# Capítulo 3

# Clases y Objetos

En los programas que ha implementado como ejemplo se han empleado fundamentalmente variables y propiedades de tipos básicos. Sin embargo, ambos elementos pueden ser también objetos. En esta práctica aprenderá a trabajar con objetos. Para empezar hay que recordar que un objeto es simplemente una instancia de una clase. La clase define cual es el comportamiento del objeto. Por lo tanto, para poder trabajar con objetos debemos previamente dejar claro el concepto de clase.

## 3.1. Clases

Como ha quedado dicho, una clase no es más que prototipo o plantilla que define las variables y los métodos comunes a un cierto tipo de objetos. Son los moldes de los cuales se pueden crear mltiples objetos del mismo tipo. La clase define las variables y métodos comunes a los objetos del mismo tipo. Después, cada objeto tendrá sus propios valores para esas variables. Por ejemplo, una caja (apilable) es algo que tiene dimensiones (largo, ancho, alto), que puede estar abierta o cerrada y que por tanto se puede abrir y cerrar:

```
public class Caja {

    // Listado de propiedades
    private double largo;
    private double ancho;
    private double alto;
    private int estado;

    // Constructor
    public Caja (double longitud, double anchura, double altura) {
        largo = longitud;
        ancho = anchura;
        alto = altura;
        estado = 0; // representa cerrada
    }

    // Metodos
    public void abrirCaja(){
```

```
        estado = 1; //representa abierta
    }
    public void cerrarCaja(){
        estado = 0; //representa cerrada
    }
    public double calcularVolumen(){
        return largo*ancho*alto;
    }
```

### 3.1.1. Declaración de una clase y sus miembros

Las clases que ha visto hasta ahora son muy sencillas, aunque todas responden a un mismo esquema (con pequeñas modificaciones). Dicho esquema es lo que denominamos sintaxis. La sintaxis para declarar una clase es la siguiente:

```
cabecera {
    listado de declaraciones propiedades y metodos
}
```

La cabecera contiene los siguientes elementos (en el orden indicado)

1. Modificadores (estos incluyen los términos `public`, `private` y `protected` que ya conoce y otros más que irá conociendo). Estos modificadores son opcionales.
2. La palabra reservada `class`.
3. El nombre de la clase (recuerde el convenio, la primera letra del nombre en mayúscula)
4. Si la clase extiende alguna superclase (esto se aclarará posteriormente) la palabra reservada `extends` seguido del nombre de la superclase.
5. Si la clase implementa algun(os) interfaces (también se aclarará) la palabra reservada `implements` seguida por un listado (separado por comas) de los interfaces que implementa la clase.

A continuación va el bloque en el que se incluyen las declaraciones de propiedades o campos (ya hemos visto que no son más que un tipo de variables) y métodos. La práctica anterior explicaba la sintaxis de declaración de propiedades. Conviene, sin embargo tener en cuenta ciertas consideraciones:

- Lo habitual es que las propiedades sean preferiblemente *private*. Esto favorece el encapsulamiento pues fuerza a que los accesos a su contenido se realicen siempre mediante métodos.
- Lo habitual es que no se asignen valores iniciales a las propiedades. Se presupone que ya se encargará de ello el método constructor.
- Recuerde que el tipo de las propiedades puede ser cualquiera de los tipos básicos que ya conoce o un objeto de cualquier clase a la que se tenga acceso. En este caso, la declaración sólo presupone la existencia de una referencia a dicho objeto. El objeto como tal no existirá hasta que se instancie con `new`.

### 3.1.2. Definición de métodos

En los ejemplos precedentes ha visto muchos métodos de modo que puede intuir la sintaxis de su declaración. Sin embargo, esta sintaxis es bastante más compleja que la utilizada en los ejemplos vistos hasta ahora. La declaración de un método tiene seis componentes. Algunos de ellos son optativos. A continuación se indican cuáles son esos seis componentes (en el orden en el que tienen que aparecer). Se indicará los que son optativos

1. Modificadores (incluyen los términos `static`, `public`, `private` y `protected` que ya conoce y otros más que irá conociendo). Estos modificadores son opcionales.

