, posiblemente, otras propiedades derivadas de las mismas.
- El código calcula una cadena que se compone del numerador en un primer momento. Esto lo consigue invocando el método *toString* del tipo *Integer.* Si el denominador es distinto de uno entonces añade además el símbolo "/" y el denominador. En este caso no es necesario invocar el método *toString* porque cuando un tipo objeto está dentro de una operación con otros operandos de tipo *String* el compilador llama automáticamente a ese método. Es decir, en ese contexto el compilador convierte automáticamente el tipo dado a *String.* Por eso es tan importante un buen diseño del mismo.

Y el diseño del *hashCode* tal como es generado por Eclipse es:

```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((x == null) ? 0 : x.hashCode());
    result = prime * result + ((y == null) ? 0 : y.hashCode());
    return result;
}
```

```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result
           + ((denominador == null) ? 0 : denominador.hashCode());
    result = prime * result
           + ((numerador == null) ? 0 : numerador.hashCode());
    return result;
}
```

Algunos comentarios:

- El *hashCode* se calcula a partir de las propiedades, del representante canónico, involucradas en la igualdad. En el caso de Racional la implementación anterior es adecuada sólo si los métodos *getNumerador* y *getDenominador* devuelven las propiedades del representante canónico. Si hay más de una propiedad involucrada podemos seguir la siguiente regla:
- El *hashCode* resultante es igual al *hashCode* de la primera propiedad más *31* por el *hashCode* de la segunda propiedad más *31*31* por el *hashCode* de la tercera propiedad etc. Alternativamente al 31 se podría haber escogido otro número primo no muy grande. El escoger un número primo hace que el *hashCode* calculado se disperse adecuadamente. Es decir, que los *hashCode* de dos objetos distintos sean en la mayoría de los casos distintos.
- Como vemos el código comprueba si una propiedad es *null* o no antes de pedir su *toString* o su *hashCode*. Podemos asumir que si el valor de la propiedad es *null* su *hashCode* es cero y su representación directamente *null*.

En cualquier caso la mayoría de los entornos, y Eclipse en particular, pueden generar automáticamente el código para los métodos *equals* y *hashCode,* y propuestas razonables para *toString*, si tenemos claro las propiedades de la relación de equivalencia. Es, siempre que sea posible, la mejor manera de hacerlo.

Uso de los métodos anteriores.

```
Racional r1 = Racional.create(3,4);
Racional r2 = Racional.create(3,4);
if(r1.equals(r2) && r1!=r2)
      mostrar(r1 + " y " + r2 + " son iguales pero no idénticos");
```

En el código anterior preguntamos si dos objetos son iguales pero no idénticos. Esto se consigue combinando el método *equals* con el operador == o su contrario *!=*. Los objetos *r1, r2* son iguales, tienen las mismas propiedades pero no son idénticos porque se han creado en dos usos distintos del operador *new*.

La implementación del orden natural del tipo Racional lo hacemos de la forma:

```
@Override
public int compareTo(Racional r) {
```

```
        if(r==null || this.getDenominador() ==null
                    || this.getNumerador() == null
                    || r.getDenominador() ==null
                    || r.getNumerador() == null ){
             throw new NullPointerException();
        }
        Integer prod1;
        Integer prod2;
        prod1 = getNumerador()*r.getDenominador();
        prod2 = getDenominador()*r.getNumerador();
        return prod1.compareTo(prod2);
}
```

En general para implementar el método *compareTo* usaremos los métodos *compareTo* de las propiedades involucradas en la igualdad o algunas otras derivadas como en el caso anterior. Como hemos comentado antes es importante que las propiedades se evalúen sobre le representante canónico. Esto se consigue si en la implementación, como es el caso, los atributos que guardan las propiedades *Numerador* y *Denominador* almacenan los valores de estas propiedades para el representante canónico.

El tipo *Punto2D* no suele dotar se de un orden natural pero en le diseño si lo hemos dotamos de uno. Un orden natural adecuado puede ser comparar en primer lugar por una propiedad elegida arbitrariamente, si resulta cero comparar por la segunda propiedad, etc.

```
@Override
public int compareTo(Punto2D p) {
        if(p==null || this.getX() ==null
                    || this.getY() == null
                    || p.getX() ==null || p.getY() == null ){
             throw new NullPointerException();
        }
        int r = getX().compareTo(p.getX());
        if(r==0){
               r = getX().compareTo(p.getX());
        }
        return r;
}
```

Como en el caso del tipo *Punto2D* hay muchos tipos cuyo orden natural se establece secuencialmente ordenando primero por una propiedad, simple o derivada, si son iguales ordenando por una segunda, etc.

El orden natural para el tipo Racional es:

```
@Override
public int compareTo(Racional r) {
	if(r==null || this.getDenominador() ==null
		|| this.getNumerador() == null
		|| r.getDenominador() ==null || r.getNumerador() == null ){
		throw new NullPointerException();
	}
	Integer prod1;
	Integer prod2;
	prod1 = getNumerador()*r.getDenominador();
	prod2 = getDenominador()*r.getNumerador();
	return prod1.compareTo(prod2);
}
```

Los órdenes alternativos para el tipo Racional, uno según el numerador y otro según del denominador del representante canónico, se pueden implementan en los métodos correspondientes:

```
public int compareByNumerador(Racional r) {
	if(r==null || this.getNumerador() ==null
		throw new NullPointerException();
	}
	return getNumerador().compareTo(r.getNumerador());
}

public int compareByDenominador(Racional r) {
	if(r==null || this.getDenominador() ==null
		throw new NullPointerException();
	}
	return getDenominador().compareTo(r.getDenominador());
}
```

Como podemos comprobar la implementación no es consistente con la igualdad definida para el tipo *Racional*. Las clases de equivalencia definidas implícitamente por este orden (dos

objetos son equivalentes si el método compare devuelve cero) no son las mismas que las definidas por la igualdad. En el orden por numerador, en la segunda implementación, dos racionales son equivalentes si sus representantes canónicos tienen el mismo numerador. Los órdenes alternativos no están obligados a ser consistentes con la igualdad pero es necesario que esta inconsistencia quede documentada en el diseño del orden alternativo en cuestión.

