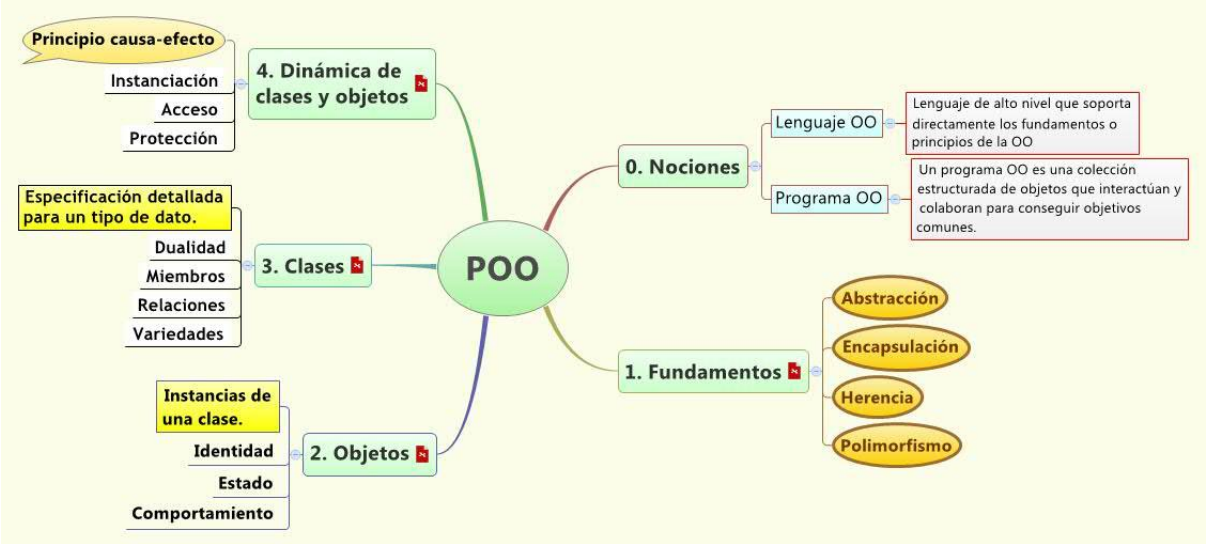
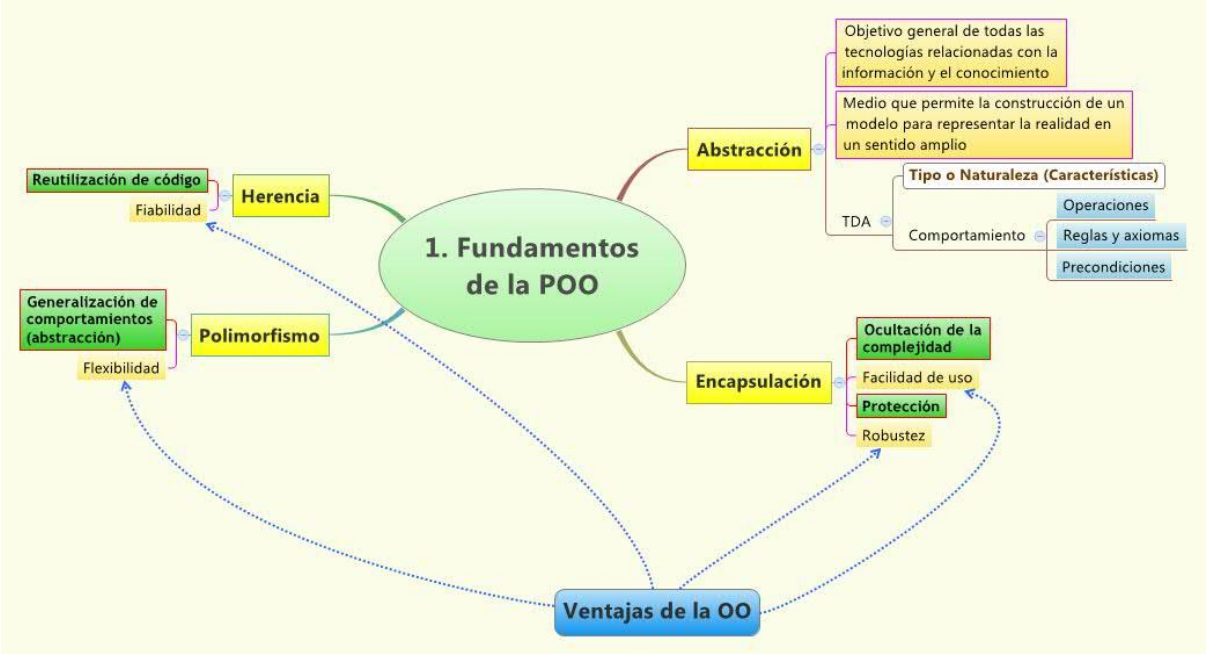
Capítulo 6. Clases, interfaces y enumerados

