Clases y objetos

Clase

Implementación de un tipo de dato.

Una clase sirve tanto de *módulo* como de *tipo*

- **Tipo**: Descripción de un conjunto de objetos (equipados con ciertas operaciones).
- **Módulo**: Unidad de descomposición del software.

Objeto

Instancia de una clase:
Unidad atómica que encapsula estado y comportamiento.

- Un objeto puede caracterizar una entidad física (un teléfono, un interruptor, un cliente) o una entidad abstracta (un número, una fecha, una ecuación matemática).
- Todos los objetos son instancias de una clase: Los objetos se crean por instanciación de las clases.
- Todos los objetos de una misma clase (p.ej. coches) comparten ciertas características: sus atributos (tamaño, peso, color, potencia del motor…) y el comportamiento que exhiben (aceleran, frenan, curvan…).

Todo objeto tiene…

- **Identidad** (puede distinguirse de otros objetos)
- **Estado** (datos asociados a él)
- **Comportamiento** (puede hacer cosas)

Las diferentes instancias de cada clase difieren entre sí por los valores de los datos que encapsulan (sus atributos).

Dos objetos con los mismos valores en sus atributos pueden ser diferentes.

TODOS los objetos de una misma clase usan el mismo algoritmo como respuesta a mensajes similares.

El algoritmo empleado como respuesta a un mensaje (esto es, el método invocado) viene determinado por la clase del receptor.

Una clase es una descripción de un conjunto de objetos similares.

Al programar, definimos una clase para especificar cómo se comportan y mantienen su estado los objetos de esa clase.

A partir de la definición de la clase, se crean tantos objetos de esa clase como nos haga falta

Clases en Java

Cada clase en Java:

- Se define en un fichero independiente con extensión `.java`.
- Se carga en memoria cuando se necesita.

La máquina virtual Java determina en cada momento las clases necesarias para la aplicación y las carga en memoria.

El programa puede ampliarse dinámicamente (sin tener que recompilar): La aplicación no es un bloque monolítico de código.

Para definir una clase en Java se utiliza la palabra reservada `class`, seguida del nombre de la clase (un identificador):

```
public class MiClase
{
  …
}
```

NOTA: Es necesario que indiquemos el modificador de acceso `public` para que podamos usar nuestra clase "desde el exterior".

Los límites de la clase se marcan con llaves `{ … }`

? Se debe sangrar el texto que aparece entre las llaves para que resulte más fácil delimitar el ámbito de los distintos elementos de nuestro programa. De esta forma se mejora la legibilidad de nuestro programa.

Acerca del nombre de las clases

El nombre de una clase debe ser un identificador válido en Java.

Por convención, los identificadores que se les asignan a las clases en Java comienzan con mayúscula.

Además, en Java, las clases públicas deben estar definidas en ficheros con extensión `.java` cuyo nombre coincida exactamente con el identificador asignado a la clase (¡ojo con las mayúsculas y las minúsculas, que en Java se consideran diferentes!)

Errores comunes

- Cuando el nombre de la clase no coincide con el nombre del fichero, el compilador nos da el siguiente error:

```
Public class MiClase must be defined
in a file called MiClase.java
```

  donde `MiClase` es el nombre de nuestra clase.

- Las llaves siempre deben ir en parejas: Si falta la llave de cierre `}` el compilador da un error:

```
‘}’ expected
```

  Lo mismo ocurre si se nos olvida abrir la llave (`{`):

```
‘{’ expected
```

? Es una buena costumbre cerrar inmediatamente una llave en cuanto se introduce en el texto del programa. Después se posiciona el cursor entre las dos llaves y se completa el texto del programa.

Uso del compilador `javac`

Al compilar un programa con `javac`,
el compilador nos muestra la lista de errores que se han detectado.

Para cada error, el compilador nos ofrece los siguientes datos:

- El nombre del fichero donde está el error.
- La línea en la que se ha detectado el error.
- Un mensaje descriptivo del tipo de error (en inglés).
- Una reproducción de la línea **donde se detectó** el error.
- La posición exacta donde se detectó el error en la línea.

