Programación Orientada a Objetos

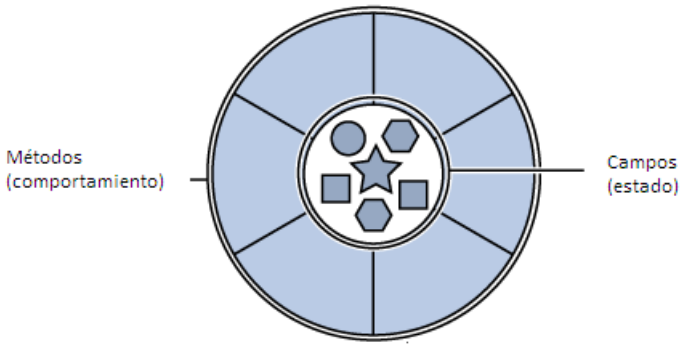
Contenido

1. ORIENTACIÓN A OBJETOS

Los objetos del mundo real comparten dos características: todos tienen **estado** y **comportamiento**. Una lámpara tiene un estado (encendida, apagada,) y comportamiento (encender, apagar,) Las bicicletas también tienen un estado (marca, modelo, velocidad actual, cadencia actual,) y comportamiento (cambiar de marcha, frenar,). Identificar el estado y comportamiento de los objetos del mundo real es una buena manera de comenzar a pensar en términos de programación orientada a objetos.

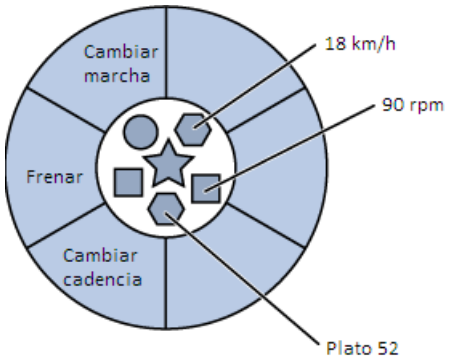
Contemplando los objetos del mundo real nos surgirán dos preguntas: "¿En qué posibles estados puede estar el objeto?", y "¿Qué comportamientos puede tomar el objeto?". La complejidad varía de un objeto a otro. Algunos objetos pueden a su vez estar formados por otros objetos.

Los **objetos software** son conceptualmente similares a los objetos del mundo real: están compuestos de estado y comportamiento relacionados. Un objeto almacena su estado en **campos** (atributos, variables) y expone su comportamiento a través de **métodos** (funciones, procedimientos). Los métodos operan sobre el estado interno de un objeto y sirve de mecanismo primario de comunicación entre objetos. Ocultar el estado interno requiriendo que todas las interacciones sean realizadas a través de los métodos es lo que se conoce como **ENCAPSULACIÓN**, un principio fundamental de la programación orientada a objetos.



Un objeto software.

Un ejemplo concreto. Una bicicleta:



Una bicicleta modelada como un objeto software.

Asignando un estado (velocidad actual, cadencia actual, marcha actual) y proporcionando métodos para cambiar el estado, el objeto mantiene el control sobre cómo el mundo exterior puede usarlo. Por ejemplo, si la bicicleta tiene 6 marchas, un método para cambiar marchas puede desechar cualquier valor que sea menor que 1 o mayor que 6.

Incluir el código dentro de los objetos software, provee varios beneficios:

1. **Modularidad:** El código fuente para un objeto puede ser escrito y mantenido independientemente del código fuente de los otros objetos.

2. **Ocultamiento de la información**: Interactuando con los objetos únicamente a través de sus métodos, los detalles de la implementación interna se ocultan al resto del sistema.

3. **Reutilización**: Una vez que un objeto se ha creado, puede ser utilizado en diferentes programas, sin más que importarlo en los mismos. No es necesario volver a escribir el código.

4. **Facilidad de depuración y Pluggability**: Si un objeto particular resulta problemático, basta con removerlo de la aplicación y "conectar" un objeto diferente como reemplazo. Esto es análogo al arreglo de problemas mecánicos en el mundo real.

2. CLASES y OBJETOS

En el mundo real nos encontraremos a menudo que muchos objetos individuales que son de un mismo tipo. Puede haber cientos de bicicletas, todas de la misma marca y modelo. Cada bicicleta ha sido construida de la misma manera y por tanto posee los mismos componentes. En términos de orientación a objetos, diríamos que una bicicleta concreta es una **instancia** de la clase de objetos conocida como Bicicleta. Una **clase** es una especia de plantilla a partir de la cual se crean los objetos individuales.