A. J. Pérez

Clases, interfaces y uniones

Aspectos básicos de la programación orientada a objetos

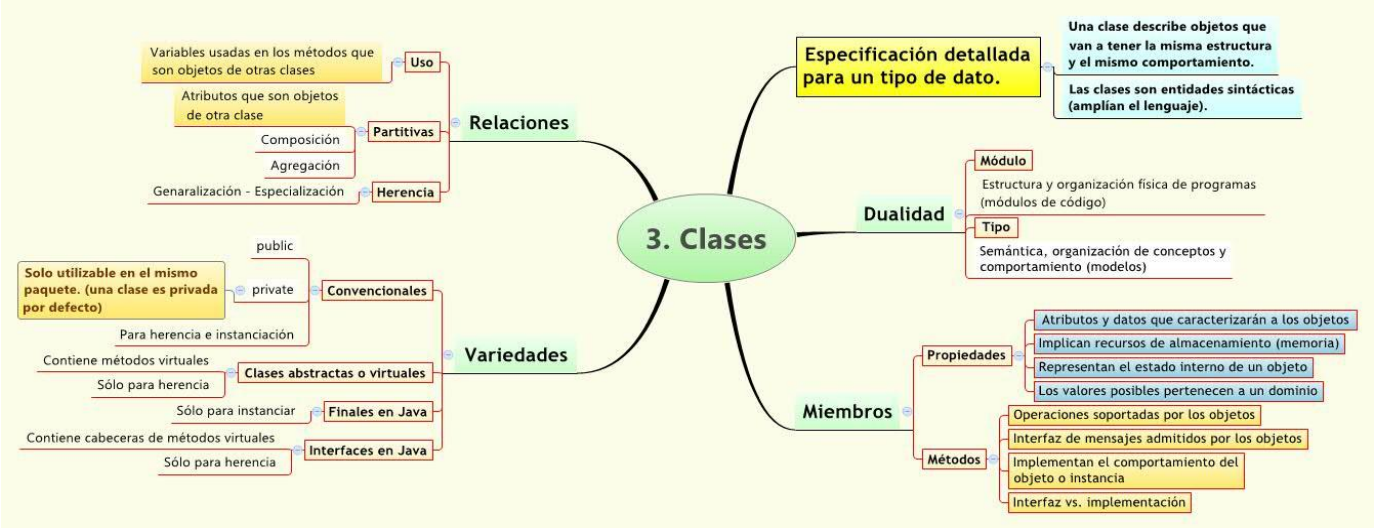




Las clases como tipos de datos del programador

Las clases son un mecanismo que permite la ampliación de un lenguaje con nuevos tipos de datos definidos por el programador. Normalmente los nuevos tipos corresponden a una

descripción de objetos reales, entidades o conceptos conocidos. Son un modelo de la realidad.



Las clases determinan las características que tendrán los objetos con **las propiedades o atributos**; además, establecen el comportamiento que tendrán esos objetos a través de los métodos.

Una clase define la estructura de un objeto y su interfaz funcional implementada con métodos.

Los objetos son ejemplares (instancias) de las clases. Por ejemplo, un objeto `usuario` con un atributo llamado `nombre` que toma el valor `"Pedro"`.

En el aspecto organizativo de un programa, para crear una clase en Java se necesita un archivo fuente que contenga:

- ❏ Una declaración de la clase con la palabra clave **`class`** seguida de un identificador válido.
- ❏ Un bloque de instrucciones delimitado por por llaves `{...}`, entre las que se coloca el contenido de la clase; se le llama el *cuerpo de la clase*.

```
public class Punto {
      //. . .
```

```
}
```

El cuerpo de una clase típica, puede incluir:

- *Atributos de clase.*
  Son las posibles variables que almacenan los valores de las características comunes y compartidas por todos los objetos de la misma clase.
- *Atributos de instancia.*
  Son las variables que almacenan valores definidos de las las propiedades particulares de cado objeto instanciado; representan el estado de cada objeto.
- *Métodos de clase.*
  Son las operaciones que no requieren la existencia de un objeto para que estén disponibles.
- *Métodos de instancia.*
  Son las operaciones que determinan el comportamiento y qué “saben hacer” cada uno de los objeto instanciados de la clase. Son comunes y compartidos por todos los objetos de la misma clase.

Los programas Java se componen normalmente de varias clases, en distintos archivos fuente, organizadas a su vez en paquetes.

La forma general de una clase se muestra a continuación.

```
public class NombreClase extends ClaseBase {

	//Atributos de clase
	static int nombreAtributo1;	//Ejemplos
	static boolean nombreAtributo2;

	//Atributos de instancia
	int nombreAtributo3;
	String nombreAtributoN;

	//Métodos de clase
	//Ejemplos de métodos
	public int nombreMetodo1(int arg1, char arg2, String argN) {
		//Cuerpo del método
		// ...
	}
	 public boolean nombreMetodo2(int arg1, char arg2, String argN) {
		//Cuerpo del método
		// ...
	}
	public String nombreMetodoN(int arg1, char arg2, String argN) {
		//Cuerpo del método
		// ...
	}

} //fin de la clase
```

`NombreClase` y **`ClaseBase`** son *identificadores*. La palabra clave **`extends`** se utiliza para indicar que **`NombreClase`** será una clase derivada o *subclase* de **`ClaseBase`**. Hay una clase predefinida, llamada **`Object`** (objeto), que está en la raíz de la *jerarquía de clases de Java*. Si se desea crear una subclase de **`Object`** directamente, se puede omitir la cláusula **`extends`**.