2. El tipo de retorno. En Java todos los metodos devuelven algo a no ser que se indique lo contrario. Debemos indicar el tipo del valor/objeto que se devuelve o `void` si el método no devuelve nada.

3. El nombre del método. Recuerde que, por convenio, los nombres de los métodos suelen ser verbos y que se escriben comenzado por una minúscula.

4. La lista de parámetros. Una lista separada por comas de parámetros de entrada, precedidos por su tipo, situada entre dos paréntesis. La lista puede ser vacía pero, aún en ese caso, siempre tienen que aparecer los paréntesis.

5. Una lista de excepciones tras la palabra reservada `throws`. Los métodos deben incluir información sobre que código debe ejecutarse (la excepción) en caso de que los parámetros reales no cumplan los requisitos de la especificación para los parámetros formales (se explicará posteriormente). Esta lista es opcional.

6. El cuerpo del método rodeado por dos llaves. En el cuerpo nos podemos encontrar declaraciones de variables locales (su ámbito de validez es el propio método) e instrucciones.

### 3.1.3. Sobrecarga de métodos

En Java se define el concepto *signatura del método*. Dado un método cualquiera, se denomina signatura del método a su nombre seguido por una lista, entre paréntesis, de los tipos de sus parámetros formales. Es decir, la signatura del mmétodo constructor del ejemplo sería `Caja (double, double, double)`.

Java permite la sobrecarga de métodos. Es decir, permite que dentro de una misma clase se declaren diferentes métodos con el mismo nombre. La única condición que exige Java sobre los métodos de una clase es que todos tengan diferentes signaturas.

La sobrecarga no da problemas porque Java decide el método que se está invocando en función de su signatura, no en función de su nombre.

La utilización de métodos sobrecargados no es, en general, una buena idea. Hace el código menos legible y, por lo tanto, más complejo de entender. Sin embargo hay ciertas situaciones en que es muy cómodo y útil.

En el punto anterior se ha comentado que por motivos de encapsulamiento se tiende a hacer privadas las propiedades de una clase. Obviamente, si se desea que otras clases puedan acceder/modificar los contenidos de dichas propiedades, será necesario que la

clase contenga métodos públicos que hagan el trabajo. Suelen ser métodos con nombres similares a `getPropiedad` y `setPropiedad` para cada una de las propiedades que se desea hacer visibles.

### 3.1.4. Constructores

Los constructores son unos métodos particulares que se emplean para instanciar objetos de la clase. No pueden ser `private` (en tal caso la clase no sería instanciable). Su nombre es el nombre de la clase y no se indica el tipo de retorno.

Para invocarlos debe usarse la palabra reservada `new`. Así, si en el código de una clase para la que la clase `Caja` del ejemplo es visible se desea instanciar un objeto de tipo `Caja`, se debe escribir `Caja miCaja = new Caja(10, 15, 10)`. Esta instrucción declara una referencia a un objeto de tipo `Caja` llamada `miCaja` y, posteriormente instancia un objeto `Caja` de dimensiones 10, 15, 10 y hace que la referencia `miCaja` se asocie con dicha caja.

Los constructures son un caso de métodos que están muy a menudo sobrecargados. Podemos tener en la clase `Caja` otro constructor con una signatura diferente:

```
public Caja () {
    largo = 1;
    ancho = 1;
    alto = 1;
    estado = 0; // representa cerrada
}
```

De este modo la instrucción `Caja miCaja = new Caja()` también funciona y nos genera una caja de dimensiones 1, 1, 1.