Ejemplos

Al compilar el fichero `Error.java`, cuyo contenido es

```
public class 2Error
{
}
```

el compilador nos da los siguientes errores:

```
Error.java:1: malformed floating point literal
public class 2Error
             ^
Error.java:4: '{' expected

^
2 errors
```

El primer error se produce porque el identificador de la clase no es válido y el segundo error es consecuencia del anterior.

Si cambiamos el identificador de la clase por un identificador válido en Java, se produce un error si el nombre de la clase con el nombre del fichero:

```
Error.java:1: class Error3 is public,
should be declared in a file named Error3.java
public class Error3
       ^
1 error
```

El compilador también nos dará un error si se nos olvida el punto y coma del final de una sentencia o de una declaración, tal como sucede en el siguiente ejemplo:

```
public class Error
{
  int x
}
```

En este caso, el error se detecta en la línea siguiente a la declaración de la variable `x`:

```
Error.java:4: ';' expected
}
^
1 error
```

En la mayoría de los lenguajes de programación, Java incluido, el compilador ignora los espacios y líneas en blanco que introduzcamos en el texto del programa.

- Los espacios y líneas en blanco los usaremos nosotros para mejorar la legibilidad del texto del programa.

Cuando el compilador nos da un error, debemos comprobar la línea en la que se detecta el error para corregirlo. Si esa línea no contiene errores sintácticos, entonces debemos analizar las líneas que la preceden en el texto del programa.

Encapsulación de datos (variables) y operaciones (métodos)

Objeto = Identidad + Estado + Comportamiento

Identidad

La identidad de un objeto lo identifica unívocamente:

- Es independiente de su estado.
- No cambia durante la vida del objeto.

Estado

El estado de un objeto viene dado por los valores de sus atributos.

- Cada atributo toma un valor en un dominio concreto.
- El estado de un objeto evoluciona con el tiempo
- Los atributos de un objeto no deberían ser manipulables directamente por el resto de objetos del sistema (*ocultamiento de información*):
  - Se protegen los datos de accesos indebidos.
  - Se distingue entre interfaz e implementación.
  - Se facilita el mantenimiento del sistema.

Interfaz vs. implementación
Los objetos se comunican a través de interfaces bien definidas sin tener que conocer los detalles internos de implementación de los demás objetos.

Ejemplo: Un coche se puede conducir…
- Sin saber exactamente de qué partes consta el motor ni cómo funciona éste (implementación).
- Basta con saber manejar el volante, el acelerador y el freno (interfaz).

Comportamiento

Los métodos que definen el comportamiento de un objeto:

- Agrupan las competencias del objeto (responsabilidades)
- Describen sus acciones y reacciones.

  Las acciones realizadas por un objeto son consecuencia directa de un estímulo externo (un mensaje enviado desde otro objeto) y dependen del estado del objeto.

El interfaz de un objeto ha de establecer un contrato, un conjunto de condiciones precisas que gobiernan las relaciones entre una clase proveedora y sus clientes (*diseño por contrato*).

Estado y comportamiento están relacionados.

Ejemplo
Un avión no puede aterrizar (acción) si no está en vuelo (estado)

Representación gráfica de una clase (notación UML)

Una clase se representa con un rectángulo dividido en tres partes:

- El nombre de la clase
  (identifica la clase de forma unívoca)
- Sus atributos
  (datos asociados a los objetos de la clase)
- Sus operaciones
  (comportamiento de los objetos de esa clase)

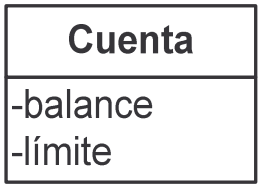
IMPORTANTE: Las clases se deben identificar con un nombre que, por lo general, pertenecerá al vocabulario utilizado habitualmente al hablar del problema que tratamos de resolver.

Representación de una clase:


```
public class Cuenta
{
  …
}
```

Declaración en Java de una clase.