Uso de clases y objetos

Cada nueva clase que se crea en un programa, añade un *nuevo tipo de dato* que se puede utilizar de la misma manera que los *tipos predefinidos* en el lenguaje.

Para acceder a lo que se especifica en una clase dada, primero hay que crear un objeto de la misma. Esto se consigue con la palabra reservada **`new`** en combinación con algunos de los constructores de la clase; se creará un objeto de una clase dada.

Si se quiere manejar el objeto recién creado, habrá que asignarlo a una variable adecuada a su tipo de clase. Esta variable es una referencia vinculada al objeto previamente creado.
Utilizando la variable, y la notación "punto", se tiene acceso a los métodos y las propiedades públicas del objeto.

Referencias a objeto

Cuando se declara una nueva variable utilizando un nombre de clase como tipo; la variable es una referencias a una instancia de la clase indicada. A una instancia de una clase se le llama también objeto.

Cuando se declara un objeto, tiene como valor por defecto **`null`**. Un objeto con valor **`null`** no es utilizables; debe ser creado antes en la memoria con el operador **`new`**.
Este ejemplo declara una variable p cuyo tipo es de la clase **`Punto`**.

```
Punto p;
System.out.println(p); //muestra null
```

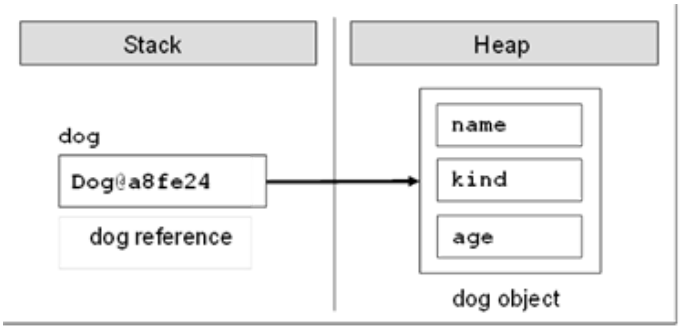
Naturaleza de los objetos

Cuando se crea un objeto, éste está constituido por dos partes:

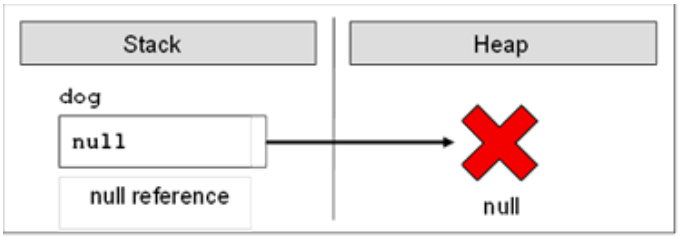
- ❏ La parte significativa (datos), que contiene su estado representado por los valores de los atributos; se almacena en la memoria del programa llamada *memoria dinámica (heap)*
- ❏ Una referencia que se encuentra en la otra parte de la memoria del programa, donde se almacenan las variables y parámetros de los métodos; *la pila de ejecución del programa*).

Suponiendo la clase **Dog** , con las propiedades `name`, `kind` y `age`; y se crea la variable `dog` de esta clase. Esta variable es una referencia al objeto que se encuentra en la memoria dinámica (heap).

Una referencia es una variable, que permite acceder a los objetos. La siguiente figura muestra un ejemplo de referencia, que tiene vínculo con el objeto real en el *heap*, que se llama `dog`. Las referencia, a diferencia de las variables de tipo primitivo, no contienen el valor real (es decir, los datos del objeto); sólo indican la dirección de memoria donde se encuentra:



Cuando se declara una variable de un tipo de dato que es una clase, y no se asocia a un objeto específico se le asigna por defecto valor **`null`**. La palabra reservada **`null`** indica que la variable apunta a ningún objeto:



Atributos de clase

Son los datos que se comparten por todos los objetos de una misma clase; corresponden a los valores que tienen los atributos declarados como estáticos en el bloque, de primer nivel de llaves, de la clase. También se les denomina *propiedades de clase*.

Este ejemplo declara una clase de nombre `Punto`, con un atributo de clase de tipo entero llamado `total`.

```
class Punto {
	static int total;   //Total de puntos instanciados -propiedad de clase-
	int x, y;
}
```

Atributos de instancia o de objeto

Son los datos que se encapsulan dentro de un objeto y corresponden a los valores que tienen los atributos declarados en el bloque, de primer nivel de llaves, de la clase. También se les

denomina *propiedades de instancia o de objeto*.

Este ejemplo declara una clase de nombre **Punto**, con dos atributos de instancia de tipo entero llamadas `x` e `y`.

```
class Punto {
	int x, y;
}
```

Métodos de instancia

Son los métodos que determinan cómo se comporta un objeto. Son comunes y compartidos por todos los objeto de una misma clase.

Este ejemplo declara una clase de nombre **Punto**, con un método de instancia llamado `mostrar()`.

```
class Punto {
	int x;
	int y;

	public void mostrar() {
		//...
	}
}
```

Métodos de clase

Son los métodos que están disponibles sin necesidad de instanciar objetos.