Un ejemplo de una posible implementación de la clase Bicicleta:

```
public class Bicicleta {
  int cadencia = 0;
  int velocidad = 0;
  int marcha = 1;

  void cambiarCadencia(int nuevoValor) {
      cadencia = nuevoValor;
  }

  void cambiarMarcha(int nuevoValor) {
      marcha = nuevoValor;
  }

  void aumentarVelocidad(int incremento) {
      elocidad = velocidad + incremento;
  }

  void frenar(int decremento) {
      velocidad = velocidad - decremento;
  }

  void imprimirEstado() {
      System.out.println("cadencia: " + cadencia + " velocidad: " +
                velocidad + " marcha: " +  marcha);
  }
}
```

Los campos `cadencia`, `velocidad`, y `marcha` representan el estado del objeto, y los métodos (`cambiarCadencia`, `cambiarMarcha`, `aumentarVelodidad` etc.) definen su interacción con el resto del mundo.

La clase `Bicicleta` no contiene un método `main`. No es una aplicación complete, tan solo una "plantilla" para bicicletas que podrá ser usada en una aplicación. La responsabilidad de crear y usar nuevos objetos `Bicicleta` pertenecerá a alguna otra clase dentro de la aplicación.

Un ejemplo de una clase en cuyo método `main` se crean dos objetos `Bicicleta` y se invoca a algunos de sus métodos:

```
public class Biclicletas {

    public static void main(String[] args) {
        // Crea dos objetos Bicicleta
        Bicicleta bike1 = new Bicicleta();
        Bicicleta bike2 = new Bicicleta();

        // Llamada a métodos de los objetos
        bike1.cambiarCadencia(50);
        bike1.aumentarVelocidad(10);
        bike1.cambiarMarcha(2);
        bike1.imprimirEstado();

        bike2.cambiarCadencia(50);
        bike2.aumentarVelocidad(10);
        bike2.cambiarMarcha(2);
        bike2.cambiarCadencia(40);
        bike2.aumentarVelocidad(10);
        bike2.cambiarMarcha(3);

        bike2.imprimirEstado();
    }

}
```

La salida de este programa mostrará la cadencia, velocidad y marcha final de ambas bicicletas:

```
cadencia: 50 velocidad: 10 marcha: 2
cadencia: 40 velocidad: 20 marcha: 3
```

2.1 Declaración de Clases

En las secciones anteriores ya se declararon algunas clases, como el ejemplo de la bicicleta. La declaración de una clase mínima es como sigue:

```
class UnhaClase {
// Declaración de atributos, constructores, y métodos
}
```

El cuerpo de la clase es lo que va entre llaves { } y puede contener (entre otros):

• Declaración de **campos** o atributos que determinan el estado de la clase y sus objetos

• **Métodos** para implementar el comportamiento de la clase y de sus objetos

• **Constructores**. Métodos para inicializar nuevos objetos

• **Modificadores** de acceso, como public o private

Por convenio el nombre de la clase debería comenzar por una letra mayúscula

2.2 Declaración de campos o atributos

En Java hay varios tipos de variables:

• Los campos o atributos de una clase que también se les llame **variables miembro** de una clase. Se conocen en toda la clase (pueden ser referenciadas desde cualquier método).

• Variables dentro de un método o de un bloque de que también se les llama **variables locales** . Sólo accesibles desde el método o bloque en que son declaradas, desde el punto en que son declaradas hasta el final del método o bloque.

• Variables que se usan en la declaración de un método que también se le llama **parámetros** A clase bicicleta, vista anteriormente, utiliza las siguientes líneas de código para definir sus atributos o variables miembro:

La clase bicicleta, vista anteriormente, utiliza las siguientes líneas de código para definir sus atributos o variables miembro:

```
int cadencia = 0;
int velocidad = 0;
int marcha = 1;
```

Para declarar un campo hay que seguir la siguiente sintaxis:

• Modificador de acceso (public, private, etc), es opcional
• Otrps modificadores (opcionales)
• El tipo de dato o la clase del campo
• El nombre del campo (por convenio debe comenzar por minúscula y utilizar camelCase)
Opcionalmente pueden ser inicializados. Los campos de tipos primitivos, a diferencia de las variables locales del mismo tipo, se inicializan automáticamente (int a 0, char a '\0', boolean a false, …)

2.3 Declaración de métodos

De modo genérico un método se declara de la siguiente manera:

```
listaModificadores tipoValorDevuelto nombreMetodo(parámetros) {
      // Cuerpo del método
}
```

La declaración de un método puede tener:

• Un serie de **modificadores**, incluyendo los de acceso (public, private, etc.)
• Tipo de dato que devuelve el método, en caso de que no devuelva nada se utiliza la palabra reservada **void**
• Nombre del método, que por convenio debe comenzar por minúscula y utilizar camelCase. Normalmente los métodos se nombran normalmente con un verbo, ya que indican una acción o comportamiento de un objeto.
• **Parámetros**, que son los valores de entrada del método

A continuación, entre llaves, se escribe la lógica del método.

Al nombre del método junto con sus parámetros se le llama la **signatura del método**.

Un método puede llamarse igual que otro, pero sus firmas deben ser distintas. A esto se le llama **sobrecarga de métodos**.

En una clase no puede haber dos métodos con la misma firma, aunque el tipo de valor que devuelvan sea distinto.