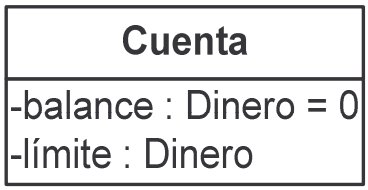
Representación de una clase con sus atributos:




```
public class Cuenta
{
  private double balance;  // Saldo
  private double limit;    // Límite
  …
}
```

Las variables de instancia
sirven para especificar en Java los atributos de la clase.

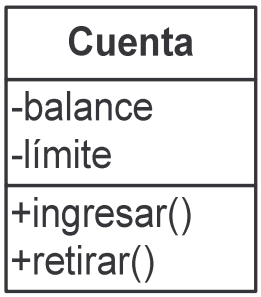
Representación del tipo y de los valores por defecto de los atributos:




```
public class Cuenta
{
  private double balance = 0;
  private double limit;
  …
}
```

NOTA: Aquí hemos decidido utilizar el tipo primitivo `double` para representar cantidades de dinero. En un programa en el que no se pudiesen permitir errores de redondeo en los cálculos, deberíamos utilizar otro tipo más adecuado (por ejemplo, `BigDecimal`).

Representación de las operaciones de una clase:

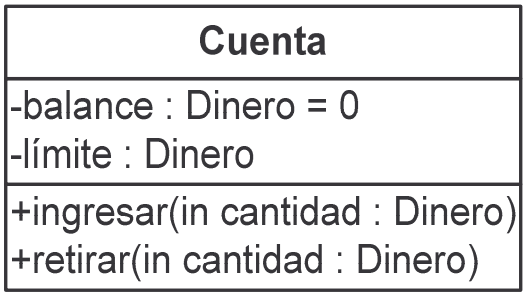



```
public class Cuenta
{
  private double balance = 0;
  private double limit;

  public void ingresar (…) …
  public void retirar (…)  …
}
```

Los métodos implementan en Java las operaciones características de los objetos de una clase concreta.

Especificación completa de la clase:



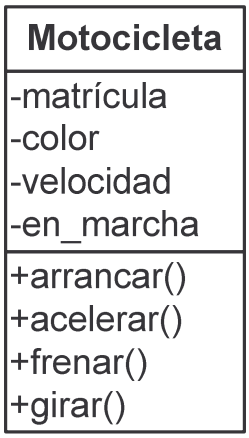

```
public class Cuenta
{
  private double balance = 0;
  private double limit;

  public void ingresar (double cantidad)
  {
     balance = balance + cantidad;
   }

  public void retirar (double cantidad)
  {
     balance = balance – cantidad;
  }
}
```

Clase de ejemplo

Representación gráfica en UML:



Definición en Java:
Fichero `Motocicleta.java`

```
public class Motocicleta
{
  // Atributos (variables de instancia)
  private String matricula;  // Placa de matrícula
  private String color;      // Color de la pintura
  private int velocidad;     // Velocidad actual (km/h)
  private boolean en_marcha; // ¿moto arrancada?

  // Operaciones (métodos)
  public void arrancar ()
  { … }
  public void acelerar ()
  { … }
  public void frenar ()
  { … }
  public void girar (float angulo)
  { … }
}
```

Uso de objetos

El operador .

El operador `.` (punto) en Java
nos permite acceder a los distintos miembros de una clase:

```
objeto.miembro
```

Cuando tenemos un objeto de un tipo determinado y queremos acceder a uno de sus miembros sólo tenemos que poner el identificador asociado al objeto (esto es, el identificador de una de las variables de nuestro programa) seguido por un punto y por el identificador que hace referencia a un miembro concreto de la clase a la que pertenece el objeto.

¿Cómo se comprueba el estado de un objeto?

Accediendo a las variables de instancia del objeto

```
objeto.atributo
```

Por ejemplo, `cuenta.balance` nos permitiría acceder al valor numérico correspondiente al saldo de una cuenta, siempre y cuando `cuenta` fuese una instancia de la clase `Cuenta`.

¿Cómo se le envía un mensaje a un objeto?

Invocando a uno de sus métodos.

```
objeto.método(lista de parámetros)
```

La llamada al método hace que el objeto realice la tarea especificada en la implementación del método, tal como esté definida en la definición de la clase a la que pertenece el objeto.