- Se accede a ellos utilizando el nombre de la clase directamente con el operador punto.
- Si hay instancia de objeto, también están disponibles.
- Son métodos declarados como estáticos.

Este ejemplo declara una clase de nombre **Punto**, con un método de clase llamado `borrar()`.

```
class Punto {
	int x, y;

	public static void borrar(Punto p) {
		//...
	}
}
```

El operador `new`

El operador `new` crea una instancia de una clase en memoria y devuelve una referencia a ese objeto.

En el ejemplo se crea una nueva instancia de **`Punto`** y se almacena en una variable `p`.

```
Punto p = new Punto();
Punto p2 = p;
p = null;
```

Aquí `p` referencia a una instancia de **`Punto`**, pero realmente no contiene al objeto, sólo lo direcciona. También se ha creado otra referencia `p2` al mismo objeto.

Cualquier cambio realizado en el objeto a través de `p2` afectará al mismo objeto al que también se refiere `p`. La asignación de `p` a `p2` no duplica el objeto; duplica la referencia creando un segundo alias para el objeto. De hecho las asignaciones posteriores a `p` simplemente desvinculan `p` del objeto original sin afectar al propio objeto. Aunque se haya asignado `p` a **`null`**, el objeto todavía está accesible a través de `p2`; si no, sería eliminado automáticamente por el *recolector de basura*.

El operador punto (.)

El operador punto se utiliza para acceder a las variables o atributos de instancia y los métodos contenidos en un objeto.

Esta es la forma general de acceder a los atributos de instancia utilizando el operador punto.

<referencia>.<atributo>

Aquí ***`referencia`*** es una referencia a un objeto y ***`atributo`*** es el nombre del atributo de instancia contenida en el objeto al que se desea acceder. El siguiente fragmento de código muestra cómo se puede utilizar el operador punto para almacenar valores en atributos o variables de instancia y para referirnos a estos.

```
Punto p = new Punto();
p.x = 10;
p.y = 20;
System.out.println("x = " + p.x + " y = " + p.y);
```

Declaración de métodos

Los métodos son subprogramas pertenecientes a una clase específica. Son miembros de una clase que se declaran al mismo nivel que los atributos o variables, dentro de la clase.

En la declaración de los métodos se especifica si devuelven un valor de un tipo concreto y si utilizan una lista de parámetros de entrada.

```
<tipoRetorno> nombreMetodo(<listaFormalArgumentos>) {
      <cuerpoMetodo>;
}
```

- El ***tipoRetorno*** es el tipo de dato que devolverá el método con la sentencia **`return`** .
- El ***nombreMetodo*** es cualquier identificador válido distinto de los ya utilizados en el ámbito actual.
- La ***listaFormalArgumentos*** es una secuencia de parejas de tipo e identificador separadas por comas. Si no se desean parámetros, la declaración del método deberá incluir un par de paréntesis vacío.

Se puede crear un método en la clase **`Punto`** que asigne valor inicial a los atributos de instancia:

```
class Punto {
      int x, y;

      void valorInicial(int x, int y) {
            this.x = x;
            this.y = y;
      }
}
```

`this`

En Java es ilegal declarar dos variables locales con el mismo nombre dentro del mismo ámbito o uno que lo incluya. Obsérvese que se ha utilizado `x` e `y` como parámetros para el método `valorInicial()` y en el interior del método se ha utilizado un valor de referencia especial llamado **`this`** para referirse directamente a las variables de instancia. Si no se hubiese utilizado **`this`** entonces `x` e `y` se hubieran referido al parámetro formal y no a los atributos de instancia lo que se conoce como *ocultación de las variables de instancia*.

El método `main()`

La manera de iniciar un programa Java es el método `main()`. Puesto que en otros lenguajes (p.e.

C y C++) `main()` se utiliza a menudo para pasar parámetros desde la línea de comandos, un concepto perdido en los interfaces gráficos de usuario, el `main()` de Java también admite esos argumentos.
El método `main()` es simplemente un punto de inicio lógico del programa. Un programa complejo podrá tener decenas de clases, y sólo una de ellas necesitará tener un método `main()`.
Para los *applets* (programas Java que se ejecutan en el contexto de los navegadores de Internet) no se utiliza el método `main()`, ya que los navegadores siguen un convenio distinto para la ejecución de los *applets*.

Llamada a un método

Se llama a los métodos dentro de un otro método de una clase utilizando el operador punto (`.`).

La forma general de una llamada:

<referenciaObjeto>.<nombreMetodo>(<listaParametros>);

- ***referenciaObjeto*** es cualquier variable que se refiere a un objeto.
- ***nombreMetodo*** es el nombre de un método de la clase con la que se declaró ***referenciaObjeto***.
- ***listaParametros*** es una lista de valores o expresiones separados por comas que coinciden en número, tipo y orden con un método declarado como ***nombreMetodo*** en la clase. Son lo que se llaman parámetros o argumentos actuales.

En el ejemplo, se llama al método `valorInicial()` sobre cualquier objeto **`Punto`**.

```
Punto p = new Punto();
p.valorInicial(10, 20);
```

Dentro de una misma clase, cuando se hace una llamada a un método perteneciente a sí misma, se puede utilizar directamente el nombre del método sin necesidad del operador punto (`.`).