Otra implementación posible de los órdenes alternativos es:

```
public int compareByNumerador(Racional r) {
       if(r==null || this.getNumerador() ==null
                  throw new NullPointerException();
       }
       int s = getNumerador().compareTo(r.getNumerador());
       if(s==0){
              s = this.compareTo(r);
       }
       return s;
}

public int compareByDenominador(Racional r) {
       if(r==null || this.getDenominador() ==null
                  throw new NullPointerException();
       }
       int s = getDenominador().compareTo(r.getDenominador());
       if(s==0){
              s = this.compareTo(r);
       }
       return s;
}
```

Como podemos ver esta implementación si es consistente con la igualdad. En general cualquier orden de un tipo que tenga orden natural puede hacerse, si se desea, compatible con la igualdad. La idea es la utilizada arriba: primero se compara la propiedad o propiedades que definen el orden y si resultara cero se usa el orden natural.

```
Racional r1 = Racional.create(1,2);
Racional r2 = Racional.create(3,2);
if(r1.compareTo(r2) > 0)
   mostrar(r1 + " es mayor que el de " + r2);
```

En general la implementación del método *compare* es similar a la del método *compareTo*. En el primer caso se comparan propiedades de los objetos *o1, o2* y en el segundo del objeto *this* con otro objeto *o.*

2.3 Los constructores con un parámetro de tipo String

Con el método *toString()* convertimos un objeto en su representación como cadena de caracteres. Muchos tipos necesitan construir objetos a partir de la información contenida en una cadena de caracteres. Con esta funcionalidad pretendemos poder construir, por ejemplo, objetos de tipo Racional a partir de cadenas de la forma "-3/4". Para poder implementar esta funcionalidad necesitamos que el tipo correspondiente tenga un constructor que tome como único parámetro una cadena de caracteres. Por lo tanto debemos ver una guía para implementar constructores que tomen una cadena de caracteres con el formato adecuado para el tipo. Los tipos envoltura proporcionados en la API de Java ya vienen con constructores de ese tipo.

Ejemplos:

```
Integer a = new Integer("+35");
Double b = new Double("-4.57");
```

Estos constructores disparan la excepción *NumberFormatException* si el formato no es el adecuado. En otros casos que diseñemos se disparará la excepción *IllegalArgumentException* si el formato no es el adecuado.

Para diseñar constructores para tipos que tomen una cadena necesitamos analizar la cadena y partirla en trozos adecuados. En el caso del tipo racional la cadena *"-3/4"* tenemos que partirla en las subcadenas "-3" que dará lugar al numerador y "4" que dará lugar al denominador. Esto lo podemos hacer con el método *split* del tipo *String*.

Si queremos que el tipo Racional, junto con la funcionalidad ofrecida más arriba, pueda crear objetos a partir de una cadena de caracteres entonces Racional debe tener un constructor de de la forma:

```
public class Racional {
```

```
...

String[] sp = s.split("[/]");
       Integer ne = sp.length;
       if(ne>2) throw
          new IllegalArgumentException("Cadena con formato no válido");
       numerador = new Integer(sp[0].trim());
       if(ne==1)
              denominador = 1;
       else
              denominador = new Integer(sp[1].trim());
       normaliza();
}

. . .
```

En el código anterior se han eliminado los espacios en blanco adicionales con el método *trim*.

2.4 Implementación de propiedades derivadas modificables

Como hemos visto un tipo puede tener propiedades derivadas. Es decir propiedades cuyo valor puede ser calculado a partir del valor de otras propiedades. Lo usual es que tomemos la decisión de escoger atributos para guardar el valor solamente de las propiedades básicas. Pero hay casos dónde es mejor escoger atributos también para las propiedades derivadas. Esto es conveniente especialmente cuando la propiedad derivada va a ser consultada muy frecuentemente y queremos impedir el cálculo repetido del valor de la propiedad. Pero la decisión de escoger atributos para las propiedades básicas y derivadas nos obliga a mantener el invariante que liga los valores de unas propiedades con otras. Esto es importante en el diseño de los constructores y en cada uno de los métodos modificadores si el tipo es mutable.

Veamos el tipo Vector2D cuyo diseño propusimos en el capítulo anterior y que reproducimos aquí:

Vector2D

Descripción: Vector en el plano. Mutable

Propiedades:

- *X: Double*, consultable, modificable.
- *Y: Double*, consultable, modificable
- *Angulo: Double,* consultable, modificable. Angulo en radianes

- *Modulo: Double,* consultable, modificable
- *AnguloEnGrados*: Double, consultable.
- *Ortogonal: Vector2D,* consultable. Vector con el mismo módulo y girado 90 grados
- *Unitario: Vector2D,* consultable. Vector con el mismo ángulo y módulo 1.
- *Opuesto: Vector2D,* consultable.

Invariante:

- $Modulo = \sqrt{X^2 + Y^2}$
- $Angulo = \tan^{-1}\frac{Y}{X}$

Propiedades relacionadas con la igualdad:

- *Criterio de igualdad*: dos objetos de este tipo son iguales si tienen las mismas propiedades.
- *Representación como cadena*: valor de las propiedades X e Y separadas por comas y entre paréntesis.

