CAPÍTULO 3

Conceptos básicos de programación orientada a objetos

La programación orientada a objetos se define como un paradigma que permite realizar una abstracción de la realidad, que se puede implementar en una aplicación de *software* con el fin de resolver problemas mediante el uso de un lenguaje de programación.

El paradigma de orientación a objetos comprende una gran cantidad de conceptos que permite el desarrollo de aplicaciones robustas.

3.1 Paquete

Un paquete es un contenedor de clases. Se utiliza para ordenar el código de forma consistente de acuerdo a los servicios implementados. Para que un código se encuentre contenido en un paquete es necesario agregar la siguiente sentencia.

```
package MiPaquete;
```

En donde "*Mi Paquete*" es el nombre del paquete que contendrá el código. Por otro lado, si se desea hacer uso de servicios implementados en otros paquetes se debe agregar el siguiente código.

```
import OtroPaquete;
```

Java contiene una gran cantidad de paquetes que proveen una gran cantidad de servicios. Algunos de estos paquetes se presentan en la Tabla 12.

Tabla 12. Paquetes básicos del *API* de Java

Paquete	Descripción
java.applet	Provee clases necesarias para crear *applets*.
java.awt	Contiene todas las clases para crear interfaces gráficas para pintar gráficas e imágenes.
java.awt.color	Provee clases para definiciones de color.
java.awt.event	Provee interfaces y clases para manejar eventos de componentes gráficos.
java.awt.font	Provee interfaces y clases relacionadas con fuentes.
java.awt.geom	Provee clases 2D para definir operaciones relacionadas con geometría de dos dimensiones.
java.awt.image	Provee interfaces y clases para crear y modificar imágenes.
java.awt.print	Provee interfaces y clases para usar el *API* de impresión.
java.beans	Provee interfaces y clases para el desarrollo de *beans,* que hace referencia a componentes basados en *JavaBeans™ architecture*.
java.beans.beancontext	Provee interfaces y clases relacionadas con *bean context*.
java.io	Provee interfaces y clases para entrada y salida de datos serializables.

java.lang	Provee clases fundamentales para el diseño del lenguaje de programación Java.
java.math	Provee clases para optimizar la precisión de entero aritmético (*BigInteger*) decimal aritmético (*BigDecimal*).
java.net	Provee clases para implementar aplicaciones de red.
java.rmi	Provee interfaces y clases para servicios *RMI*.
java.security	Provee interfaces y clases para el *framework* de seguridad.
java.sql	Provee interfaces y clases para procesar datos almacenados en fuentes de datos como bases de datos.
java.text	Provee interfaces y clases para manipular texto, fechas, números y mensajes.
java.util	Contiene el *framework* de colecciones, modelo de eventos, servicios de fecha y tiempo, internacionalización y clases misceláneas.

3.2 Clase

Una clase se define como un tipo abstracto de dato que contiene atributos y métodos. A través de una clase se implementa un concepto abstraído de la realidad. En este caso, los atributos hacen referencia a las características del concepto abstraído y los métodos hacen referencia a los servicios de dicho concepto.

La sintaxis de la clase debe ser la siguiente:

```
public class MiClase{
   //Definición de atributos
   //Definición de métodos
}
```

En Java se debe seguir una buena práctica que consiste en implementar cada clase en un archivo independiente con extensión ***.java***. Para el ejemplo anterior, el archivo debe denominarse *MiClase.java*.

3.2.1 Atributos

Los atributos hacen referencia a las características que se le incluyen a la clase. Estos atributos pueden ser declaraciones de tipos primitivos de datos o declaraciones de clases.

3.2.2 Visibilidad

La visibilidad se refiere al nivel de accesibilidad de los atributos y métodos. Los niveles de accesibilidad se dan por los siguientes términos:

1. *private.* Se puede acceder desde un método implementado desde la misma clase.
2. *public*. Se puede acceder desde un método implementado en cualquier clase.
3. *protected*. Se puede acceder desde un método implementado en una clase que herede la clase que contiene esta visibilidad y desde clases implementadas en el mismo paquete.

3.2.3 Métodos

Los métodos hacen referencia a los servicios que se le incluyen a la clase. En estos métodos se implementa el código necesario del servicio. Un método contiene los siguientes elementos:

1. Visibilidad. Se debe establecer si el método es *private*, *public* o *protected*.
2. Retorno. Un método puede retornar información. Si el método no retorna información se debe colocar la palabra reservada "*void*".