No es necesario programar siempre un constructor para cada clase, pero hay que tener mucho cuidado con ello. Cuando creamos una clase sin constructores, el propio compilador suministra un constructor por defecto para la clase. El problema es cómo decide Java ese constructor. El constructor por defecto es siempre un constructor sin parámetros. En concreto es el constructor sin parámetros de la superclase (ya se explicará el concepto). Si dicho constructor tampoco existe Java dará un error. Debe tenerse siempre en cuenta que cualquier clase que no se indique que tiene una superclase, tiene siempre una por defecto. Es la clase `Object`. Esta superclase sí tiene un constructor sin argumentos.

### 3.1.5. Paso de parámetros

Los parámetros en Java funcionan de un modo similar a los parámetros de las acciones y funciones de cualquier otro lenguaje. Simplemente, el parámetro real toma el lugar del formal durante la ejecución del código. Cuando se llama a un método deben pasársele los parámetros reales de modo que coincidan en cantidad tipo y orden con la declaración de parámetros formales.

Los parámetros de un método pueden ser de cualquier tipo: tipos básicos como `int` o `double` o referencias a objetos como `Caja` o arrays, pero debe tenerse en cuenta que el tipo de paso del parámetro es diferente en ambos casos. Los tipos de datos primitivos siempre se pasan por valor. Es decir, son siempre parámetros de entrada cuyo valor no se puede modificar dentro del método. Los objetos y arrays se pasan siempre por referencia

(de hecho, se pasa la referencia por valor), es decir, son siempre parámetros de salida o entrada/salida.

Lo dicho en el párrafo anterior puede generar una duda. ¿Que se puede hacer si se desea que un método modifique un parámetro de un tipo básico. La respuesta a este punto es que Java tiene dos versiones para cada tipo básico. La *normal*, que es la que se ha usado hasta ahora y la versión objeto. La clase `Number` del paquete `java.lang` declara una versión objetual de cada uno de los tipos básicos. Su nombre es el mismo que el de los tipos básicos pero comenzando por mayúsculas. Esta clase define métodos públicos que permiten *encajar* un valor de un tipo básico dentro de un objeto de modo que se pueda pasar por referencia. De igual modo hay métodos para *desencajar* los valores objetuales en valores de tipo básico. En muchas situaciones lo de encajar y desencajar se hace de modo automático. El paquete `java.lang` también define la clase `StringBuilder` que permite crear strings modificables y las clases `BigInteger` y `BigDecimal` para realizar cálculos de alta precisión (los valores de estas clases deben operarse siempre mediante métodos, no con operadores).

Cabe destacar que Java no admite que pasemos un método como parámetro. A pesar de ello, obviamente, si un método admite parámetros de tipo objeto, podrá invocar cualquier método público del objeto que reciba.

### 3.1.6. Número arbitrario de parámetros

Java admite la creación de métodos que admiten un número arbitrario de parámetros de un mismo tipo. Para ello se pone el tipo de los parámetros seguidos de tres puntos y un nombre. En el fondo lo que se está pasando es un array de valores del mismo. Por lo tanto, se accede a cada valor de los pasados teniendo en cuenta que es un array cuyo nombre es el indicado en la declaración. Para llamar al método se puede emplear un array o un número indeterminado de valores del tipo correspondiente.

### 3.1.7. Nombre y ámbito de los parámettros

Los parámetros de un método son, a todos los niveles, similares a las variables locales. Utilizan los mismos convenios de nombrado y tienen su misma visibilidad. Por lo tanto, el nombre de un parámetro no debe coincidir con el nombre de otro parámetro o una variable local. Si puede ser el mismo que el de parámetros o variables locales de otros métodos o que las propiedades del objeto. En este último caso, se dice que el parametro está ocultando (shadow) la propiedad. Esta forma de ocultación es perniciosa para la legibilidad del programa. Además obliga a utilizar nombres cualificados para acceder a las propiedades de la clase (como verá enseguida).