Operaciones:

- *Vector2D proyectaSobre(Vector2D v):* Observadora. Devuelve le producto escalar de *this* por el vector unitario em la dirección de v.
- *Vector2D suma(Vector2D v):* Observadora.
- *Vector2D multiplica(Double factor):*Observadora. Devuelve *(factor*X, factor*Y).*
- *Double multiplicaVectorial(Vector2D v):* Observadora. Devuelve *this.X*v.Y-this.Y*v.X*
- *Double multiplicaEscalar(Vector2D v):* Observadora. Devuelve *this.X*v.X+this.Y*v.Y*
- *Vector2D rota(Double angulo).* Observadora

Factoría:

- *Vector2D createCartesiano(Double x, Double y)*
- *Vector2D create(Punto2D p)*
- *Vector2D create(Vector2D p)*
- *Vector2D createPolarEnGrados(Double modulo, Double angulo)*
- *Vector2D createPolarEnRadianes(Double modulo, Double angulo)*

De este tipo nos interesa fundamentalmente el hecho de que tenga propiedades derivadas que queremos que sean modificables. Decidimos escoger atributos para las propiedades *X, Y, Modulo, Angulo.* Esto nos obliga a mantener el invariante en los constructores y métodos *set.*

```
public class Vector2D {

	public static Vector2D createCartesiano(Double x, Double y) {
		return new Vector2D(x, y);
	}

	public static Vector2D create(Punto2D p) {
		return new Vector2D(p.getX(), p.getY());
	}

	public static Vector2D create(Vector2D p) {
		return new Vector2D(p.getX(), p.getY());
	}

	public static Vector2D createPolarEnGrados(Double modulo,Double angulo){
		return createPolarEnRadianes(modulo, Math.toRadians(angulo));
	}

	public static Vector2D createPolarEnRadianes(
			Double modulo,Double angulo){
		Preconditions.checkArgument(modulo >=0);
		return createCartesiano(
				modulo*Math.cos(angulo),modulo*Math.sin(angulo));
	}

	private Double x;
	private Double y;
	private Double modulo;
	private Double angulo;

	private Vector2D(Double x, Double y) {
		super();
		this.x = x;
		this.y = y;
		this.modulo = Math.hypot(x, y);
		this.angulo = Math.atan2(y, x);
	}

	public Double getX() {
```

```
        return x;
    }

    public Double getY() {
        return y;
    }

    public void setX(Double x) {
        this.x = x;
        this.modulo = Math.hypot(x, y);
        this.angulo = Math.atan2(y, x);
    }

    public void setY(Double y) {
        this.y = y;
        this.modulo = Math.hypot(x, y);
        this.angulo = Math.atan2(y, x);
    }

    public Double getModulo() {
        return modulo;
    }

    public Double getAngulo() {
        return angulo;
    }

    public void setAngulo(Double angulo) {
        this.angulo = angulo;
        this.x = modulo*Math.cos(angulo);
        this.y = modulo*Math.sin(angulo);
    }

    public void setModulo(Double modulo) {
        this.modulo = modulo;
        this.x = modulo*Math.cos(angulo);
        this.y = modulo*Math.sin(angulo);
    }

    ...
```

Como podemos ver hemos decidido que al modificar la X mantenemos constante la Y y calculamos el valor de *Angulo y Modulo*. Algo similar hacemos cuando modificamos *Angulo o Modulo.* La precondición necesaria para los posibles valores del módulo se comprueba en la

factoría. Solamente hemos diseñado un constructor y hemos delegado en la factoría otras combinaciones posibles de los valores de las propiedades para construir objetos.

2.5 Implementación de tipos Inmutables.

Los objetos inmutables son simplemente objetos cuyo estado no se puede cambiar después de la construcción. Ejemplos de objetos inmutables son *String* e *Integer*. Un tipo debería ser inmutable a menos que haya una muy buena razón para que sea mutable. Si un tipo no puede ser hecho inmutable es buena técnica de diseño limitar su mutabilidad tanto como sea posible.

Es conveniente seguir algunas pautas para implementar un tipo inmutable.

- Garantizar la clase no se puede modificar: hacer la clase final, o utilice factorías estáticas y mantener constructores privados
- Hacer que los atributos privados y final.
- No proporciona ningún método que puede cambiar el estado del objeto de ninguna manera
- Si la clase tiene algún atributo mutable, entonces debe ser copiados de forma defensiva cuando pasan entre el tipo y su llamador

Un ejemplo podría ser el tipo Alarma diseñado como inmutable:

```
public final class Alarma implements Comparable<Alarma>{

	public static Alarma create(Calendar fechaAlarma) {
		return new Alarma(fechaAlarma);
	}

	public static Alarma create(Alarma a) {
		return new Alarma(a);
	}

	private final Calendar fechaAlarma;

	private Alarma(Calendar fechaAlarma) {
		checkFechaAlarma(fechaAlarma);
		this.fechaAlarma = (Calendar) fechaAlarma.clone();
	}

	private Alarma(Alarma a) {
		this.fechaAlarma = (Calendar) a.fechaAlarma.clone();
	}
```

```
        private void checkFechaAlarma(Calendar fechaAlarma) {
                Calendar ahora = Calendar.getInstance();
                if (fechaAlarma.before(ahora)) {
                        throw new IllegalArgumentException(
                  "Alarma.checkFechaAlarma::
                   La fecha de la alarma no puede ser anterior al momento actual");
                }
        }

        public Calendar getFechaAlarma(){
                return (Calendar) fechaAlarma.clone();
        }
        ...
}
```

3. Reutilización de código: Herencia

A la hora de implementar un tipo puede es oportuno, si se puede, reutilizar el código de otros tipos ya implementados.

Hay diferentes formas de reutilizar código:

- *Uso*: Al implementar una clase podemos declarar variables de tipos ya implementados. Decimos que estamos usando esos tipos.
- *Herencia de Clases*: Cuando una clase extiende otra tiene disponible el código de los métodos de la clase padre.
- *Composición de Clases*: Cuando implementamos una clase podemos declarar uno o varios objetos privados de otros tipos ya implementados. Tenemos entonces disponibles todos los métodos de los atributos declarados.

En general el uso de un tipo puede hacerse sin restricciones. Sin embargo el uso de la herencia o la composición como mecanismos de reutilización si requiere algunos comentarios adicionales. La herencia es un mecanismo de reutilización sencillo pero no recomendable en muchos casos. La composición es, en general, más recomendable que la herencia como mecanismo de reutilización de código.

La herencia, como mecanismo de reutilización de código, solo es conveniente cuando vamos a implementar un tipo S que es un subtipo de T del que ya disponemos de una implementación en una clase dada. En el resto de los casos se recomienda usar composición de clases. Es conveniente insistir en que si una clase B hereda (*extends*) de otra A entonces el tipo definido por B es un subtipo (con la definición dada en el capítulo anterior) del tipo definido por A y por lo tanto los objeto de B pueden ocupar el sitio donde se espera un objeto de tipo A.