Métodos de acceso

Las clases pueden implementar métodos especializados para la manipulación lectura/escritura del estado interno de los objetos -valores de los atributos-. **Cada atributo suele tener asociado un par de métodos; uno de lectura y otro de modificación.**

Constructores

Las clases pueden implementar métodos especiales llamados *constructores*. **Un constructor es un**

método que inicializa los atributos de un objeto inmediatamente después de su creación.

Los constructores deben cumplir los siguientes reglas:

- Deben tener exactamente el mismo nombre de la clase a la que pertenecen.
- No pueden tener indicación de valor de retorno, ni siquiera **`void`**.
- Pueden estar sobrecargados con diferentes listas de argumentos.
- No puede ser invocado fuera del contexto de la creación de un objeto.

Una vez definido un constructor, se puede utilizar junto con el operador **`new`**.

La clase **`Punto`**, con un constructor sería:

```
public class Punto {
	int x;
	int y;

	public Punto(int x, int y) {
		this.x = x;
		this.y = y;
	}
}
```

```
public class PruebaConstructor {

	public static void main(String args[]) {
		Punto p = new Punto(10, 20);
		System.out.println("x = " + p.x + " y = " + p.y);
	}
}
```

La llamada y ejecución del *método constructor* se produce justo después de crear la instancia del nuevo objeto en memoria y antes de que se complete la asignación a la referencia `p`.

Sobrecarga de métodos

Es posible y a menudo deseable crear más de un **método con el mismo nombre, pero con listas de parámetros distintas.** A esto se le llama *sobrecarga de métodos*. Se sobrecarga un método cuando en una clase se añade otro con el mismo nombre.

En el ejemplo se presenta una versión de la clase **`Punto`** que utiliza sobrecarga de métodos para crear un constructor alternativo.

```
public class Punto {
```

```
	int x, y;

	public Punto(int x, int y) {
		this.x = x;
		this.y = y;
	}

	public Punto() {
		this.x = 0;
		this.y = 0;
	}
}
```

```
public class PruebaConstructor {

	public static void main(String args[]) {
		Punto p = new Punto();
		System.out.println("x = " + p.x + " y = " + p.y);
	}
}
```

Crea un objeto **Punto** que llama al segundo constructor sin parámetros en vez de al primero.

Llamada explícita a constructor

Un refinamiento y **optimización adicional que se suele hacer en los constructores es la llamada entre sí para no tener código repetido.** A menudo es una buena idea crear constructores relacionados y que las clases resulten fáciles de mantener y evolucionar.

```
public class Punto {
	int x, y;

	public Punto(int x, int y) {
		this.x = x;
		this.y = y;
	}

	public Punto() {
		this(0, 0);

	}
}
```

Modificadores de atributos y métodos

`final`

En Java todos los métodos y los atributos, por defecto, se pueden sobrescribir (reescribir o redefinir). **Si se desea indicar que ya no se quiere permitir que las clases derivadas puedan redefinir los atributos y métodos**, éstos se pueden declarar como **`final`**. El modificador de tipo **`final`** implica que todas las referencias futuras a este elemento se basarán en esta definición.

Se puede utilizar **`final`** como modificador en declaraciones de método cuando se desea impedir que las clases derivadas puedan redefinir un método concreto. Se volverá a tratar este aspecto cuando se estudie el tema de la herencia, en el Capítulo 11. Programación orientada a objetos

A veces se desea crear un método que pueda utilizarse sin necesidad de instanciar objetos, directamente asociado con el nombre de la clase. Para conseguir esto se hay que declarar estos métodos como **`static`**.

Los métodos estáticos sólo pueden llamar a otros métodos **`static`** directamente, y no se pueden hacer referencia a **`this`** o **`super`** de ninguna manera porque no estaría asegurada la existencia del correspondiente objeto.

Los atributos también se pueden declarar como **`static`**, lo que en términos de POO serían atributos de clase compartidos por todos los objetos o instancias. Es por otro lado, lo que se puede hacer en Java para tener lo más parecido a la noción de variables globales, accesibles desde cualquier parte del código.

Se puede declarar un bloque **`static`**, que se ejecuta una sola vez, si se necesitan realizar cálculos para inicializar las variables **`static`**. El ejemplo siguiente muestra una clase que tiene un método **`static`**, algunos atributos de clase **`static`** y un bloque de inicialización **`static`**.

```
public class Estatic {

	static int a = 3;
	static int b;

	static void metodo(int x) {
		System.out.println("x = " + x);
		System.out.println("a = " + a);
		System.out.println("b = " + b);
	}
```

```
	//bloque de instrucciones estáticas
	static {
		System.out.println("bloque static inicializado");
		b = a * 4;
	}
}
```

Salida:

```
bloque static inicializado
x = 42
a = 3
b = 12
```

Paquetes

Los paquetes son los módulos de Java. Recipientes que contienen clases. Se utilizan tanto para mantener el espacio de nombres dividido en compartimentos más manejables, como un mecanismo de encapsulación y restricción de visibilidad. Se pueden poner clases dentro de los paquetes restringiendo la accesibilidad a éstas en función de la localización de los otros miembros del paquete.