## 3.2. Objetos

### 3.2.1. Declaración, instanciación e inicialización de un objeto

Para poder trabajar con variables de tipo objeto se deben hacer tres cosas:

1. Declarar la variable. Se hace de igual manera como se declaran las variables de tipos básicos. Es decir, se pone el nombre de la clase a la que pertenece el objeto

seguido por el nombre que tendrá la referencia al objeto en el programa. Declarar una variable de tipo objeto no crea la variable. Simplemente crea una referencia para objetos del tipo indicado.

2. Instanciar la variable. Para ello hace falta utilizar la palabra reservada `new` seguido del algún constructor de la clase. El operador `new` asigna una cantidad de memoria adecuada para el objeto instanciado y devuelve una referencia para el objeto. El operador `new` requiere a continuación un argumento que tiene que ser el constructor de alguna clase visible. La referencia que devuelve new suele asignarse a una variable del tipo adecuado, aunque también podría usarse como parámetro real en la invocación de un método.

3. Inicializar el objeto. La inicialización consiste en darle valores iniciales a las propiedades del objeto. En el fondo se hace simultaneamente a la instanciación. Al invocar el constructor este se encargará de ejecutar el código que crea los valores iniciales (si así se ha programado).

### 3.2.2. Uso de objetos

Hay dos formas de acceder a las propiedades de un objeto. Desde dentro del mismo objeto (más bien dicho desde su clase) se puede acceder a las propiedades mediante su nombre (ya sabemos que es único). Para acceder a las propiedades visibles de un objeto desde fuera del objeto se usa el operador `.`, para ello escribimos el nombre de una referencia al objeto seguida de `.` y seguida del nombre de la propiedad. Lo dicho es cierto desde el momento en que el objeto se ha instanciado, por lo que en Java es correcto programar asignaciones como: `double longitud = new Caja(100, 10, 1).largo;` (bueno, sería posible si esta propiedad no fuese private. El resultado sería que instanciaríamos una nueva caja de las dimensiones indicadas y después cargaríamos su longitud en la variable indicada.

Para invocar métodos de un objeto se emplea el mismo operador. Es decir, para ejecutar un método de un objeto ponemos el nombre de la referencia al objeto seguido del operador `.`, seguido del nombre al metodo a invocar y seguido de un listado de parámetros entre paréntesis. Obviamente, el listado de parámetros debe coincidir en longitud y tipos con alguna de las signatiras del método invocado. También es posible escribir cosas como `double vol = new Caja(100, 10, 1).calcularVolumen;`.

### 3.2.3. El recolector de basura

En alguno de los ejemplos anteriores se han instanciado objetos sin que queden referenciados por ninguna variable. ¿Que ocurre con esos objetos? Obviamente no podemos volver a acceder a ellos dado que no sabemos como acceder. Podemos ver que en el fondo no es más que una zona de memoria no accesible. En muchos lenguajes esto constituye un problema. Es labor del programador liberar explicitamente esa memoria. En Java el programador puede olvidarse de estos objetos.

La máquina virtual Java incorpora el *recolector de basura*. Se trata de una rutina que se ejecuta cada cierto tiempo. Su cometido es detectar qué objetos no son referenciados por ninguna variable para eliminarlos de la memoria. En muchos casos un objeto queda sin referencias de modo implícito, por ejemplo, cuando se ha sálido del ámbito que lo declaró.

A veces el programador lo puede hacer de modo explícito haciendo que las variables que referencian a dicho objeto cojan el valor `null` (nótese que deben ponerse a `null` todas las variables que referencian al objeto).

No hay forma de controlar cuando se ejecuta el recolector de basura. Depende de la máquina virtual. Por lo tanto, no hay modo de saber si un objeto al que no hay ninguna referencia sigue ocupando memoria o no.

## 3.3. Otros aspectos a considerar sobre las clases

### 3.3.1. Lo que un método devuelve

Un método finaliza (y por tanto devuelve el control al punto donde se invocó) cuando completa su código, alcanza una instrucción `return` o lanza una excepción (ya se explicarán).