3.1 Detalles de la herencia entre clases

Una clase hereda de otra cuando consta en la cabecera de la misma la palabra reservada **extends** seguida del nombre de la clase de la que se hereda. Cada clase sólo puede heredar de una única clase.

Si la clase B hereda de la clase A, implica que todos los atributos y métodos declarados **public** o **protected** contenidos en la clase A pasan a estar disponibles en la clase B.

```
public class A {
       public String atr1;
       protected Integer atr2;
       private Boolean atr3;

       public void metodo1(){…}
       private void metodo2(){…}
       protected void metodo3(){…}
}
```

```
public class B extends A{
       private String atr4;

       public void metodo4();
}
```

Atributos accesibles en la clase *B: atr1, atr2 y atr4.* Métodos accesibles en la clase *B: metodo1(), metodo3() y metodo4().*

Una clase B que herede de otra A puede reutilizar, también sus constructores. Para ello la clase invocará en la primera línea del cuerpo de su constructor al constructor apropiado de la clase

padre. Esto se consigue mediante el uso de la palabra reservada **super**, seguida de los parámetros reales separados por comas y encerrados entre paréntesis. Igualmente una clase puede reutilizar otros constructores propios ya definidos. Esto se consigue mediante el uso de la palabra reservada **this**, seguida de los parámetros reales separados por comas y encerrados entre paréntesis. La palabra reservada **this** tiene otros usos. Designa el objeto actual por una parte y por otra **this.a**, **this.m(e)** designan el atributo a y la invocación al método m de la clase que estamos definiendo.

```
public class A {
	public String atr1;
	protected Integer atr2;
	private Boolean atr3;

	public A(String a1, Integer a2,  Boolean a3){…}
	(…)
}
```

```
public class B extends A{
	private String atr4;

	public B(String a1, Integer a2,  Boolean a3, String a4){
		super(a1,a2,a3);
		atr4=a4;
	}
	(…)
}
```

Asimismo, los métodos heredados pueden ser **redefinidos**, siendo común utilizar en dicha redefinición una llamada al método de la clase padre. Para invocar los métodos de la superclase se utiliza la palabra reservada **super** seguida de un punto, el nombre del método a invocar y los parámetros reales entre paréntesis.

```
public class A {
	(…)
	public void metodo1(){
		super.metodo1();
		(…)
	}
}
```

Ejemplo de reutilización mediante herencia

```
public class Persona {
        (…)

        public Persona(String nombre, String apellidos){
                (…)
        }
        (…)

        public String toString(){
                return getNombre()+" " +getApellidos();
        }
}
```

```
public class Empleado extends Persona {

        private Integer añosDeServicio;
        private String departamento;

        public Empleado(String nombre, String apellidos, Integer años,
           String dep){
                super(nombre,apellidos);
                añosDeServicio=años;
                departamento=dep;
        }

        public Integer getAñosDeServicio(){
                return añosDeServicio;
        }

        public String getDepartamento(){
                return departamento;
        }

        public String toString(){
                return super.toString()+" "+getAñosDeServicio()
                   + " " + getDepartamento();
        }
}
```

Como podemos ver el tipo definido por la clase *Empleado* es un subtipo de *Persona*. Esto hace que la herencia sea adecuada para implementar Empleado reutilizando Persona.

3.2 Igualdad en tipos y subtipos

Como hemos recomendado previamente la clase que implementa el subtipo podemos implementarla reutilizando mediante herencia la que implementa el tipo padre.

Posteriormente tendremos que implementar los nuevos métodos del subtipo y volver a implementar los métodos relacionados con la igualdad (si es más estricta que la del tipo). Tal como explicamos en el capítulo anterior

Como vimos en el capítulo anterior cuando definimos un subtipo *S* de otro *T* podemos tener dos relaciones de equivalencia: *eq1* una relación de equivalencia sobre objetos de tipo *T,* y otra *eq2* una relación de equivalencia entre objetos de tipo *S.* Pero los objetos de tipos *S* también son de tipo *T*. Además en una población de objetos de tipo *T* puede haber algunos de tipo *S*. Para tener en cuenta esta situación, como explicamos en le capítulo anterior, definimos una nueva relación de equivalencia a partir de *eq1* y *eq2*. Según esta nueva relación de equivalencia dos objetos son iguales solamente si:

- Ambos son de tipo S y tienen iguales la propiedades definidas en *eq2*
- Ambos son de tipo T pero ninguno de tipo S y tienen iguales las propiedades definidas en *eq1*.

De lo anterior concluimos que un objeto de tipo S no será igual a otro de tipo T que no sea, a su vez, también de tipo de S. Las clases de equivalencia de la nueva relación de equivalencia extendida estarán formadas por las clases definidas sobre objetos estrictamente de tipo T pero no de S por *eq1* más las clases de equivalencia definidas por *eq2* sobre los objetos de tipo S. E entorno Eclipse genera automática una implementación adecuada para los métodos *equals, hashCode y toString.*

```
public class Pixel2D extends Punto2D {

      private Color color;
```

```
public Pixel2D() {
	super();
	this.color = Color.RED;
}

public Pixel2D(Double x, Double y, Color color) {
	super(x, y);
	this.color = color;
}

public Color getColor() {
	return color;
}

public void setColor(Color color) {
	this.color = color;
}

@Override
public String toString() {
	return "Pixel2D [color=" + color + ","+ super.toString()+ "]";
}

@Override
public int hashCode() {
	final int prime = 31;
	int result = super.hashCode();
	result = prime * result
		+ ((color == null) ? 0 : color.hashCode());
	return result;
}

@Override
public boolean equals(Object obj) {
	if (this == obj)
		return true;
	if (!super.equals(obj))
		return false;
	if (!(obj instanceof Pixel2D))
		return false;
	Pixel2D other = (Pixel2D) obj;
	if (color == null) {
		if (other.color != null)
			return false;
	} else if (!color.equals(other.color))
```

```
                    return false;
                return true;
        }

}
```

4. Reutilización de código: Composición

En la composición de clases se suele usar una técnica conocida como *Delegación de Métodos*. Es un mecanismo no disponible usualmente en todos los lenguajes de programación orientados a objeto pero fácil de implementar. Esencialmente consiste en declarar (en una clase) un atributo (usualmente privado) de un tipo disponible para ser usado (ya definido e implementado) y definir el cuerpo de uno de los métodos de la clase como la invocación a uno de los métodos del atributo declarado (hablaremos de delegar un método en el objeto dado). Por lo tanto el término *Composición de Clases* suele usarse a la vez que la *Delegación de Métodos*.