Cada archivo `.java` tiene cuatro secciones internas; hasta ahora sólo se ha utilizado una de ellas en los ejemplos. Esta es la forma general de un archivo fuente de Java:

- ❏ Una única sentencia de paquete (opcional)
- ❏ Las sentencias de importación deseadas (opcional)
- ❏ Una única declaración de clase pública.
- ❏ Las clases privadas de paquete deseadas (opcional)

La sentencia package

Lo primero que se permite en un archivo Java es una sentencia **`package`**, que le dice al compilador en qué paquete se deberían definir las clases incluidas. Si se omite la sentencia **`package`**, las clases se ubican en el *paquete por defecto*, que no tiene nombre. **El compilador Java utiliza directorios o carpetas del sistema de archivos anfitrión para almacenar los paquetes.**

Si se declara que una clase está incluida en un paquete llamado *`MiPaquete`*, entonces el archivo fuente de esa clase se debe almacenar en un directorio o carpeta llamado *`MiPaquete`*.
Hay que destacar que Java es sensible a las mayúsculas y que el nombre del directorio o carpeta debe coincidir con el nombre exactamente. Esta es la forma general de la sentencia package.

`package <paq1>[ .<paq2> [ .<paq3> ]];`

Se puede crear una jerarquía de paquetes dentro de paquetes separando los niveles por puntos. Esta jerarquía se debe reflejar en el sistema de archivos de desarrollo de Java.

Un paquete declarado como:
`package java.awt.imagen;`

Se almacena, según el sistema anfitrión, en:
- `java/awt/imagen` (Wndows)
- `java\awt\imagen` (UNIX)
- `java:awt:imagen` (Macintosh)

Compilación de clases en paquetes

Cuando se intenta poner una clase en un paquete, el primer problema es que la estructura de directorios debe coincidir exactamente con la jerarquía de paquetes. No se puede renombrar un paquete sin renombrar el directorio en el cual están almacenadas las clases. El otro problema que presenta la compilación es más sutil. Cuando se quiere llamar a una clase que se encuentre en un paquete se debe prestar atención a su ubicación en la jerarquía de paquetes.

Por ejemplo, se crea una clase llamada **Prueba** en un paquete llamado *`paquetePrueba`*. Para ello se crea el directorio llamado *`paquetePrueba`*, se pone en él `Prueba.java` y se compila. Si se intenta ejecutar; el intérprete no lo encontrará. Esto se debe a que esta clase compilada se encuentra ahora en un paquete llamado *`paquetePrueba`* y no se puede referir simplemente escribiendo, en línea de comandos: `java Prueba` Sin embargo si es posible referirse a cualquier paquete enumerando su jerarquía de paquetes, separando los paquetes por puntos. Por lo tanto, si se intenta ejecutar ahora, escribiendo en línea de comandos: `Java paquetePrueba.Prueba`, el intérprete tampoco lo encontrará.

El motivo de este nuevo fallo se encuentra en la variable ***`CLASSPATH`***. Probablemente contenga algo parecido a `".;C:\java\class"` que indica al intérprete que busque en el directorio de trabajo actual y en el directorio de instalación del JDK. El problema es que no hay un directorio prueba en el directorio de trabajo actual porque ya se está en el directorio prueba.

Hay dos opciones:
- Cambiar de directorio o carpeta un nivel arriba y ejecutar `Java paquetePrueba.Prueba`
- Añadir la ruta de `paquetePrueba` a la variable de entorno ***`CLASSPATH`***. Después se podrá utilizar `Java paquetePrueba.Prueba` desde cualquier directorio y Java encontrará el archivo `.class` adecuado. Si se trabaja con código fuente en un directorio llamado `C:\mijava`; ***`CLASSPATH`*** sería `.;C:\mijava;C:\java\class`

La sentencia `import`

Lo siguiente que suele aparecer en un fichero fuente después de una sentencia **`package`** *y antes de las definiciones de la clase, puede ser una lista de sentencias* **`import`**. Las clases suelen están

almacenadas en algún paquete con nombre. Para no tener que introducir el nombre completo de ruta de paquete para cada clase, Java proporciona la sentencia **`import`** para que se puedan ver ciertas clases o paquetes enteros. La forma general de la sentencia import es:

```
import <paquete1>.[ <paquete2> ].( <nombreClase> | * );
```

- ***`paquete1`*** es el nombre de un paquete de alto nivel, ***`paquete2`*** es el nombre de un paquete opcional contenido en el paquete exterior separado por un punto (**`.`**). No hay ningún límite teórico a la profundidad de la jerarquía de paquetes.
- Finalmente, ***`nombreClase`*** explícito o un asterisco (***`*`***) que indica que el compilador Java debería buscar en todo el paquete.

Ejemplos:

```
import java.util.Date;
import java.io.*;
```

Protección de accesos

Java proporciona varios mecanismos para permitir un control de acceso entre clases en circunstancias diferentes.

- Dentro de una misma clase todas los atributos y métodos son visibles para todas las otras partes.
- Por la existencia de paquetes, Java debe distinguir cuatro categorías de visibilidad entre elementos de clase:
  - Clases derivadas dentro del mismo paquete.
  - Otras clases dentro del mismo paquete.
  - Clases derivadas en paquetes distintos.
  - Otras clases en paquetes distintos.