2.4 Manejo de Objetos

2.4.1 Creación de objetos

Las clases proporcionan un molde a partir del cual se pueden crear objetos. Podemos tener una clase Bicicleta pero si en un taller tenemos 20 bicicletas, habrá que instanciar 20 veces la clase Bicicleta, por cada una de ellas, con su número de serie, etc.

Declaración de objetos

La declaración de un objeto (en realidad la declaración de una variable que referencia o apunta a un objeto) se hace igual que con cualquier otra variable, pero el tipo de dato de la variable será una clase. Por ejemplo:

```
// Variable de tipo primitivo (int)
int x;
// Variable que referencia un obxecto de tipo Bicicleta
Bicicleta bike;
```

La variable **bike** es una referencia a un objeto de tipo **Bicicleta**. Tendrá un valor indeterminado hasta que se cree realmente el objeto, instanciando la clase Bicicleta.

Instanciación de objetos

La declaración de la variable no crea un objeto. Para ello es necesario utilizar el operador **new** tal y como se muestra en el siguiente ejemplo:

```
bike = new Bicicleta();
```

Podemos simultanear la declaración y la instanciación:

```
Bicicleta bike = new Bicicleta();
```

Inicialización de objetos

Inicializar un objeto implica dar valores a sus atributos cuando lo creamos. Podríamos darle valor mediante asignaciones en la declaración de los campos, pero esto implicaría que todos los objetos tuviesen, en un primer momento, el mismo estado. E modo correcto es mediante la utilización de **constructores**, que veremos posteriormente.

2.4.2 Uso de objetos

Una vez creados los objetos podemos trabajar con ellos, invocando sus métodos, y cambiar su estado, a través de la modificación de sus atributos.

Acceso a los campos de un objeto

Accedemos a los campos de un objeto con el operador de acceso (un punto) Para ello, escribimos el nombre del objeto seguido del operador de acceso (el punto) y el nombre del atributo. Por ejemplo:

```
bike.velocidad = 50;
```

Objetos diferentes pertenecientes a la misma clase tienen cada uno una copia de las variables miembro (campos o atributos) de la clase. Por lo tanto, cada objeto tendrá sus propios variables para esas variables. Con el operador de acceso accedemos a un atributo de un objeto concreto (el objeto que aparece precediendo al nombre del atributo).

Invocación de métodos

Para llamar a los métodos de un objeto empleamos la misma sintaxis que para acceder a sus atributos. Nuevamente, hay que recordar que accedemos a un método con un objeto concreto (el objeto que aparece precediendo al nombre del método):

```
Bike.frenar(15);
```

2.4.3 Destrucción de objetos

Java permite crear tantos objetos como se quiera sin tener que preocuparse de destruirlos una vez que ya no se precisan. Existe un proceso llamado recolector de basura (**garbage collector**) encargado de hacerlo. Podemos indicar explícitamente que una referencia ya no es necesaria asignándole el valor **null**. En otros lenguajes de programación (cómo en el lenguaje C) este trabajo tiene que hacerlo el programador, pero en Java el garbage collector periódicamente libera memoria utilizada por objetos que ya no están referenciados. De hecho, quien lo activa es la propia JVM (Java Virtual Machine).

2.5 Paso de parámetros

Los métodos pueden tener parámetros:

```
public int suma (int x, int y) {
      return (x + y);
}
```

Así, el método suma tiene dos parámetros enteros y devuelve su suma (otro entero). Los parámetros se utilizan en el cuerpo del método. Durante la ejecución de este, el método tomará los valores de los argumentos que se le pasan desde el programa que lo llama. Cuando se invoca un método, los argumentos utilizados deben coincidir en tipo y orden con la declaración de parámetros. Por ejemplo, para el método suma anterior los argumentos tienen que ser de tipo entero:

```
resultado1 = suma (4, 8);
resultado2 = suma (resultado1, 100);
```

Tenemos que distinguir entre parámetros y argumentos:

• Parámetros: se refiere a la lista de variables dentro de la declaración de un método

• Argumentos: son los valores reales que se pasan en el momento de invocación del método

Podemos usar cualquier tipo de parámetros dentro de un método o constructor, esto incluye los tipos primitivos y las referencias a objetos.

No podemos declarar dos parámetros con igual nombre dentro del mismo método o constructor. El nombre de un parámetro tampoco puede coincidir con el nombre de una variable local del método. Por ejemplo, el siguiente código dará un error de compilación:

```
public class Prueba {
      public void prueba(int i) {
            float i; // Falla
      }
}
```

El nombre de un parámetro puede coincidir con el nombre de un atributo de la clase.