Composición de clases

Como hemos explicado arriba la composición de clases consiste en construir una clase a partir de la funcionalidad parcial o total ofrecida por otras. Para implementarla se declaran tantos atributos privados como clases a componer y se delegan los métodos a implementar de la nueva clase en alguno de los objetos declarados. La relación de composición no crea relación tipo-subtipo entre la clase construida y las reutilizadas. Esto hace que este mecanismo sea el más indicado para la reutilización de código en la mayoría de los casos. Generalmente se prefiero frente al uso de herencia.

Sea la clase A que queremos implementar componiendo las clases C1, C2, C3. Sean m1, n1, … los métodos a reutilizar de C1 (usualmente serán un subconjunto de todos los métodos públicos que ofrece). Igualmente para C2, C3. El esquema de implementación es:

```
public class A {
        private C1 c1;
        private C2 c2;
        private C3 c3;
```

```
        …
        public A(){
                c1 = new C1();
                c2 = new C2();
                c3 = new C3();
                …
        }
        public tm1 m1() { c1.m1();}
        public tn1 n1() { c1.n1();}
        …
        public tm2 m2() { c2.m2();}
        …

}
```

Ejemplo de reutilización mediante composición

Veamos el tipo Fecha que presenta características interesantes a tener en cuenta. Es un tipo inmutable lo que quiere decir que no tiene **métodos modificadores**. Es decir no tiene métodos que cambien el estado del objeto.

Tipo Fecha:

- *Propiedades:*
    - *DiaSemana*, entero entre 1 y, dependiendo del mes, 30, 31, 28 ó 29, consultable
    - *DiaSemanaCadena*, *String*, solo consultable, derivada
    - *Mes*, entero entre 0 y 11 inclusive, solo consultable
    - *MesCadena*, String, derivada
    - *Año*, entero, solo consultable
    - *Bisiesto, boolean,* solo consultable

- *Propiedades de la igualdad:*
    - *Orden natural*: el usual entre fechas
    - *Criterio de Igualdad*: si tienen el mismo día, mes y año
    - *Representación como cadena*: *DiaSemanaCadena*, " a ", Día, " de ", *Mes*, " de ", Año.

- *Operaciones:*
    - *Suma*, parámetro un entero positivo. Devuelve una fecha posterior en el número de días indicado por el parámetro
    - *Resta*, parámetro un entero positivo. Devuelve una fecha anterior en el número de días indicado por el parámetro
    - *Resta*, parámetro una fecha. Devuelve un entero, positivo, negativo o cero, con el número de días transcurridos entre las fechas representadas por el objeto y el parámetro

- *Factoría:*
    - *Fecha create(Integer d, Integer m, Integer a)*
    - *Fecha create().*
    - *Fecha create(String s)*
    - *Fecha create(Fecha f)*

Casos de prueba para el tipo Fecha

Método	this	p	r	Excepción
Constructores		(29,1,2009)		IllegalArgumentException
		(29,1,2000)	(29,2,2000)	
		“1/1/2009”	(1,2,2009)	
getDia	(14,2,1951)		14	
getMes	(14,6,1999)		6	
getAño	(10,2,2000)		2000	
getDiaSemana	(8,9,2009)		“Jueves”	
	(1,0,2000)		“Sábado”	
	(1,2,2009)		“Domingo”	
	(1,0,1900)		“Lunes”	
	(2,2,2011)		“Miércoles”	
getMesCadena	(10,11,2002)		“Diciembre”	
suma	(26,8,2009)	7	(3,9,2009)	
resta	(26,8,2009)	(1,0,2009)	268	
	(8,9,2009)	10	(28,8,2009)	

toString	(8,9,2009)		"Jueves a 8 de Octubre de 2009"	
	(1,0,1900)		"Lunes a 1 de Enero de 1900"	

La implementación de este tipo puede hacerse reutilizando otro ya ofrecido por el API de Java. Es el tipo *GregorianCalendar* que construye fechas del calendario gregoriano. La reutilización la hacemos por composición puesto que ambos tipos no tienen ninguna relación de subtipado. Los detalles se ofrecen abajo. El resto de los detalles se deja como ejercicio.

Hay que tener en cuenta que el tipo *GregorianCalendar* numera los meses de 0 en adelante hasta 11. Tampoco produce excepciones si la fecha no existe por lo que hay que implementar el código que dispare la excepción en caso de que la fecha sea inválida. Por otra parte el nombre de los meses y días de la semana en español vienen proporcionados por un objeto de tipo *Locale* que se usa en los métodos *getDiaSemanaCadena.* Pero no es posible encontrar los nombres de los meses de esa forma por lo que usamos un *array* con los nombres de los meses en español.