Las tres palabras clave:

- **`private`**
- **`public`**
- **`protected`**

Se combinan de varias maneras para generar los distintos niveles de acceso. La tabla siguiente resume las posibilidades.

	private	defecto	private protected	protected	public
Misma clase	sí	sí	sí	sí	sí
Clase derivada, mismo paquete	no	sí	sí	sí	sí
Otras clases, mismo paquete	no	sí	no	sí	sí
Clase derivada, diferente paquete	no	no	sí	sí	sí
Otra clase, diferente paquete	no	no	no	no	sí

Interfaces Java

Las interfaces son como las clases, pero sin variables de instancia y con métodos declarados sin cuerpo. Las clases pueden implementar varias interfaces. Para implementar una interfaz, todo lo que se necesita una clase es una implementación del conjunto completo de métodos de la interfaz. Las interfaces están en una jerarquía distinta de las clases, por lo que es posible que varias clases que no tengan relación, en cuanto a la jerarquía de clases, implementen la misma interfaz.

La forma general de una interfaz es:

```
public interface NombreInterface {

	final tipoDato NOMBRE_ATRIBUTO_1 = valor1;
	final tipoDato NOMBRE_ATRIBUTO_2 = valor2;
	//...
	final tipoDato NOMBRE_ATRIBUTO_N = valorN;

	tipoRetorno nombreMetodo1(listaParametros);
	tipoRetorno nombreMetodo2(listaParametros);
	//...
	tipoRetorno nombreMetodoN(listaParametros);
}
```

Aquí `NombreInterface` es cualquier identificador válido según estilo estándar. Observese que los métodos que se declaran no tienen cuerpo de sentencias. Todos los métodos que están incluidos en interfaces son **`public`** aunque no se indique expresamente. Los atributos declarados dentro de las interfaces deben ser **`final`** , lo que significa que no las puede cambiar la clase que las implementa y además deben ser inicializadas con un valor constante. Un ejemplo de declaración de interfaz que contiene un método que toma un único parámetro.

```
public interface Callback {
	void callback(int param);
}
interface {

}
```

La sentencia `implements`

la palabra clave **`implements`** permite especificar qué interfaces se hace responsable de

completar y cumplir una clase cuando se implementa. Ampliando la definición anterior de clase, una clase que implementa algunas interfaces tiene la forma general siguiente:

```
public class NombreClase extends ClaseBase impplements Interface1,
Interface2, … InterfaceN {

	//Atributos
	int nombreAtributo1;	//Ejemplos
	String nombreAtributo2;
	boolean nombreAtributoN;

	//Métodos
	//Ejemplos de métodos
	public int nombreMetodo1(int arg1, char arg2, String argN) {
		//Cuerpo del método
		// …
	}
	public boolean nombreMetodo2(int arg1, char arg2, String argN) {
		//Cuerpo del método
		// …
	}
	public String nombreMetodoN(int arg1, char arg2, String argN) {
		//Cuerpo del método
		// …
	}

} //fin de la clase
```

Enumerados Java

Los tipos enumerados de Java (enum) se incorporaron a Java con el JDK 1.5. El uso típico más evidente es el de actuar como contenedor de constantes.

```
public enum  RolUsuario {
       NORMAL,
       ADMIN,
       INVITADO;
}
```

Una forma más avanzada, asignando parámetros a cada valor enumerado, podría ser:

```
public enum  RolUsuario {

       NORMAL("Normal"),
       ADMIN("Admin"),
       INVITADO("Invitado");

      private final String valor;

      // Constructor privado.
      RolUsuario(String valor) {
        this.valor = valor;

    public String getValor() {
        return valor;
    }
  }
```

En el ejemplo se asigna un nombre en mayúsculas que se muestra por defecto y un valor interno que es asignado por el constructor privado.

Ejercicios

1. (*) Define una clase **`Estudiante`**, que contiene la siguiente información para los estudiantes: nombre y apellidos, curso, grado, universidad, correo electrónico y número de teléfono.

2. (*) Añade un atributo estático en la clase **`Estudiante`**, para almacenar el número de total de objetos creados de esa clase.

3. (*) Añade un método en la clase **`Estudiante`**, para mostrar por consola la información completa sobre el estudiante.

4. (**) Modifica el código actual a la clase del **`Estudiante`** con el fin de encapsular los datos de la clase a través de propiedades privadas creando los necesarios métodos de acceso.

5. (**) Añade varios constructores de la clase **`Estudiante`**, con diferentes listas de parámetros. Los datos para los cuales no haya entrada serán inicializados con **`null`** ó `0`.

6. (**) Escribe la clase **`EstudianteTest`**, para probar la funcionalidad de la clase Estudiante. Prepara los casos de prueba en el método `main()` en **`EstudianteTest`**.

7. (**) Agrega un método estático en una clase **`EstudianteTest`**, que crea varios objetos de tipo **`Estudiante`** y los almacena en atributos estáticos. Escribe un método de prueba que muestra en la consola la información de los estudiantes.