2.5.1 Paso de parámetros por valor

Cuando pasamos un parámetro por valor, estamos operando sobre una copia de los datos. Cualquier cambio que se haga al valor del parámetro dentro del método será válido únicamente dentro diera método. Cuando el método finaliza el parámetro deja de existir; las modificaciones hechas en él, no se conservan. En Java los tipos primitivos de datos se pasan siempre por valor.

```
public class PasoPorValor {
	public static void main(String[] args) {
		int x = 3;
		// Invoca al método metodo1 con x como argumento
		metodo1(x);
		// Imprime por pantalla el valor de x para ver si cambió
		System.out.println("Después de invocar a metodo1, x = " + x);
	}
	// Método metodo1
	public static void metodo1(int p) {
		p = 10;
	}
}
```

La salida do programa será:

```
Después de invocar a metodo1, x = 3
```

2.5.2 Paso de objetos como parámetros.

Cuando pasamos un objeto como argumento de un método, estamos pasando una referencia a dicho objeto (pasamos la dirección de memoria donde esta almacenado), con lo que, si modificamos su estado, es decir, sus atributos, estas modificaciones permanecerán incluso después de que el método finalice.

2.6 Devolución de valores

Cuando invocamos a un método, la ejecución de este finaliza cuando ocurre alguna de las siguientes situaciones:

1. Se completan todas las sentencias dentro del método

2. Se ejecuta una sentencia **return**

3. Se lanza una excepción y esta no es tratada por el método

Una vez que el método finaliza el flujo de ejecución vuelve al método que lo invocó. Los métodos pueden devolver valores. El valor que devuelve un método se especifica, por lo tanto, en la propia declaración del método. Para devolver el valor usamos la sentencia **return** dentro del cuerpo del método seguido del valor a devolver.

Los métodos declarados con la palabra **void** no devuelven ningún valor por lo que no es necesario usar la sentencia return en ellos. Si devolvemos un valor desde un método void obtendremos un error de compilación. Análogamente, cualquier método que no sea declarado como void debe

tener una sentencia return con el correspondiente valor de retorno. El tipo de dato devuelto debe coincidir con el tipo declarado en el método:

```
public void setVelocidad(int velocidad) {
    if (velocidad >= 0) {
        this.velocidad = velocidad;
    }
}

public int getVelocidad() {
      Return this.velocidad;
}
```

El método **setVelocidad(int velocidad)** no devuelve ningún valor, por ello es declarado como **void**. El método **getVelocidad()** devuelve un número entero (sentencia **return**) por ello en su firma figura **int** como tipo de valor devuelto.

A parte de tipos primitivos, un método puede también devolver una referencia a un objeto. Por ejemplo:

```
@Override
public String toString() {
    return "cadencia: " + getCadencia() + " velocidad: "
             + getVelocidad() + " marcha: " + getMarcha();
}
```

2.7 Constructores

El constructor de una clase es un método especial que cuando se invoca permite crear instancias de esa clase, es decir, objetos. El constructor debe tener el mismo nombre que la clase y no devuelve ningún tipo de dato. Por ejemplo, un posible constructor para la clase Bicicleta:

```
public Bicicleta(int c, int v, int m) {
    cadencia = c;
    velocidad = v;
    marcha = m;
}
```

Este constructor, además, inicializa los campos de la clase. Este sería el modo preferible de inicializarlos en lugar de asignarles valores directamente en su declaración.

Ahora, para crear objetos de la clase Bicicleta escribiríamos, por ejemplo:

```
// Constructor sin argumentos
Bicicleta bike1 = new Bicicleta();
// Constructor con argumentos
Bicicleta bike2 = new Bicicleta(50, 25, 4);
```

Como vemos, puede haber más de un constructor por clase, siempre que tengan diferentes firmas.

En caso de no declarares explícitamente un constructor para una clase, se usa automáticamente el constructor de la superclase (lo veremos posteriormente en el apartado de Herencia), Si la superclase no tiene constructor el compilador automáticamente proporciona un constructor sin argumentos.

Podemos utilizar los modificadores de acceso para autorizar o denegar acceso a los constructores, igual que se fueran métodos, aunque normalmente los constructores tienen que poder ser accedidos desde otras clases para poder instanciar objetos de esa clase, por lo que suelen declararse como public.

2.8 Setters/Getters

Los campos de una clase, en caso de no indicar un modificador para los mismos, son de tipo **package**, esto es, pueden ser accedidos desde cualquier otra clase dentro del mismo paquete. Normalmente, es una buena práctica de programación ocultar la implementación de la clase, y solo permitir el acceso a sus campos, de ser necesario, a través de métodos. Con esto se consigue, por un lado, evitar accesos indebidos, y por otro lado aislar la implementación de modo que un cambio en la misma no altere el funcionamiento de programas ya realizados.

Es habitual declarar los atributos de una clase como **private** (no visible fuera de la clase) y los métodos que los manipulan como **public** (visible desde cualquier clase, en cualquier paquete). De este modo, para modificar o acceder a un atributo se usarán métodos A estos métodos se les llama métodos **setter** (para modificar un campo) y **getter** (para consultar un campo). En caso de que el campo sea de tipo boolean o Boolean, en lugar de **gett** se utiliza **is**.