El retorno puede ser un tipo primitivo de dato o una clase. Si un método tiene retorno, en la implementación del método, debe estar presente la palabra reservada "*return*".

3. Nombre. Identificador del método en la clase.
4. Parámetros. Un método puede recibir de *0* a *n* parámetros. Un parámetro puede ser un tipo primitivo de dato o una declaración de una clase. Los parámetros deben estar separados por comas.

Cada método implementa un código que debe estar contenido entre "*{*" y "*}*". La sintaxis de los métodos es la siguiente.

```
//método publico sin retorno y sin parámetros
public void miMetodo(){
   instrucción 1;
   instrucción 2;
   ..
   instrucción n;
}

//método privado con retorno int y sin parámetros
prívate int miMetodo(){
   instrucción 1;
   instrucción 2;
   ..
   instrucción n;
   return valorInt;
}

//método privado con retorno int y con parámetros
prívate int miMetodo(int parametro1, boolean parametro2, MiClase
parametro3){
   instrucción 1;
   instrucción 2;
   ..
   instrucción n;
   return valorInt;
}
```

3.2.4 Encapsulamiento

Es una característica que indica que los atributos que definen propiedades propias de la clase deben tener visibilidad private. De esta forma se ofrece seguridad a la información depositada en dichos atributos.

3.2.5 Apuntador *this*

El apuntador "*this*" permite acceder a los atributos y métodos de la clase. El uso del apuntador no es obligatorio, pero se recomienda usarlo como buena práctica. Es posible que el parámetro de un método tenga el mismo nombre que un atributo, en este caso el uso del apuntador *this* es obligatorio para que el compilador identifique si está haciendo referencia al atributo o al parámetro del método.

3.3 Objeto

Un objeto es la referencia e instancia de una clase. Al crear una referencia se asigna un espacio de memoria dinámica al objeto, pero no es utilizable. Al crear la instancia, el objeto es utilizable. La sintaxis de la referencia es la siguiente.

```
MiClase m;
```

Donde *m* es la referencia del objeto. La sintaxis de la instancia es:

```
m = new MiClase();
```

Al hacer la instancia se puede acceder a los atributos y métodos públicos y protegidos si aplica, a través del objeto *m*. Otra sintaxis para realizar referencia e instancia en la misma línea de código es:

```
MiClase m = new MiClase();
```

3.4 Sentencia *static*

Una clase puede tener atributos y/o métodos propios o no del objeto. La sentencia "*static*" define estos atributos y métodos de tal forma que puedan ser accedidos sin requerir una instancia de la clase. Por otro lado, un atributo "*static*" toma el mismo valor para todos los objetos que sean instancia de la clase que lo contiene. Por ejemplo, la clase *Math* contiene el método "*sin*" el cual calcula el seno de un parámetro dado.

Ejemplo:

```
public class MiClase{

   public static int miValor;

   public static long factorial(long n) {
       long fact=1;
       for(int i=1; i<n; i++){
          fact *= i;
       }
       return fact;
   }
}
```

En donde se puede hacer uso del método factorial de la siguiente forma.

```
long valor = MiClase.factorial(5);
```

También permite hacer uso del atributo *miValor* de la siguiente forma.

```
MiClase c1 = new MiClase();
c1.miValor = 10;
MiClase c2 = new MiClase();
MiClase c3 = new MiClase();
```

En el código anterior, el atributo *miValor* tendrá el valor 10 para los objetos c1, c2 y c3, solo con asignarlo en un objeto de ellos, que para el caso es en c1.

3.5 Sentencia final

Una clase puede tener atributos finales que hacen referencia a constantes que no pueden cambiar su valor en tiempo de ejecución de la aplicación. La sintaxis es la siguiente:

```
public class MiClase{

   public final static int uno=1;
   public final static int dos=2;

}
```

Por ejemplo el atributo "*PI*" cuyo valor se encuentra implementado en la clase del *API* de Java *Math,* puede ser accedido sin requerir instancia de la clase *Math* y su valor es constante.