Por último la implementación de *equals, hashCode* y *compareTo* se delegan en la variable privada *calendar*. Abajo se pueden ver los detalles relevantes. Para comprenderlos hay que tener en cuenta los detalles de la clase GregorianCalendar ofrecida por Java.

```
public class Fecha implements Comparable<Fecha> {


	public static Fecha create(Integer d, Integer m, Integer a) {
		return new Fecha(d, m, a);
	}

	public static Fecha create() {
		return new Fecha();
	}

	public static Fecha create(String s) {
		return new Fecha(s);
	}
```

```
public static Fecha create(Fecha f) {
	return new Fecha(f.getDiaMes(), f.getMes(), f.getAño());
}

private GregorianCalendar calendar;


public boolean esValida(Integer d, Integer m, Integer a){
	boolean r = true;
	try{
		Calendar c = new GregorianCalendar(a,m-1,d);
		c.setLenient(false);
		c.getTime();
	}
	catch(IllegalArgumentException e){
		r = false;
	}
	return r;
}

private Fecha(Integer d, Integer m, Integer a) {
	super();
	if(!esValida(d, m, a))
	     throw new IllegalArgumentException("Fecha Inválida");
	this.calendar = new GregorianCalendar(a,m-1,d);
}

private Fecha() {
	super();
	this.calendar = new GregorianCalendar();
}

private Fecha(GregorianCalendar g) {
	super();
	this.calendar = g;
}

 private Fecha(String s) {
	 DateFormat formatter =  new SimpleDateFormat("dd/MM/yy");
	 formatter.setLenient(false);
	 this.calendar = new GregorianCalendar();
	 try {
		Date date = (Date)formatter.parse(s);
		this.calendar.setTime(date);
	} catch (Exception e) {
```

```
                throw new IllegalArgumentException(
                    "Formato de fecha inválido o fecha no válida");
            }
        }

        public Integer getDiaSemana(){
            return calendar.get(Calendar.DAY_OF_WEEK);
        }

        public Integer getDiaMes(){
            return calendar.get(Calendar.DAY_OF_MONTH);
        }

        public Integer getDiaAño(){
            return calendar.get(Calendar.DAY_OF_YEAR);
        }

        public Integer getMes(){
            return calendar.get(Calendar.MONTH);
        }

        public Integer getAño(){
            return calendar.get(Calendar.YEAR);
        }

        public String getMesCadena(){
            return  calendar.getDisplayName(
                 Calendar.MONTH,Calendar.LONG, new Locale("ES"));
        }

        public String getDiaSemanaCadena(){
            return calendar.getDisplayName(
                 Calendar.DAY_OF_WEEK,Calendar.LONG, new Locale("ES"));
        }

        public Fecha suma(Integer a){
            GregorianCalendar ncalendar=(GregorianCalendar)calendar.clone();
            ncalendar.add(Calendar.DAY_OF_MONTH, a);
            return new Fecha(ncalendar);
        }

        public Fecha resta(Integer a){
            return suma(-a);
        }

        private static long millisOfDay = 1000*24*60*60;
```

```
        public Integer resta(Fecha f){
                long r;
                long dif = this.calendar.getTimeInMillis()
                    -f.calendar.getTimeInMillis();
                r =  dif/millisOfDay;
                return (int) r;
        }

        public boolean esBisiesto(){
                return this.calendar.isLeapYear(getAño());
        }


        @Override
        public String toString() {
                return getDiaSemanaCadena()+" a "+getDiaMes()
                    +" de "+getMesCadena()+" de "+getAño();
        }

        ...

}
```

5. Algunos ejemplos

Veamos como ejemplo el diseño de una *Recta2D* y un posible subtipo el *Semiplano2D*. Ambos tipos ya los hemos propuesto como ejercicios en el capítulo anterior. El diseño de los tipos es:

Recta2D …

Descripción: Recta en el plano. Inmutable. Ecuación de la recta $Ax + By + C = 0$.

Propiedades:

- *A: Double*, consultable.
- *B: Double*, consultable.
- *C: Double,* consultable.
- *Punto: Punto2D*, consultable.
- *Vector: Vector2D, consultable.*
- *Angulo: Double,* consultable. Angulo en radianes

- *AnguloEnGrados*: Double, consultable.
- *DistanciaAlOrigen*: Double, consultable.
- *Distancia(Punto2D p)*: Double, consultable.
- *Paralela(Punto2D p)*: Recta2D.
- *Perpendicular(Punto2D p)*: Recta2D.

Invariante:

- $Vector = (B, -A)$
- $Distancia(p) = \frac{Ax+By+C}{\sqrt{A^2+B^2}}$

Propiedades relacionadas con la igualdad:

- *Criterio de igualdad*: dos objetos de este tipo son iguales si tienen la misma distancia al origen (teniendo en cuenta el signo) y la diferencia de sus ángulos es 0 o 180.
- *Representación como cadena*: de la forma $Ax + By + C = 0$

Operaciones:

- *boolean contienePunto(Punto2D p):* Observadora. Si $Ax + By + C = 0$
- *Punto2D cortaA(Recta2D v):* Observadora.

Factoría:

- *Recta2D createEnRadianes(Punto2D p, Double angulo).*
- *Recta2D createEnGrados(Punto2D p, Double angulo).*
- *Recta2D create(Punto2D p, Double angulo).*
- *Recta2D create(Punto2D p, Vector2D d).*
- *Recta2D create(Punto2D p1, Punto2D p2).*
- *Recta2D create().* Devuelve el eje X.

Como se sabe una recta en el plano es el conjunto de puntos del plano que cumplen $Ax + By + C = 0$. Una recta tiene por lo tanto las propiedades *A, B, C.* Pero $\lambda(Ax + By + C) = 0,\ \lambda \neq 0$ representan recta iguales. Es necesario escoger un conjunto de propiedades definan un representante canónico.

Una recta viene definida, también, por un punto *p* y un vector *v* que indica su dirección. Alternativamente por un punto *p* y el ángulo α que forma con el eje X. La recta tiene otras propiedades como su distancia al origen que viene dada por

$$DistanciaAlOrigen = \frac{C}{\sqrt{A^2 + B^2}}$$

Esta definición de distancia incluye el signo que resulta del cálculo anterior. A su vez la dirección de una recta viene definida por un vector o alternativamente por un ángulo. Hay muchos vectores posibles para definir la dirección de una recta. Son los vectores de la forma:

$$Vector = \lambda(B, -A), \qquad \lambda \neq 0$$

El ángulo que forma la recta con el eje *X* es el mismo que forma su vector director:

$$\alpha = \tan^{-1}\frac{-A}{B}, \quad con - \pi \leq \alpha \leq \pi$$

Dos rectas iguales tienen la misma distancia al origen (definida por la fórmula anterior) y el el mismo ángulo α con el eje X. Pero las rectas con la misma distancia al origen y ángulos α y $\alpha + \pi$ son también iguales.