Esto permite seguir lo que se conoce como **ENCAPSULACIÓN**. Un ejemplo:

```
public class Bicicleta {

    [...]
    // marchas válidas [1-7]
    private int marcha;
    private boolean parada;
```

```
    [...]
    public int getMarcha() {
        return marcha;
    }

    public void setMarcha(int marcha) {
        if (marcha >= 1 && marcha <= 7) {
           this.marcha = marcha;
        }
    }

    public boolean isParada() {
        return parada;
    }

    [...]

}
```

En este ejemplo, al tener que acceder al campo marcha mediante setter, nos aseguramos de que no se introduce una marcha errónea. Tampoco podemos modificar 'a pelo' el estado de parada, sólo podemos consultar su estado mediante isParada.

2.9 Método toString

Si intentamos imprimir un objeto (salvo los String y los envoltorios de tipos primitivos), veremos que la salida que obtenemos no se corresponde con el contenido de los campos del objeto (que seguramente era lo que querríamos obtener); en su lugar se muestra algo del tipo:

```
System.out.println(bike1);

-----------------------------------------------
// salida por pantalla
bicicletas.Bicicleta@6e8cf4c6
```

Lo que no está mostrando es el nombre del paquete, el nombre de la clase, y la dirección de memoria a partir de la cual se almacena la información del objeto (en este caso la **Bicicleta bike1**). Para obtener el estado del objeto podríamos utilizar un método, como el método **imprimirEstado**() del ejemplo de la **Clase Bicicleta**.

```
bike1.imprimirEstado();

-----------------------------------------------
// salida por pantalla
cadencia: 50 velocidad: 10 marcha: 2
```

Pero hay un método especial: **toString()** que se llama automáticamente cada vez que la referencia al objeto aparece en el contexto en el cual se esperaría un **String**. Por ejemplo, vamos a imprimir el estado de la Bici.

Codificamos el método **toString**:

```
@Override
public String toString() {
   return "cadencia: " + getCadencia() + " velocidad: "
          + getVelocidad() + " marcha: " + getMarcha();
}
```

Llamamos a **toString** desde el programa principal.

```
public static void main(String[] args) {
    Bicicleta bike1 = new Bicicleta(50, 25, 4);
    System.out.println(bike1);
}
```

La salida sería:

```
cadencia: 50 velocidad: 25 marcha: 4
```

El método **toString** se hereda automáticamente para cualquier clase que creemos. Todas las clases tienen este método, aunque no lo codifiquemos, pero su comportamiento por defecto se limita a imprimir la dirección de memoria del objeto.

2.10 Método equals

Cuando queremos el contenido de variables de tipos primitos para verificar su igualdad, podemos utilizar los operadores:

== (igual)

¡= (distinto)

Pero esto no funciona con los objetos, ya que lo que se compara es su dirección de memoria, no su estado. O sea, el resultado será cierto, si las variables comparadas o bien son la misma, o bien referencia al mismo objeto. En otro caso será falso.

Si lo que queremos es comparar en función del estado de los objetos, debemos utilizar el método **equals**. Este método existe en todas las clases (se hereda automáticamente), pero en la mayoría

de ellas no está “correctamente” codificado (se limita a comparar las direcciones de memoria, al igual que lo haría el operador **==**), así que será responsabilidad del programador proveer el código que determine si dos objetos son iguales o no. En la clase **String** sí es funcional.

Ejemplo, utilizamos el operador == para comparar dos **int** con el mismo valor, para comparar dos **Bicicleta** con el mismo estado, y por último para comparar dos variables (referencias) que apunta al mismo objeto **Bicicleta**.

```
public static void main(String[] args) {
    int a = 15;
    int b = 15;
    Bicicleta bike1 = new Bicicleta(50, 25, 4);
    Bicicleta bike2 = new Bicicleta(50, 25, 4);
    Bicicleta bike3 = bike1;

    System.out.println("a y b son " + (a == b ? "iguales" : "distintos"));
    System.out.println("bike1 y bike2 son " + (bike1 == bike2 ? "iguales" :
"distintos"));
    System.out.println("bike1 y bike3 son " + (bike1 == bike3 ? "iguales" :
"distintos"));

}
```

El resultado sería:

```
a y b son iguales
bike1 y bike2 son distintos
bike1 y bike3 son iguales
```

Java no sabe como decidir si una bicicleta es igual a otra. Es nuestra responsabilidad fijar ese criterio. Vamos a incorporar un nuevo campo en Bicicleta: **numeroSerie**, y vamos a suponer que dos Bicicleta son iguales si su **numeroSerie** es igual. Implementamos el método **equals**:

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final Bicicleta other = (Bicicleta) obj;
    return Objects.equals(this.numeroSerie, other.numeroSerie) ;
}
```

Comparamos ahora las bicicletas utilizando equals:

```
public static void main(String[] args) {
    Bicicleta bike1 = new Bicicleta(50, 25, 4, "432hljsrwr32");
    Bicicleta bike2 = new Bicicleta(50, 25, 4, "432hljsrwr32");
    Bicicleta bike3 = bike1;