Ejemplos

```
cuenta.ingresar(150.00);
```

- Si `cuenta` es el identificador asociado a una variable de tipo `Cuenta`, se invoca al método `ingresar` definido en la clase `Cuenta` para depositar una cantidad de dinero en la cuenta.

- La implementación del método `ingresar` se encarga de actualizar el saldo de la cuenta, sin que nosotros nos tengamos que preocupar de cómo se realiza esta operación.

```
System.out.println("Mensaje");
```

- `System` es el nombre de una clase incluida en la biblioteca de clases estándar de Java.

- `System.out` es un miembro de la clase `System` que hace referencia al objeto que representa la salida estándar de una aplicación Java.

- `println()` es un método definido en la clase a la que pertenece el objeto `System.out`.

- La implementación del método `println()` se encarga de mostrar el mensaje que le pasamos como parámetro y hace avanzar el cursor hasta la siguiente línea (como si pulsásemos la tecla ↵).

- `System.out.println()` es una llamada a un método, el método `println()` del objeto `System.out`

- La línea completa forma una sentencia (terminada con un punto y coma) que delega en el método `println()` para que éste se encargue de mostrar una línea en pantalla.

Creación de objetos

Antes de poder usar un objeto hemos de crearlo…

El operador new

El operador `new` nos permite crear objetos en Java.

```
Tipo identificador = new Tipo();
```

Si escribimos un programa como el siguiente:

```
public class Ingreso
{
  public static void main (String args[])
  {
    Cuenta cuenta;                    // Error

    cuenta.ingresar(100.00);
  }
}
```

El compilador nos da el siguiente error:

```
Ingreso.java:7:
variable cuenta might not have been initialized
    cuenta.ingresar(100.00);
    ^
```

Hemos declarado una variable que, inicialmente, no tiene ningún valor. Antes de utilizarla, deberíamos haberla inicializado (con un objeto del tipo adecuado):

```
Cuenta cuenta = new Cuenta();
```

Observaciones

- Se suele crear una clase aparte, que únicamente contenga un método `main`, como punto de entrada de la aplicación.
- En la implementación del método `main` se crean los objetos que sean necesarios y se les envían mensajes para indicarles lo que deseamos que hagan.

Constructores

Cuando utilizamos el operador `new` acompañado del nombre de una clase, se crea un objeto del tipo especificado (una instancia de la clase cuyo nombre aparece al lado de `new`).

Al crear un objeto de una clase concreta, se invoca a un método especial de esa clase, denominado **constructor**, que es el que se encarga de inicializar el estado del objeto.

Constructor por defecto

Por defecto, Java crea automáticamente un constructor sin parámetros para cualquier clase que definamos.

```
public class Cuenta
{
  // Constante
  public static final double LIMITE_NORMAL = 300.00;

  // Variables de instancia
  private double balance = 0;
  private double limit = LIMITE_NORMAL;

  // Métodos
  ...
}
```

> Al crear un objeto de tipo `Cuenta` con `new Cuenta()`, se llama al constructor por defecto de la clase `Cuenta`, con lo cual se crea un objeto de tipo `Cuenta` cuyo estado inicial será el indicado en la inicialización de las variables de instancia `balance` y `limit`.

Constructores definidos por el usuario

Los lenguajes de programación nos permiten definir constructores para especificar cómo ha de inicializarse un objeto al crearlo.

> El nombre del constructor ha de coincidir con el nombre de la clase.

Podemos definir un constructor
para inicializar las variables de instancia de una clase,
en vez de hacerlo en la propia declaración de las variables de instancia:

```
public class Cuenta
{
  // Constante
  public static final double LIMITE_NORMAL = 300.00;

  // Variables de instancia
  private double balance;
  private double limit;

  // Constructor sin parámetros
  public Cuenta ()
  {
    this.balance = 0;
    this.limit   = LIMITE_NORMAL;
  }

  // Métodos
  ...
}
```

> La palabra reservada `this` hace referencia al objeto
> que realiza la operación cuya implementación especifica el método.