8. (**) Define las siguientes clases:
    a. **`TelefonoMovil`**: modelo, fabricante, precio, dueño, bateria, pantalla.
    b. **`Bateria`**: modelo, tiempo de inactividad, horasCapacidad.
    c. **`Pantalla`**: tamaño, fabricante, modelo.

9. (**) Declara varios constructores para cada una de las clases creadas en el ejercicio anterior, con diferentes listas de parámetros (indicando información general acerca de un estudiante o parte de ella). Los atributos para los que no se reciban argumentos se inicializan con **`null`** ó `0`.

10. (**) En la clase **`TelefonoMovil`** de los ejercicios anteriores, agrega un atributo estático llamado `nokia95` , que almacena información sobre el modelo de teléfono móvil Nokia 95 . Agrega un método, que muestra la información del atributo estático.

11. (**) Añade un tipo enumerado **`TipoBateria`**, para contener los diferentes tipos de la batería (`IonLi, NiMH, NiCd`, ...). Utiliza el enumerado como un nuevo campo para la clase de **`Bateria`**.

12. (**) Redefine el método `toString()` de la clase **`TelefonoMovil`**.

13. (**) En la clase **`TelefonoMovil`**, define atributos privados para encapsular objetos de las clases **`Bateria`** y **`Pantalla`**.

14. (**) Escribe la clase **`TelefonoMovilTest`**, que pone a prueba la funcionalidad de la clase **`TelefonoMovil`**.
    - ➢ Crea algunos objetos de la clase.
    - ➢ Guardarlos en una lista dinámica.
    - ➢ Muestra información sobre los objetos creados.
    - ➢ Muestra información sobre el campo estático `nokian95`.

15. (**) La clase **`Llamada`**, que contiene información acerca de una conversación realizada a través de **`TelefonoMovil`**. Debe contener información sobre la fecha, hora de inicio y la duración de la llamada.

16. (**) En la clase **`Llamada`**, añade un atributo estático que sea una lista que actúa de registro histórico de llamadas para hacer un seguimiento.

17. (**) En la clase **`TelefonoMovil`** añade métodos para agregar y eliminar llamadas en el registro histórico. Añade un método que elimina todas las llamadas del registro histórico.

18. (**) En la clase **`TelefonoMovil`** agrega un método que calcula el coste total de las llamadas registradas en el registro histórico. Recibe el precio por minuto de una llamada.

19. (**) Crea una clase **`TelefonoMovilHistoricoTest`**, con el que poner a prueba la funcionalidad de la clase **`TelefonoMovil`**, creando un objeto, agregando un par de llamadas y mostrando la información para cada llamada. Suponiendo que el precio por minuto es de 0,37, calcular e imprimir el precio total de las llamadas. Elimina la conversación más larga de las llamadas de la lista y calcular de nuevo el costo total de todas las conversaciones. Por último, elimina las llamadas del registro histórico.

20. (**) Define las clases **`Biblioteca`** y **`Libro`**. La biblioteca debe contener un nombre y una lista de libros. Los libros deben incluir el título, autor, editorial, año de publicación y número de ISBN. En la clase que **`Biblioteca`**, agrega métodos para dar de alta un libro en la biblioteca, buscar un libro por autor, mostrar información para un libro y eliminar un libro de la biblioteca.

21. (**) Escribe una clase de prueba **`BibliotecaTest`** que crea un objeto de `biblioteca`, añade un par de libros a la misma y muestra información sobre cada uno de ellos. Implementa la funcionalidad de prueba que busca todos los libros escritos por *Stephen King* y los elimina. Vuelve a mostrar la información para cada uno de los libros restantes.

22. (**) En una **`Escuela`** hay aulas y estudiantes. Cada aula tiene un número de maestros.

Cada maestro tiene una serie de disciplinas en las que enseñan. Los estudiantes tienen un único nombre y número de la clase. Las clases tienen un identificador de texto único. Los alumnos tienen un nombre, número de clases y han realizado un número de ejercicios. Modela la escuela con clases de Java. Se deben declarar las clases junto con sus atributos, métodos y constructores, etc. Define una clase de prueba que demuestre que las otras clases funcionan correctamente.

23. (**) Define una clase **`Fraccion`** , que contiene información acerca de una fracción racional (por ejemplo, ¼ o ½). Define un método estático `parse()` para crear una fracción desde un texto, por ejemplo,"-3 / 4"). Define las propiedades y constructores apropiados de la clase. Escribe un método para devolver el valor decimal de la fracción (por ejemplo 0.25).

24. (**) Escriba una clase **`FracionTest`**, que prueba la funcionalidad de la clase **`Fraccion`** del ejercicio anterior. Pruebas especialmente el correcto funcionamiento de `parse()` con diferentes datos de entrada.

25. (**) Escribe un método para simplificar una fracción (por ejemplo, si el numerador y el denominador son, respectivamente, 10 y 15, la fracción simplificada debe ser 2/3).

Fuentes y bibliografía

- CARRERES, J. *Manual de Java*. [en línea] http://www.oocities.org/collegepark/quad/8901/indice.html
- NAKOV, S. *Fundamentos de programación con Java*. Ed. Faber, Veliko Tarnovo, 2009. [en línea][traducción automática del búlgaro] http://www.introprogramming.info/intro-java-book/