    System.out.println("bike1 y bike2 son " + (bike1.equals(bike2) ?
"iguales" : "distintos"));
    System.out.println("bike1 y bike3 son " + (bike1.equals(bike3) ?
"iguales" : "distintos"));
}
```

Resultado:

```
bike1 y bike2 son iguales
bike1 y bike3 son iguales
```

2.11 Método main

El método **main** es un método especial, no se corresponde con una modelación del comportamiento de ningún objeto, de hecho, la clase donde se incluye (solo puede haber uno por aplicación) tampoco modela ningún objeto real, sino que es una clase "ficticia" para colocar dicho método. Este método es invocado automáticamente al ejecutar una aplicación Java, por tanto, es en el dónde debemos incluir el comienzo de nuestro código.

El método **main** se declara como sigue:

```
public static void main(String[] args)
```

NOTA: los modificadores los veremos en detalle en el siguiente tema.

- Modificador: **public**

  Indica que puede ser accedido desde fuera de la clase y también desde fuera del paquete.

- Modificador: **static**

  Es un método de clase, no de objeto. Esto implica que no es necesario instanciar ningún objeto de la clase que lo contiene para poder invocarlo.

- Tipo de valor devuelto: **void**

  No puede devolver nada. Si utilizamos una sentencia return, ha de ir en solitario (sin valor a devolver) e implicaría la finalización del programa.

- Parámetro: **String[] args**

Admite un único parámetro: un array de cadenas de caracteres. Este parámetro se le proporcionará desde la línea de comandos. Por ejemplo, vamos a poner el esqueleto de un programa que enseñe por pantalla el contenido de un fichero. El nombre del fichero evidentemente podrá variar de una ejecución a otra del programa. Una opción sería incluir sentencias para pedirlo por teclado una vez iniciado el programa (con por ejemplo la sentencia **nextLine()** de la clase **Scanner**), pero en lugar de esto vamos a hacer que el funcionamiento sea similar al comando **cat** del sistema: desde la línea de comandos indicamos el nombre del programa (cat) y sus parámetros (el fichero a visualizar):

- `Cat /home/alumno/fichero.txt`

En java, el código:

```
public class Muestra {
    public static void main(String[] args) {
            // miramos que haya un argumento
            if (args.length != 1) {
                  System.out.println("Se necesita un argumento: “
                    + fichero a visualizar");
          }
            // La primera posición del array contendrá el primer
            //   arguemto
            // y así sucesivamente
            String nombreFichero = args[0];
            /*
               aquí iría el código necesario
               para visulizar el fichero
            */
      }
}
```

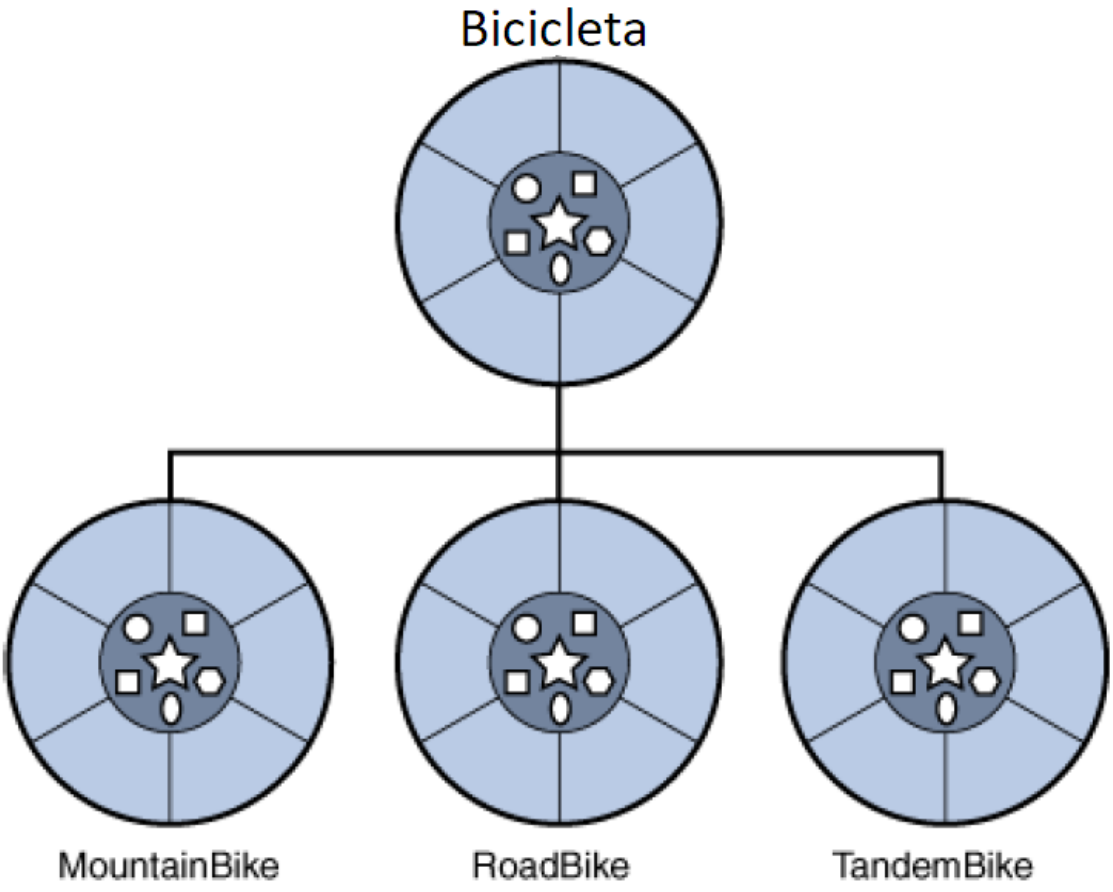
Lo ejecutaríamos, desde la línea de comandos:

- `Java Muestra /home/alumno/fichero.txt`

3. HERENCIA

Diferentes clases de objetos puede tener partes en común. Las bicicletas de montaña, bicicletas de carretera, y tándems, comparten características comunes de una bicicleta (velocidad actual, cadencia actual, marcha actual). Aparte, cada una define características adicionales que la hace diferentes: un tándem tiene dos sillines y dos manillares; las bicicletas de montaña tienen (normalmente) una horquilla con suspensión,

La programación orientada a objetos permite que una clase herede de otra el estado y comportamiento comunes. En este ejemplo, **Bicicleta** se convierte en la superclase de **MountainBike**, **RoadBike**, **yTandemBike**. En Java cada clase puede tener una única superclase, y cada superclase un número ilimitado de subclasses:



Jerarquía de clases Bicicleta

La sintaxis para crear una subclase consiste en añadir a la declaración de clase la palabra reservada **extends** seguida del nombre de la superclase. El código de la superclase no aparecer dentro del código fuente de cada subclase.