Escogemos, por todo lo anterior, la distancia al origen (con signo) y el ángulo definido como:

$$Angulo = f(x) = \begin{cases} \alpha, & \alpha \geq 0 \\ \alpha + \pi, & \alpha < 0 \end{cases}$$

Por último a la recta le exigimos que implemente el tipo *Objeto Geométrico*.

```
public interface ObjetoGeometrico2D {
	ObjetoGeometrico2D rota(Punto2D p, Double angulo);
	ObjetoGeometrico2D traslada(Vector2D v);
}
```

El código puede verse abajo. Se han utilizado algunos métodos para simplificar la presentación de números reales.

```
public static String simplify(Double d){
		String s = "";
```

```
            if(d==1.){
                  s+="+";
            }else if(d==-1.){
                  s+="-";
            }else if(d>0.){
                  s+="+"+d.toString();
            }else if(d!=0.){
                  s+=d.toString();
            }
            return s;
}
```

El código resultante es:

```
public class Recta2D implements ObjetoGeometrico2D {
      public static Recta2D createEnGrados(Punto2D p, Double angulo) {
            return new Recta2D(p, Math.toRadians(angulo));
      }

      public static Recta2D create(Punto2D p, Double angulo) {
            return new Recta2D(p, angulo);
      }

      public static Recta2D create(Punto2D p, Vector2D d) {
            return new Recta2D(p, d);
      }

      public static Recta2D create(Punto2D p1, Punto2D p2) {
            return new Recta2D(p1, p2);
      }

      public static Recta2D create() {
            return new Recta2D();
      }

      public static Recta2D create(Recta2D r) {
            return new Recta2D();
      }

      private Double a;
      private Double b;
      private Double c;
```

```
	private Vector2D vector;
	private Punto2D punto;
	private Double angulo;
	private Double distanciaAlOrigenConSigno;

	private Recta2D() {
		this(Punto2D.create(),Vector2D.createPolarEnRadianes(1.,0.));
	}

	private Recta2D(Punto2D p, Double angulo) {
		this(p,Vector2D.createPolarEnRadianes(1., angulo));
	}

	protected Recta2D(Punto2D p, Vector2D vector) {
		this.punto = Punto2D.create(p);
		this.vector = Vector2D.create(vector);
		this.angulo = Math.atan2(vector.getY(), vector.getX());
		this.angulo = this.angulo< 0 ? this.angulo+Math.PI : this.angulo;
		this.a = this.vector.getY();
		this.b = -this.vector.getX();
		this.c = -(a*punto.getX()+b*punto.getY());
		this.distanciaAlOrigenConSigno
		    = getDistancia(Punto2D.getOrigen());
	}

	private Recta2D(Punto2D p1, Punto2D p2) {
		this(p1,p2.minus(p1));
	}

	protected Recta2D(Recta2D r) {
		this(r.getPunto(),r.getVector());
	}

	public Vector2D getVector() {
		return vector;
	}

	public Punto2D getPunto() {
		return punto;
	}

	public Double getAngulo() {
		return angulo;
	}

	public Double getA() {
```

```
		return a;
	}

	public Double getB() {
		return b;
	}

	public Double getC() {
		return c;
	}

	public Double getAnguloEnGrados() {
		return Math.toDegrees(angulo);
	}

	public Double getAngulo(Recta2D r) {
		return this.vector.getAngulo(r.getVector());
	}

	public Double getAnguloEnGrados(Recta2D r) {
		return Math.toDegrees(getAngulo(r));
	}

	public Boolean contienePunto(Punto2D p) {
		return a*p.getX()+b*p.getY()+c == 0.;
	}

	public Double getDistanciaAlOrigenConSigno() {
		return this.distanciaAlOrigenConSigno;
	}

	public Double getDistancia(Punto2D p) {
		Double r = a*p.getX()+b*p.getY()+c;
		r = r/Math.hypot(a, b);
		return r;
	}

	public Recta2D getParalela(Punto2D p) {
		return Recta2D.create(p, vector);
	}

	public Recta2D getPerpendicular(Punto2D p) {
		return Recta2D.create(p, vector.getOrtogonal());
	}

	public Punto2D cortaA(Recta2D r) {
```

```
		Punto2D p = null;
		Double d = this.a*r.b-r.a*this.b;
		if(d!=0.){
			p = Punto2D.create(
				this.b*r.c-r.b*this.c, r.a*this.c-this.a*r.c);
		}
		return p;
	}

	@Override
	public Recta2D rota(Punto2D p, Double angulo) {
		Punto2D p1 = this.punto;
		Punto2D p2 = this.punto.add(vector);
		return Recta2D.create(p1.rota(p, angulo), p2.rota(p, angulo));
	}

	@Override
	public Recta2D traslada(Vector2D v) {
		return Recta2D.create(this.getPunto().traslada(v), this.angulo);
	}

	@Override
	public String toString() {
		return Math2.simplify(a) + " X " +
			Math2.simplify(b)+ " Y " +
			Math2.simplify(c) + " = 0";
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime
				* result
				+ ((this.angulo == null) ? 0
						: this.angulo.hashCode());
		result = prime
				* result
				+ ((this.distanciaAlOrigenConSigno == null) ? 0
						:
this.distanciaAlOrigenConSigno.hashCode());
		return result;
	}

	@Override
	public boolean equals(Object obj) {
```

```
            if (this == obj)
                  return true;
            if (obj == null)
                  return false;
            if (!(obj instanceof Recta2D))
                  return false;
            Recta2D s2 = (Recta2D) obj;
            if (this.angulo == null) {
                  if (s2.angulo != null)
                        return false;
            } else if (!this.angulo
                        .equals(s2.angulo))
                  return false;
            if (this.getDistanciaAlOrigenConSigno() == null) {
                  if (s2.getDistanciaAlOrigenConSigno() != null)
                        return false;
            } else if (!this.getDistanciaAlOrigenConSigno()
                        .equals(s2.getDistanciaAlOrigenConSigno()))
                  return false;
            return true;
      }
}
```

Semiplano2D extends Recta2D

Descripción: Semiplano en 2D. Inmutable. Ecuación del semiplano $Ax + By + C \leq 0$.