- Si el método se declaró como `void` la instrucción `return` puede no existir o, si existe, no devolver nada. Es decir, debe ser de la forma `rerturn;`. Esto puede ser útil para simplificar el código de salida en caso de iteraciones anidadas.
- Si el método se declaro indicando que devuelve un tipo básico, debe existir una instrucción `return` seguida de una variable o una constante del tipo indicado o de una expresión que se evalúe en el tipo declarado (como ocurre en el métod `calcularVolumen` del ejemplo).
- Si el método se declaró indicando que devuelve una referencia a una clase, entonces puede devolverse una referencia a un objeto instanciado de dicha clase o de alguna otra clase que herede de dicha clase. Por ejemplo, en el caso de que `Hombre` sea una subclase de `Persona` cualquier método programado para devolver una persona puede devolver realmente un `Hombre`.
- También pueden declararse métodos que devuelvan un interfaz en lugar de una clase. Con ello se indica que el valor devuelto debe ser una referencia a un objeto de una clase que implemente el interfaz.

### 3.3.2. La palabra reservada `this`

Existen situaciones en que necesitamos hacer referencia a un objeto desde su mismo código. Para ello podemos emplear la palabra reservada `this`. Los usos de esta palabra son variados:

- Como se ha explicado, el nombre de un parámetro de un método puede ocultar alguna propiedad de la clase donde está definido. Aún así tenemos acceso a dicha propiedad por medio de `this`. Por ejemplo, el código:

```
public class Punto{
    public int x = 0;
    public int y = 0;
    //constructor
```

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

es equivalente a:

```
public class Punto{
    public int x = 0;
    public int y = 0;
    //constructor
    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Aunque cada argumento del método ensombrece una propiedad, aún podemos acceder a ellas.

- Desde un constructor de una clase también se puede utilizar `this` para invocar otro constructor de la misma clase. Por ejemplo, en:

```
public class Rectangulo {
    private int x, y;
    private int width, height;

    public Rectangulo() {
        this(0, 0, 0, 0);
    }
    public Rectangulo(int width, int height){
        this(0, 0, width, height);
    }
    public Rectangulo(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    ...
}
```

se utiliza esta característica para acelerar la construcción de un constructor sobrecargado proporcionando argumentos para los constructores con argumentos (también se abusa del uso anteriormente comentado).

Cabe destacar que el uso de this dentro de un constructor para llamar a otro constructor, si aparece, sólo puede hacerlo en la primera línea del constructor.

### 3.3.3. Miembros de clase vs de instancia

En alguos ejemplos aparece el modificador `static`. Este modificador indica que el miembro (propiedad o método) al que se le aplica es un miembro de clase, no de instancia. ¿Que significa eso?

En general, cuando creamos diferentes objetos de una clase, por cada objeto se reserva en memoria espacio para cada una de las propiedades. De este modo los valores de una misma propiedad pueden ser diferentes para cada objeto. Sin embargo, cuando declaramos una propiedad como `static` sólo se reserva un espacio cuyo valor comparten todos los objetos de la clase. A estas variables se les suele denominar variables de clase.

Las variables de clase tienen otra peculiaridad. Existen incluso aunque no se haya instanciado ningún objeto de la clase. De hecho, existen sólo por utilizar la clase. Por lo tanto deben inicializarse directamente en su declaración, no dentro de un constructor. De hecho hay que tener mucho cuidado con ellas en los constructores dado que debemos tener siempre en cuenta que tienen un valor que es desconocido cuando se instancia el objeto.

Para acceder a una variable de clase podemos hacerlo de dos formas. La primera es de igual forma a como accedemos a cualquier otra propiedad de un objeto instanciado. La otra, que es la que se recomienda utilizar, consiste en llamarla poniendo `NombreDeClase.variableDeClase`

A continuación tiene un ejemplo. ¿Que cree que hace las variables `numeroDeIds` e `id` de esta clase?. Si no lo ve, puede probar a instanciar unos cuantos objetos, imprimiendo después este valor para entenderlo.