3.6 Clasificación de métodos

Los métodos se pueden clasificar en cuatro tipos que son los siguientes:

1. Constructores. Un constructor es el primer método que se ejecuta al realizar la instancia de un objeto. Uno de los usos principales de un constructor es la inicialización de los atributos de la clase. El método constructor debe tener visibilidad pública y no posee retorno. La sintaxis es la siguiente:

```
public class MiClase{

   //Definición de atributos
   prívate int atributo1;
   prívate int atributo2;

   //Definición de método constructor
   public MiClase(){
       this.atributo1=0;
       this.atributo1=0;
   }
}
```

2. Consultores. Un consultor es el método que permite retornar el valor de un atributo con visibilidad private al aplicar el concepto de encapsulamiento. La sintaxis es la siguiente:

```
public class MiClase{

   //Definición de atributos
   prívate int atributo1;
   prívate int atributo2;

   //Método constructor
   public MiClase(){
       this.atributo1=0;
       this.atributo1=0;
   }
```

```
    //Método consultor
    public int getAtributo1() {
        return this.atributo1;
    }

    //Método consultor
    public int getAtributo2() {
        return this.atributo2;
    }
}
```

3. Modificadores. Un modificador es el método que permite asignar valor a un atributo con visibilidad *private* al aplicar el concepto de encapsulamiento. La sintaxis es la siguiente:

```
public class MiClase{

    //Definición de atributos
    prívate int atributo1;
    prívate int atributo2;

    //Método constructor
    public MiClase(){
        this.atributo1=0;
        this.atributo1=0;
    }

    //Método consultor
    public int getAtributo1() {
        return this.atributo1;
    }

    //Método consultor
    public int getAtributo2() {
        return this.atributo2;
    }

    //Método modificador
    public void setAtributro1(int atributo1) {
        this.atributo1 = atributo1;
    }

    //Método modificador
    public void setAtributro2(int atributo2) {
        this.atributo2 = atributo2;
    }
}
```

4. Analizadores. Un analizador es el método que permite implementar la lógica del servicio del mismo, es decir, allí se implementan los algoritmos requeridos. La sintaxis es la siguiente:

```
public class MiClase{

   //Definición de atributos
   prívate int atributo1;
   prívate int atributo2;

   //Método constructor
   public MiClase(){
       this.atributo1=0;
       this.atributo1=0;
   }

   //Método consultor
   public int getAtributo1() {
       return this.atributo1;
   }

   //Método consultor
   public int getAtributo2() {
       return this.atributo2;
   }

   //Método modificador
   public void setAtributro1(int atributo1) {
       this.atributo1 = atributo1;
   }

   //Método modificador
   public void setAtributro2(int atributo2) {
       this.atributo2 = atributo2;
   }

   //Método analizador
   public int calcularMayor() {
       if(this.atributo1 > this.atributo2){
          return this.atributo1;
       }else{
          return this.atributo2;
       }
   }
}
```

3.7 Sobrecarga de métodos

La sobrecarga de métodos es una característica que permite que varios métodos en una misma clase tengan el mismo nombre. La forma en que el compilador identifica cuál es el método a utilizar en tiempo de ejecución, se debe a que estos deben poseer diferentes parámetros y/o retorno. La diferencia puede estar dada en el número de parámetros y/o en el tipo de los mismos. Por ejemplo se plantean los siguientes métodos sobrecargados.

```
//Método sobrecargado 1. Sin parámetro y sin retorno.
public void miMetodo(){

}

//Método sobrecargado 2. Con parámetro y con retorno int.
public int miMetodo(int parametro1){

}

//Método sobrecargado 3. Con parámetros y con retorno boolean.
public boolean miMetodo(int parametro1, int parametro2){

}

//Método sobrecargado 4. Con parámetros diferentes a la
sobrecarga 3 y con retorno boolean.
public boolean miMetodo(int parametro1, long parametro2){

}
```

3.8 Recursividad

La recursividad es la característica en la programación que permite hacer un llamado a un método desde el mismo método. Esta característica simplifica el desarrollo. Cada llamado recursivo equivale a una iteración en una estructura de repetición como el "*while*" o el "*for*". Tiene la ventaja de utilizar casi los mismos recursos que en un proceso iterativo regular. Por otro lado, existen algoritmos que necesariamente deben ser implementados de forma recursiva como algoritmos fractales y árboles.

Para aplicar el concepto de recursividad, el método debe necesariamente retornar un valor, recibir por parámetro al menos un valor, implementar una condición de ruptura del proceso recursivo e implementar una función recursiva.