```
public class MountainBike extends Bicicleta {
      // aquí irían los campos y métodos que definen una Mountain Bike
}
```

3.1 this y super

Dentro de un método o constructor la palabra reservada **this** es una referencia al objeto actual, es decir, al objeto que está llamando al método o constructor. Con this podemos acceder a atributos y a métodos de un objeto concreto. Suele a usarse this para distinguir los parámetros de un método o constructor de los atributos de ese objeto cuándo estos se llaman igual. Por ejemplo:

```
public void setCadencia(int cadencia) {
    if (cadencia >= 0) {
        this.cadencia = cadencia;
    }
}
```

En el anterior ejemplo los parámetros del constructor y los atributos del objeto se llaman igual, por lo que tenemos que usar la palabra reservada **this** para distinguirlos.

La palabra reservada super tiene dos funciones. Si es utilizada dentro de un constructor, invoca al constructor de la clase padre, y ha de ser la primera línea de código y utilizarse una única vez.

```
public MountainBike(double recorridoSuspension, int cadencia, int velocidad,
                    int marcha, String numeroSerie) {
    super(cadencia, velocidad, marcha, numeroSerie);
    this.recorridoSuspension = recorridoSuspension;
}
```

Utilizada en otro contexto sirve para referenciar componentes de la clase padre que han sido sobrescritos en la clase hijo. Una clase hijo puede reescribir el código de métodos heredados (para indicarlo se utiliza la anotación **@Override**), de modo que las instancias de objetos de dicha clase utilizarán el código modificado en lugar del original, pero este último seguirá estando disponible con la utilización de super. Vamos a sobreescribir el método **setVelocidad** en la clase **MountainBike** de modo que no permita velocidades superiores a 40:

```
@Override
public void setVelocidad(int velocidad) {
    if (velocidad < 0 || velocidad > 40)
      this.velocidad = velocidad;
}
```

Si ahora, desde otro método de la clase **MountainBike** llamamos a **serVelocidad**, se ejecutará el anterior código, pero podemos seguir llamando al **setVelocidad** de la clase Bicicleta anteponiendo la palabra clave super:

```
void otroMetodo() {
```

```
    // llama al método de MountainBike
    setVelocidad(15);
    // llama al método de Bicicleta
    super.setVelocidad(65);
}
```

4. INTERFACES

Los objetos definen su interacción con el mundo exterior a través de métodos. Los métodos forman la **interface** del objeto con el resto del mundo.

En Java una **interface** es un grupo de métodos relacionados (sin incluir el código de los mismos) y opcionalmente atributos estáticos y finales (o sea, constantes). Es en cierto sentido similar a una clase, con la particularidad de que sus métodos carecen de código, y sus posibles atributos no permiten modificaciones. La idea es definir una especie de plantilla con aquel comportamiento que creemos debe incluir siempre una clase de determinado tipo. Las clases puede implementar interfaces (pueden implementar más de uno simultáneamente). Lo que hacemos al indicar que una clase implemente una interface es asegurarnos que disponga del código necesario para modelar el comportamiento de los métodos definidos den la interface.

Por ejemplo, pensemos que vamos a desarrollar una serie de clase para representar figuras geométricas en un plano. Queremos que para cualquier figura se puedan realizar operaciones de mostrar, ocultar, pintar, girar y desplazar además de poder calcular su área y su perímetro. Podríamos pensar en crear una clase padre que implementase todos estos métodos, pero eso no es viable ya que el código irá en función de la figura geométrica en concreto que vayamos a manejar. Otra opción es crear una interface que incluya estos métodos y para cada clase de figura geométrica que creemos, indicarle que implemente la interface. Esto hará que no nos podamos "olvidar" de suministrar el código necesario para cada operación, ya que Java nos obligará a incluirlo:

```
public interface FiguraGeometrica {
  void mostrar();
  void ocultar();
  void pintar(int color);
  void girar(double gradosGiro);
  void desplazar(double dx, double dy);
  double calculaArea();
  double calculaPerimetro();
}

class Cuadrado implements FiguraGeometrica {
    Punto punto;
    double lado;
    double gradosGiro;
    int color;
    boolena visible;
}
```

En el anterior ejemplo, el compilador dará error y nos recordará que en la clase **Cuadrado** faltan por implementar los métodos definidos en la interface

```
error: Cuadrado is not abstract and does not override abstract method
calculaPerimetro() in FiguraGeometrica
class Triangulo implements FiguraGeometrica {
[...]
```

Hasta que no incluyamos en la clase Triangulo el código correspondiente a cada uno de los métodos, nuestro programa no compilará.

```
public interface FiguraGeometrica {
    void mostrar();
    void ocultar();

    void pintar(int color);

    void girar(double gradosGiro);

    void desplazar(double dx, double dy);

    double calculaArea();

    double calculaPerimetro();
}

class Cuadrado implements FiguraGeometrica {
    Punto punto;
    double lado;
    double gradosGiro;
    int color;
    boolean visible;

    @Override
    public void mostrar() {
        visible = true;
    }

    @Override
    public void ocultar() {
        visible = false;
    }

    @Override
    public void pintar(int color) {
        this.color = color;
    }

    @Override
    public void girar(double gradosGiro) {
        this.gradrosGiro = gradosGiro;
    }

    @Override
    public void desplazar(double dx, double dy) {
        punto.x += dx;
        punto.y += dy;
    }

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

    @Override
    public double calculaPerimetro() {
        return 4 * lado;
    }
}
```

5. PAQUETES

Un paquete **(package)** es un espacio de nombres que permite organizar un conjunto de clases e interfaces relacionados. Conceptualmente se puede pensar en un paquete de manera similar a las carpetas o directorios de un Sistema Operativo. Puedes querer tener ficheros HTML en una carpeta, imágenes en otra, y scripts en otra. Debido a que el software escrito en Java puede estar compuesto por cientos o miles de clases individuales, tiene sentido mantener una organización, colocando las clases e interfaces relacionados dentro de paquetes diferentes.

La plataforma Java proporciona una enorme librería de clases (un conjunto de paquetes) disponible para el uso en cualquier programa. Esta librería es conocida como "Application Programming Interface", o "**API**". Sus paquetes representan las tareas más comúnmente asociadas con programación de propósito general. Esto permite al programador centrarse en el diseño de su aplicación particular, en lugar de en la infraestructura requerida para hacerlo funcionar.

Cuando en un programa ponemos, por ejemplo:

```
import java.util.Scanner;
```

Estamos indicando que vamos a utilizar una clase llamada **Scanner** que se encuentra en un paquete denominado **java.util** y así podemos distinguirlo además de otra posible clase con el mismo nombre creada por nosotros o perteneciente a alguna librería que estemos utilizando.

```
package es.amador_abelleira;

// mi propia clase Scanner
class Scanner {

}

public class Ejemplo {

    public static void main(String[] args) {
        // el Scanner de la librería java.util
        java.util.Scanner tec = new java.util.Scanner(System.in);
        // Mi clase Scanner
        Scanner mio = new Scanner();
        // Otro modo:
        es.amador_abelleira.Scanner mio2 = new es.amador_abelleira.Scanner();
    }

}
```

Para nuestros propios paquetes deberíamos utilizar un nombre basado en un domino registrado por nosotros, para así evitar chocar con otros con el mismo nombre. Debemos además seguir las siguientes convenciones. Supongamos que nuestros domino es **amadorabelleira.es** y queremos crear un paquete con clases, interfaces, etc. relacionados con **astronomía**:

- El nombre del paquete se define de manera inversa al dominio de la organización o grupo y se le añade un nombre relacionado con la utilidad del paquete:

  ```
  es.amador_abelleira.astronomia
  ```

- El nombre del paquete debería definirse en minúsculas. Si existen varias palabras en el nombre, se pueden separar con guión bajo (_).