```
public class Caja{
    private int largo;
    private int ancho;
    private int alto;
    private int id;
    private static int numeroDeIds = 0;

    public Caja(int l, int a, int al){
        largo = l;
        ancho = an;
        alto = al;
        id = ++numeroDeIds;
    }
    public int getId() {
        return id;
    }
    ...
}
```

El modificador `static`, como ya sabe, se puede también aplicar a métodos. La idea es la misma, disponer de un método asociado a la clase cuya existencia no depende de la existencia de alguna instancia de objetos de la clase. Este tipo de métodos es necesario a veces para poder acceder a variables de clase. Debe tenerse en cuenta que:

- Los métodos de instancia (no estáticos) pueden acceder a variables y métodos de instancia directamente.

- Los métodos de instancia (no estáticos) pueden acceder a variables y métodos de clase directamente.
- Los métodos de clase pueden acceder a variables y métodos de clase directamente.
- Los métodos de clase no pueden acceder a variables y métodos de instancia directamente. Tampoco pueden usar `this`.

El modificador `static` también permite, combinado con `final`, definir constantes (`final` significa que la variable a la que se aplica no es modificable). Por ejemplo, podemos declarar el valor de pi como `static final double PI = 3.141596;`. Como se ve en el ejemplo, los nombres de las constantes se escriben siempre en mayúsculas. Si el nombre tiene más de una palabra se usa _ para separar las palabras. Debe tenerse en cuenta que cuando definimos constantes de tipo string o tipos primitivos, el compilador usará dicho valor a la hora de generar bytecodes, por lo que si queremos cambiar el valor deberemos recompilar.

### 3.3.4. Inicializando Campos

Ya hemos visto que las propiedades de una clase pueden inicializarse en su declaración. Aunque esta forma de inicializar es poco potemte (si la inicialización precisa de cierta programación. por ejemplo un bucle para un array, no es posible) es muy cómoda. Por eso Java define una forma similar de inicializar que es válida también para variables de clase. Se denomina *bloque de inicialización estático*. Estos bloques son bloques precedidos por la palabra `static`.

```
class Ejemplo {
    ...
    static {
        //codigo de iniciacion
    }
    ...
}
```

Los bloques de inicialización estática pueden aparecer en cualquier lugar (aunque es costumbre poner las propiedades antes de los métodos, no es necesario) e incluso puede haber más de uno. El sistema de ejecución asegura que se llaman en el orden en el que aparecen en el código.

Aunque esta posibilidad existe, es mejor usar un método privado estático:

```
class Ejemplo {
    public static tipoDeVariable variable = inicialia();
    private static tipoDeVariable inicialia() {
        //codigo de iniciacion
    }
    ...
}
```

A pesar de ello, los bloques de inicialización ienen sentido en ciertas circunstancias. Normalmente el código para inicializar las propiedades de un objeto se pone en el constructor. Es posible que algunas instrucciones deban ejecutarse en cualquiera de los constructores (sobrecargados, obviamente) de una clase. Para ello empleamos los bloques de inicialización. Son iguales a los estáticos pero sin la palabra `static`. El compilador de Java copia estos bloques en todos los constructores de una clase.

Un último problema referente a la inicialización viene por la herencia (se verá en la siguiente práctica). Si queremos que las subclses usen un método de inicialización de la superclase sin problemas es preferible indicar que el método es `final`:

```
class Ejemplo {
    private Tipo variable = inicilizaInstancia ();
    protected final Tipo inicilizaInstancia () {
        //codigo de inicializacion
    }
    ...
}
```

## 3.4. Subclases

Java permite definir clases dentro de una clase. Es decir crear clases similaesr a:

```
class ClaseExterna {
    ...
    class ClaseAnidada {
        ... }
    ...
}
```

Una clase anidada es un miembro más de la clase donde se define. Una clase anidada no estática tiene acceso al resto de elementos de la clase donde se declara, incluso a los privados. Para evitar esto se puede declarar la clase interna como `static`.

Una clase interna puede declararse como `public`, `protected`. `private` o *package privative*.

Las subclases vienen bien cuando queremos agrupar un conjunto de clases que sabemos se usan sólo dentro de una clase. Dejar estas clases como membros de unpaquete podr´ a dificultar su comprensión. Además permiten aumentar el nivel de encapsulación de las aplicaciones pues podemos hacer privados los miembros a los que sólo deb´ an acceder las clases internas.

A pesar de estas ventajas, las subclases son complicadas de usar y entender. Una clase interna estática es similar a un método de clase. No puede acceder directamente a propiedades o métodos de la clase externa. Sólo puede hacerlo mediante una referencia a objetos de la clase externa. En el fondo, una clase anidada estática se comporta como si no fuese interna.

Para acceder a una clase anidada estática debemos usar el nombre de la clase. Por tanto,podemos instanciarla con `ClaseExterna.ClaseAnidadaEstatica objetoAnidado = new ClaseExterna.ClaseAnidadaEstatica objetoAnidado()`.

Las clases internas no estáticas son como las variables de instancia. Tienen acceso directo a propiedades y métodos de la clase externa y no pueden definir miembros estáticos (obviamente, pues los objetos de su clase sólo existen dentro de instancias de la clase externa). Por tanto, para instanciar un objeto de la clase interna debemos instanciar uno de la externa `OuterClass.InnerClass innerObject = outerObject.new InnerClass();`

El problema de usar clases estáticas internas es que se complica el funcionamiento de algunos mecanismos, como, por ejemplo el de ensombrecido (shadow). No odemos acceder a una propiedad ensombrecida directamente, debemos usar `this`, pero ¿Que sucede ahora que podemos tener diferentes niveles de *sombra*.

Veamos un ejemplo. ¿Cual cree que será la salida de el siguiente código?.

```
public class ShadowTest {

    public int x = 0;

    class NivelUno {

        public int x = 1;

        void metodoNivelUno (int x) {
            System.out.println("x = " + x);
            System.out.println("this.x = " + this.x);
            System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);
        }
    }

    public static void main(String... args) {
        ShadowTest st = new ShadowTest();
        ShadowTest.NivelUno fl = st.new NivelUno();
        fl.metodoNivelUno(23);
    }
}
```

La cosa se complica si además le digo que existen dos tipos especiales de clases internas no estáticas denominadas clases locales (definidas dentro de un bloque de, por ejemplo, dentro de un método)y clases anónimas (locales sin nombre). Mejor dejamos estos extremos para otros cursos o para que usted mismo las estudie por su cuenta.

## 3.5. Tipos Enumerados

Los tipos enumerados son ina *novedad* e Java. Se incluyeron en la version 5 (la actual es la 7). En realidad, son clases especializadas que permiten definir y usar conjuntos de constantes, como los días de la semana o los meses del año. Para definirlos se usa la palabra reservada `enum`, por ejemplo:

```
public enum Day {
    LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO
}
```

Note que los valores del enumerado, dado que son constantes, se escriben con mayúsculas.

Los enumerados de Java son mucho más potentes que los de otros lenguajes. Como hemos dicho, son en realidad clases (denominadas tipos enumerados), por lo que su cuerpo puede incluir incluso métodos. De hecho, como en realidad son subclases de clases predefinidas (extienden implicitamente `java.lang.Enum`, el compilador incluye algunos métodos heredados, como el método estático `values` que devuelve un array con los valores del enumerado en el orden en el que se escribieron. Esto permite programar cosas como el ejemplo que escribo a continuación. Compilelo y pruébelo:

```
public enum Planeta {
    MERCURIO (3.303e+23, 2.4397e6),
    VENUS (4.869e+24, 6.0518e6),
    TIERRA (5.976e+24, 6.37814e6),
    MARTE (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27, 7.1492e7),
    SATURNO (5.688e+26, 6.0268e7),
    URANO (8.686e+25, 2.5559e7),
    NEPTUNO (1.024e+26, 2.4746e7);

    private final double masa; // en kg
    private final double radio; // en metros
    Planeta(double masa, double radio) {
        this.masa = masa;
        this.radio = radio;
    }
    private double getMasa() { return masa; }
    private double getRadio() { return radio; }