Por ejemplo, si se desea implementar el algoritmo del factorial se podría implementar el siguiente método para resolver el algoritmo:

```
public long factorial(long n) {
   long fact=1;
   for(int i=1; i<n; i++){
       fact *= i;
   }
   return fact;
}
```

Entonces, suponiendo que *n*=5, el algoritmo realiza 5 iteraciones, en donde en cada iteración se presentan los siguientes resultados en las variables:

1. Primera iteración: fact=1*1=1
2. Segunda iteración: fact=1*2=2
3. Tercera iteración: fact=2*3=6
4. Cuarta iteración: fact=6*4=24
5. Quinta iteración: fact=24*5=120

También se puede hacer la implementación del algoritmo del factorial de forma recursiva.

```
public long factorial(long n) {
   if(n==1 || n==0){
       return 1;
   }else{
       return n*factorial(n-1);
   }
}
```

Otra forma más avanzada de codificar el mismo algoritmo recursivo es la siguiente:

```
public long factorial(long n) {
   return (n==1)?1:n*factorial(n-1);
}
```

Entonces, suponiendo que *n*=5, el algoritmo realiza 5 llamadas recursivas, en donde en cada llamada se presentan los siguientes resultados.

1. Primer llamada: retorna 5*factorial(4)
2. Segunda llamada: retorna 4*factorial(3)
3. Tercer llamada: retorna 3*factorial(2)
4. Cuarta llamada: retorna 2*factorial(1)
5. Quinta llamada: retorna 1

Para hacer el primer llamado al método se puede considerar la siguiente sentencia:

```
long f = factorial(5);
```

En la Figura 1 se observa cómo cada llamada aporta a la consecución del resultado del algoritmo.

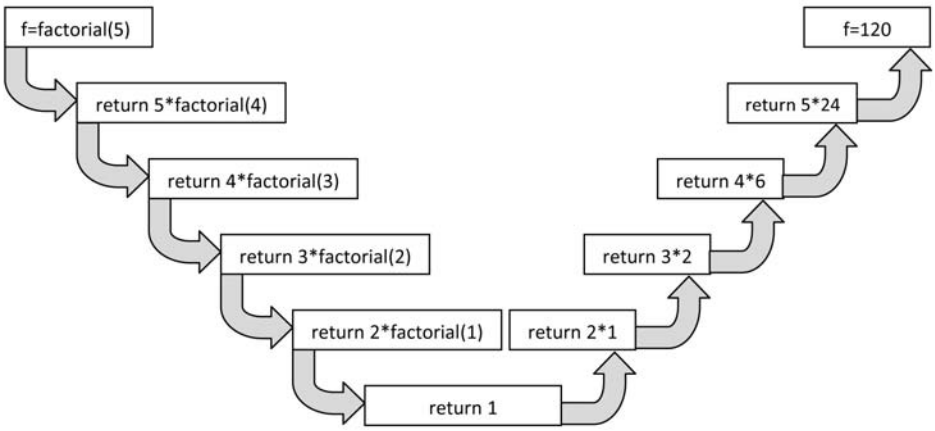

Figura 1. Representación de funcionamiento de recursividad

3.9 Bajo acoplamiento

Es la característica en el paradigma de orientación a objetos que indica que los diferentes subsistemas deben estar unidos de forma mínima. Esto indica, que las clases que se construyen deben ser lo

más reducidas, sin involucrar elementos que impliquen conceptos diferentes a los tratados en la clase.

Un ejemplo de bajo acoplamiento podría ser una memoria *USB*. Esta memoria es un sistema totalmente independiente al sistema del computador. Se conectan los dos sistemas a través del puerto *USB* y de forma automática, ambos sistemas quedan integrados. Además, esta operación se puede realizar mientras el computador se encuentre en funcionamiento. Así mismo es posible desconectar la memoria sin problemas.

Un ejemplo de alto acoplamiento, con base en el anterior es el disco duro. Este se encuentra altamente acoplado, debido a que no se puede desconectar mientras el computador se encuentra en funcionamiento. Además, sin este sistema el computador no puede funcionar. Por otro lado, un ejemplo de bajo acoplamiento es la memoria *USB*, la cual se puede conectar y desconectar de forma simple y no afecta ni al computador ni a la memoria misma.

3.10 Alta cohesión

Es la característica en el paradigma de orientación a objetos que indica que, las propiedades y servicios de una clase deben ser consistentes con el concepto que abstrae dicha clase.