También podemos definir constructores con parámetros:

```
public class Cuenta
{
  // Constante
  public static final double LIMITE_NORMAL = 300.00;

  // Variables de instancia
  private double balance;
  private double limit;

  // Constructor con parámetros
  public Cuenta (double limit)
  {
    this.balance = 0;
    this.limit = limit;
  }

  // Métodos
  ...
}
```

Ahora, para crear un objeto de tipo Cuenta, hemos de especificar los parámetros adecuados para su constructor:

```
  Cuenta cuenta = new Cuenta(Cuenta.LIMITE_NORMAL);
```

O bien

```
  Cuenta cuentaVIP = new Cuenta(6000.00);
```

La cuenta VIP tendrá un límite de 6000€ en vez del límite normal.

En cuanto definimos un constructor, ya no podemos utilizar el constructor por defecto de la clase (el constructor sin parámetros que Java crea automáticamente si no especificamos ninguno).

Si sólo está definido el constructor anterior

```
public Cuenta (double limit)
```

al crear un objeto con

```
Cuenta cuenta = new Cuenta();
```

el compilador de Java nos da el siguiente error:

```
CuentaTest.java:5: cannot find symbol
symbol  : constructor Cuenta()
location: class Cuenta
    Cuenta cuenta = new Cuenta();
                    ^
1 error
```

porque no existe ningún constructor
sin parámetros definido para la clase `Cuenta`.

Para facilitarnos la creación de objetos,
Java nos permite definir varios constructores para una misma clase
(siempre y cuando tengan parámetros diferentes).

De forma que podemos incluir los dos constructores
en la implementación de la clase `Cuenta` y escribir lo siguiente:

```
public class CuentaTest
{
  public static void main (String args[])
  {
    Cuenta cuentaNormal = new Cuenta();
    Cuenta cuentaVIP    = new Cuenta(6000.00);
    …
  }
}
```

Referencias

Cualquier tipo que definamos en Java con una clase
es un tipo no primitivo.

Cuando declaramos una variable de un tipo primitivo en Java, estamos reservando espacio en memoria para almacenar un **valor** del tipo correspondiente.

Sin embargo,
cuando declaramos una variable de un tipo no primitivo en Java,
lo único que hacemos es reservar una zona en memoria donde se almacenará una **referencia** a un objeto del tipo especificado (y no el objeto en sí, de ahí la necesidad de utilizar el operador `new`).

Inicialización por defecto de los objetos en Java

Cuando se crea un objeto con el operador `new`, por defecto:

- Las variables de instancia de tipo numérico (`byte, short, int, long, float y double`) se inicializan a 0.
- Las variables de instancia de tipo `char` se inicializan a `'\0'`.
- Las variables de instancia de tipo `boolean` se inicializan a `false`.
- Las variables de instancia de cualquier tipo no primitivo se inicializan a `null` (una palabra reservada del lenguaje que indica que la referencia no apunta a ninguna parte).

IMPORTANTE

Para acceder a un miembro de un objeto
(leer el valor de una variable de instancia o invocar un método)
hemos de tener una referencia a un objeto distinta de `null`

Uso de sentencias de asignación

Cuando usamos datos de tipos primitivos,
las sentencias de asignación copian valores:

```
// Intercambio de los valores de dos variables

int x = 100;
int y = 200;
int tmp;

tmp = y;   // La variable temporal
           // almacena el valor inicial de y

y = x;     // Se almacena en y el valor de x (100)

x = tmp;   // Se almacena en x
           // el valor original de y (200)
```

Cuando usamos objetos (datos de tipos no primitivos),
las sentencias de asignación copian referencias:

```
Cuenta miCuenta = new Cuenta();
Cuenta tmp;

tmp = miCuenta;         // Copia la referencia

tmp.ingresar(150);      // Se hace un ingreso en
                        //   ¡¡¡ miCuenta !!!
```

MUY IMPORTANTE

En una sentencia de asignación,
no se crea una copia del dato representado por un objeto
(salvo que trabajemos con tipos primitivos)