Propiedades:

- *Opuesto: Semiplano2D*, consultable.

Propiedades relacionadas con la igualdad:

- *Criterio de igualdad*: dos objetos de este tipo son iguales si tienen la misma distancia al origen (teniendo en cuenta el signo) y el mismo α. Donde $\alpha = \tan^{-1}\frac{-A}{B}, \quad con -\pi \leq \alpha \leq \pi$
- *Representación como cadena*: en la forma $Ax + By + C \leq 0$

Operaciones:

- *boolean contiene(Punto2D p):* Observadora. Si $Ax + By + C \leq 0$

- *boolean contiene(Poligono2D p):* Observadora.
- *Poligono2D intersecta(Poligono2D v):* Observadora.

Factoría:

- *Semiplano2D create(Recta2D r, Punto2D p):* Define el semiplano a partir de una recta y un punto del plano que no pertenezca a la recta.
- *Semiplano2D create(Semiplano2D r);*

Un semiplano viene definido por $Ax + By + C \leq 0$. Todos los semiplanos de la forma $\lambda(Ax + By + C) \leq 0$ para $\lambda > 0$ son iguales. La recta que define el semiplano es $Ax + By + C = 0$. Un semiplano viene definido de forma única la recta y por α. Donde $\alpha = \tan^{-1}\frac{-A}{B}, \ con - \pi \leq \alpha \leq \pi$.

De las consideraciones anteriores vemos que un semiplano podemos considerarlo como un subtipo de una recta. La relación de igualdad para el semiplano es más estricta que para la recta.

Abajo puede consultarse el código.

```
public class Semiplano2D extends Recta2D implements ObjetoGeometrico2D {

	private double alfa;
	private double a;
	private double b;
	private double c;

	public static Semiplano2D create(Recta2D r, Punto2D s) {
		return new Semiplano2D(r,s);
	}

	public static Semiplano2D create(Semiplano2D r) {
		return new Semiplano2D(r.getPunto(),
				r.getVector(),r.alfa,r.a,r.b,r.c);
	}

	private Semiplano2D(Punto2D p, Vector2D vector, double alfa, double a,
			double b, double c) {
		super(p, vector);
		this.alfa = alfa;
		this.a = a;
		this.b = b;
```

```
		this.c = c;
	}

	private Semiplano2D(Recta2D r, Punto2D pc) {
		super(r);
		if(getA()*pc.getX()+getB()*pc.getY()+getC() <=0. ){
			a = getA();
			b = getB();
			c = getC();
		} else {
			a = -getA();
			b = -getB();
			c = -getC();
		}
		double y = this.a;
		double x = -this.b;
		this.alfa = Math.atan2(y, x);
	}


	public Semiplano2D getOpuesto(){
		return new Semiplano2D(getPunto(),getVector(),
			this.alfa+Math.PI,-this.a,-this.b,-this.c);
	}

	public Boolean contains(Punto2D p) {
		return a*p.getX()+b*p.getY()+c <= 0.;
	}

	public Boolean contains(Poligono2D a) {
		Boolean r = true;
		for(Punto2D p:a.getVertices()){
			r = r & contains(p);
			if(!r) break;
		}
		return r;
	}

	...

	@Override
	public String toString() {
		return Math2.simplify(a) + " X " +
			Math2.simplify(b)+ " Y " + Math2.simplify(c) + " < 0";
	}
```

```
	@Override
	public int hashCode() {
		final int prime = 31;
		int result = super.hashCode();
		long temp;
		temp = Double.doubleToLongBits(alfa);
		result = prime * result + (int) (temp ^ (temp >>> 32));
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (!super.equals(obj))
			return false;
		if (!(obj instanceof Semiplano2D))
			return false;
		Semiplano2D other = (Semiplano2D) obj;
		if (Double.doubleToLongBits(alfa) != Double
				.doubleToLongBits(other.alfa))
			return false;
		return true;
	}

}
```

6. Otros problemas de implementación

En muchos casos es necesario dotar a varios tipos de un conjunto de operaciones comunes. Es el caso de los objetos geométricos en dos dimensiones. Entre estas operaciones comunes podemos incluir *rotar*, *trasladar* y *mostrar (draw*) en la pantalla.

Este objetivo puede conseguirse diseñando un interface con los métodos adecuados y haciendo que cada uno de los objetos geométricos lo implementen.

```
public interface ObjetoGeometrico2D {
	ObjetoGeometrico2D rota(Punto2D p, Double angulo);
	ObjetoGeometrico2D traslada(Vector2D v);
	void draw(Graphics2D g);
}
```

Los tipos *Punto2D, Recta2D, Semiplano2D, Poligono2D* deberían implementar este interface. También podemos diseñar un agregado de objetos geométricos que nos permita aplicar las operaciones anteriores de forma conjunta. La implementación de este agregado se deja como ejercicio.

Ahora tenemos que resolver el problema de implementar estas operaciones para los tipos que implementan *ObjetoGeometrico2D.*
Dada es clase la implementación de los métodos del objeto geométrico para el tipo *Punto2D* es:

```
public class Punto2D implements Comparable<Punto2D>, ObjetoGeometrico2D {

	...

	public Punto2D traslada(Vector2D v){
		return add(v);
	}

	public Punto2D rota(Punto2D p, Double angulo){
		Vector2D v = minus(p).rota(angulo);
		return p.add(v);
	}

	public void draw(Graphics2D g) {
		MarcoDeTrabajo.fill(g, this);
	}
	...
}
```

Para facilitar el trabajo se ha diseñado la clase Marco de Trabajo. Nos sirve de adaptador entre las capacidades gráficas de la pantalla y las necesidades de implementación de los objetos geométricos. La clase usa las capacidades gráficas de Java. En esta clase se puede definir una ventana de un tamaño dado y se escogen un sistema de coordenadas cercano al que solemos usar e independiente de las coordenadas que usa el sistema gráfico de Java. Los detalles veremos en capítulos posteriores cuando estudiamos los aspectos gráficos.

7. Ejercicios propuestos

Implementar cada uno de los tipos propuestos en el capítulo anterior