Por ejemplo, si se tiene una clase *Triángulo*, esta clase podría contener los siguientes atributos:

- Identificación
- Base
- Altura

Además podría tener servicios como:

- Calcular Área
- Calcular Perímetro

En este ejemplo la clase *Triángulo* se encuentra altamente cohesionada ya que los atributos y métodos hacen referencia

directa a características y comportamientos de concepto abstraído que es el triángulo.

Si se incluye por ejemplo el método calcular volumen, esta clase estaría bajamente cohesionada, ya que un triángulo no posee volumen. Este método tendría que ser trasladado a la clase pirámide.

3.11 Manejo de excepciones

En el lenguaje Java, una "*Exception*" hace referencia a una condición anormal que se produce en tiempo de ejecución de la aplicación. Algunas excepciones son denominadas fatales, las cuales provocan la finalización de la ejecución de la aplicación. Generalmente, las excepciones se generan por que falla la operación como consecuencia de un error de uso de la aplicación por parte del usuario. Para ilustrar el concepto, se presentan los siguientes ejemplos:

- Si el usuario intenta abrir un archivo e ingresa de forma incorrecta la ruta del mismo, la aplicación presenta una excepción que debe controlarse para presentarle información de error de ruta del archivo al usuario.

- Si el usuario desea ingresar un número para realizar una operación aritmética, pero erróneamente ingresa un carácter, la aplicación presenta una excepción de formato de número que debe controlarse para indicarle al usuario que no se puede realizar la operación aritmética.

Las excepciones se representan mediante clases derivadas de la clase *Throwable*, sin embargo, las clases con las que se desarrolla, se derivan de la clase *Exception* que pertenece al paquete **java.lang**.

3.11.1 Estructura *try*, *catch* y *finally*

Las excepciones en Java deben ser capturadas mediante el uso de las estructuras "*try*", "*catch*" y "*finally*". En el bloque *try* se debe implementar el código del proceso que se desea ejecutar. En el

bloque *catch* se implementa el código alterno que se ejecutará en caso de que se presente una situación anormal o excepción en la ejecución del código implementado en el bloque *try*. Es posible tener varios bloques *catch* que resuelvan diferentes tipos de excepción. El bloque *finally* es opcional, pero en caso de implementarse, este se ejecutará independientemente, si se presenta o no excepción. Este se implementa posterior a la implementación del bloque *try* y del bloque *catch*. La sintaxis es la siguiente:

```
public void miMetodo(){
    ..
    try{
        instrucción 1;
        instrucción 2;
        ..
        instrucción n;
    }catch(Exception e){
        //Instrucciones del manejo de la excepcion
    }finally{
        //Instrucciones que se ejecutan en cualquiera de los dos casos
    }
    ..
}
```

3.11.2 Sentencia *throws*

En caso que el código de un método genere una "*Exception"*, pero no se desee manejar dicha excepción, es posible enviar el manejo de la misma al método que hace el llamado. Este envío del manejo de la excepción se realiza mediante la inclusión de la sentencia "*throws*" seguida del nombre de la excepción posterior a los parámetros del método. Esta sentencia obliga a que el método que hace el llamado, implemente el manejo de la excepción a través del bloque "*try catch"* o envíe a su vez la excepción al método que hace el llamado a través de la sentencia "*throws"*. La sintaxis es la siguiente.

```
public void miMetodo()throws Exception{
    ..
}
```

3.11.3 Excepciones estándar del *API* de Java

Las excepciones en Java se representan mediante dos tipos de clases derivadas de la clase *Throwable* que son *Error* y *Exception*.

La clase *Error* está relacionada con errores de compilación, errores del sistema o errores de la *JVM*. Estos errores son irrecuperables y no dependen del desarrollador.

La clase *Exception* es la que debe tener en cuenta el desarrollador, debido a que de esta se derivan clases que manejan las excepciones que pueden ser controladas en tiempo de ejecución. Las clases derivadas de *Exception* más usuales son:

1. **RuntimeException***:* contiene excepciones frecuentes en tiempo de ejecución de la aplicación.
2. **IOException***:* contiene excepciones relacionadas con entrada y salida de datos.