    // constante gravitatoria universal (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;

    double gravedadSupercie() {
        return G * masa / (radio * radio);
    }
    double pesoEnSuperficie(double otraMasa) {
        return otraMasa * gravedadSupercie();
    }
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println(Üsa: java Planeta <peso_en_la_tierra>");
            System.exit(-1);
        }
        double pesoEnTierra = Double.parseDouble(args[0]);
        double masa = pesoEnTierra/TIERRA.gravedadSupercie();
        for (Planeta p : Planeta.values())
        System.out.printf("Tu peso en %s es %f %n", p, p.pesoEnSuperficie(masa));
    }
```

}

## 3.6. Prácticas a desarrollar

**Ejercicio 3**: En la práctica 1 hizo un programa que simulaba el lanzamiento de un par de dados. Vamos a reutilizar parte de aquel código para programar un juego complejo. En el juego se enfrentan dos jugadores (le permito que sean jugador 1y jugador 2 o que el programa solicite nombres para los jugadores). Una partida del juego consiste en cinco turnos. En cada turno cada jugador tirará dos dados (recuerde que se puede simular el lanzamiento de un dado mediante la elección de un entero comprendido en el conjunto {1, 2, 3, 4, 5, 6} aleatoriamente[1]). En cada turno gana el jugador que obtenga un mayor valor al sumar lo que salga en cada dado (obviamente si la suma es igual no gana ninguno).

Vencerá la partida el jugador que gane más turnos con una excepción. Hay una jugada denominada *ojos de tigre* que consiste en obtener 2 unos. En caso de que algún jugador obtenga en una tirada esa jugada habrá ganado la partida (a no ser que su contricante la obtenga también en el mismo turno).

Un ejemplo del juego podría ser:

```
Introduzca el nombre del jugador1: Kepa Sakolegui
Introduzaca el nombre del jugador 2: Sevelinda Parada
Comienza la partida
    Turno 1
        Juega Kepa
            El lanzamiento del primer dado es 3
            El lanzamiento del segundo dado es 5
        El total de los dos lanzamientos es 8
        Juega Sevelinda
            El lanzamiento del primer dado es 1
            El lanzamiento del segundo dado es 2
        El total de los dos lanzamientos es 3
    Kepa ha ganado el turno 1
    Turno 2
        Juega Kepa
            El lanzamiento del primer dado es 1
            El lanzamiento del segundo dado es 1
        Enhorabuena, has obtenido ojos de tigre
        Juega Sevelinda
            El lanzamiento del primer dado es 1
            El lanzamiento del segundo dado es 1
        Enhorabuena, has obtenido ojos de tigre
    Ha habido empate en el turno 2
    Turno 3
        Juega Kepa
            El lanzamiento del primer dado es 6
```

[1]Visite en el API de Java la clase `Math` para ver como obtener un número aleatorio.

```
            El lanzamiento del segundo dado es 6
        El total de los dos lanzamientos es 12
        Juega Sevelinda
            El lanzamiento del primer dado es 1
            El lanzamiento del segundo dado es 1
        Enhorabuena, has obtenido ojos de tigre
    Sevelinda ha ganado el turno 2 con ojos de gato
Sevelinda gana la partida
```

De las muchas formas de hacer este programa quiero una que utilice profusamente los conceptos que se han visto. Debe crear la clase `Dado.class` con un valor privado y un método público para lanzarlo y otro para ver el resultado. También debe crear las clases `Turno.class` cuyo contenido le dejo pensar y `Partida.class`, que es obviamente un array de turnos. Ninguna de las clases anteriores puede tener un método main. Por último, la clase `Casino.class` contendrá el main así como la identificación de jugadores.