Las clases derivadas de *Exception* pueden pertenecer a distintos paquetes de Java. Algunas de ellas pertenecen a **java.lang**, otras a **java.io** y a otros paquetes. Por derivarse de la clase *Throwable* todos los tipos de excepciones pueden usar los métodos siguientes:

1. *String getMessage():* extrae el mensaje asociado con la excepción.
2. *String toString():* devuelve un String que describe la excepción.
3. *void printStackTrace():* indica el método donde se lanzó la excepción.

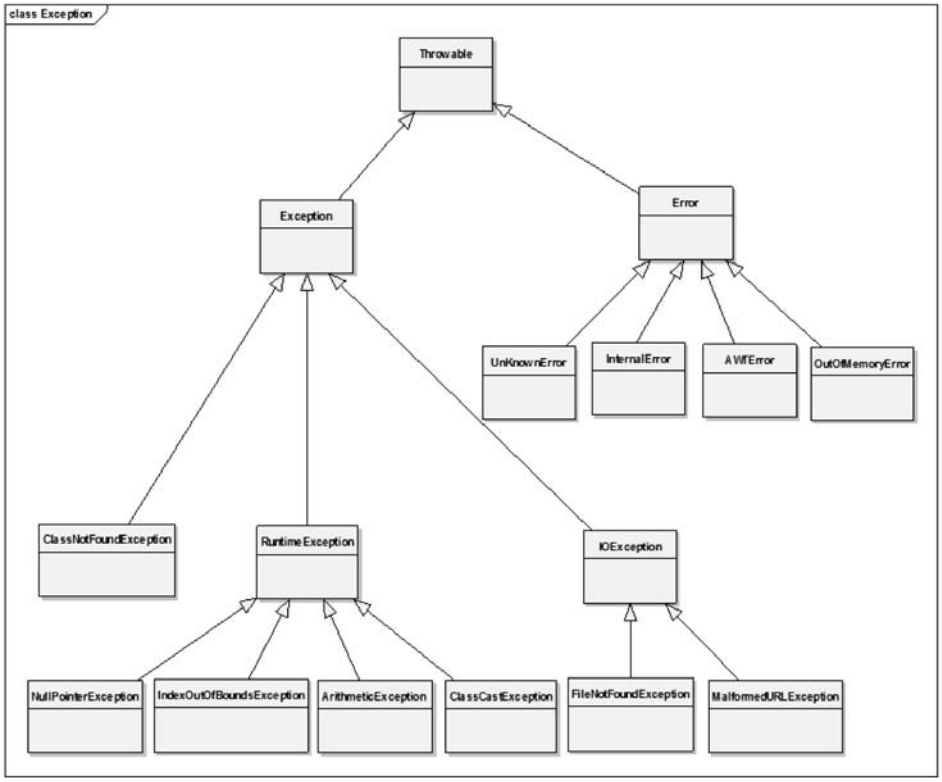

Figura 2. Jerarquía simplificada de clases derivadas de *Throwable*

3.11.4 Creación de excepciones en Java

En un proyecto es posible crear excepciones propias solo con heredar de la clase *Exception* o de una de sus clases derivadas. Las clases derivadas de *Exception* suelen tener dos constructores:

1. Un constructor sin argumentos.
2. Un constructor que recibe un *String* como argumento. En este *String* se suele definir un mensaje que explica el tipo de excepción generada. Este mensaje debe enviarse a la clase *Exception* mediante la sentencia *super(String).*

La sintaxis es la siguiente:

```
class MiExcepcion extends Exception {
   public MiExcepcion() { // Constructor por defecto
       super();
   }
   public MiExcepción(String s) { // Constructor con mensaje
       super(s);
   }
}
```

3.12 Ejercicios propuestos

1. Implemente una clase denominada *Cuadrado* que contenga un atributo privado, dos métodos constructores sobrecargados con y sin parámetros, métodos consultores, métodos modificadores y métodos analizadores que calculen el área y perímetro del cuadrado.

2. Implemente una clase denominada *Triángulo* que contenga un atributo privado, dos métodos constructores sobrecargados con y sin parámetros, métodos consultores, métodos modificadores y métodos analizadores que calculen el área y perímetro del triángulo.

3. Implemente una clase denominada *Rectángulo* que contenga un atributo privado, dos métodos constructores sobrecargados con y sin parámetros, métodos consultores, métodos modificadores y métodos analizadores que calculen el área y perímetro del rectángulo.

4. Implemente una clase denominada *Operaciones*, que contenga métodos estáticos que calculen el factorial de un número y que verifique si un número es primo.