POO

Convenciones y modificadores

Clases

- Como nombre utilizaremos un sustantivo
- Puede estar formado por varias palabras
- Cada palabra comenzará con mayúscula, el resto se dejará en minúscula
  - Por ejemplo: DataInputStream
- Si la clase contiene un conjunto de métodos estáticos o constantes relacionadas pondremos el nombre en plural
  - Por ejemplo: Resources

Campos y variables

- *Campos y variables*: simples o complejos
- Utilizaremos sustantivos como nombres

```
Properties propiedades;  File
ficheroEntrada;
int numVidas;
```

- Puede estar formado por varias palabras, con la primera en minúsculas y el resto comenzando por mayúsculas y el resto en minúsculas
  - Por ejemplo: numVidas
- En caso de tratarse de una colección de elementos, utilizaremos plural
  - Por ejemplo: clientes
- Para variables temporales podemos utilizar nombres cortos, como las iniciales de la clase a la que pertenezca, o un carácter correspondiente al tipo de dato

```
int i;
Vector v;
DataInputStream dis;
```

Constantes

- *Constantes*: Se declarán como *final* y *static*

```
final   static   String TITULO_MENU = “Menu”;
final   static   int ANCHO_VENTANA = 640;
final   static   double PI = 3.1416;
```

- El nombre puede contener varias palabras
- Las palabras se separan con ‘_’
- Todo el nombre estará en mayúsculas
  - Por ejemplo: MAX_MENSAJES

Métodos

- *Métodos*: con el tipo devuelto, nombre y parámetros

```
void imprimir(String mensaje)
{
    ...// Codigo del método
}
Vector insertarVector(Object elemento, int posicion)
{
    ...// Codigo del método
}
```

- Los nombres de los métodos serán verbo
- Puede estar formados por varias palabras, con la primera en minúsculas y el resto comenzando por mayúsculas y el resto en minúsculas
  - Por ejemplo: imprimirDatos

Constructores

- *Constructores*: se llaman igual que la clase, y se ejecutan con el operador *new* para reservar memoria

```
MiClase()
{
    …// Codigo del constructor
}
MiClase(int valorA, Vector valorV)
{
    …// Codigo del otro constructor
}
```

- No hace falta destructor, de eso se encarga el *garbage collector*
- Constructor superclase: `super(…)`

Paquetes

- *Paquetes*: organizan las clases en una jerarquía de paquetes y subpaquetes
- Para indicar que una clase pertenece a un paquete o subpaquete se utiliza la palabra *package* al principio de la clase

```
package paquete1.subpaquete1;
class MiClase {
```

- Para utilizar clases de un paquete en otro, se colocan al principio sentencias *import* con los paquetes necesarios:

```
package otropaquete;
import paquete1.subpaquete1.MiClase;
import java.util.*;
class MiOtraClase {
```

Paquetes

- Si no utilizamos sentencias *import*, deberemos escribir el nombre completo de cada clase del paquete no importado (incluyendo subpaquetes)

```
class MiOtraClase {
    paquete1.subpaquete1.MiClase a = ...;      // Sin import
    MiClase a = ...;                           // Con import
```

- Los paquetes se estructuran en directorios en el disco duro, siguiendo la misma jerarquía de paquetes y subpaquetes

```
./paquete1/subpaquete1/MiClase.java
```

Paquetes

- Siempre se deben incluir las clases creadas en un paquete
  - Si no se especifica un nombre de paquete la clase pertenecerá a un paquete “sin nombre”
  - No podemos importar clases de paquetes “sin nombre”, las clases creadas de esta forma no serán accesibles desde otros paquetes
  - Sólo utilizaremos paquetes “sin nombre” para hacer una prueba rápida, nunca en otro caso

Convenciones de paquetes

- El nombre de un paquete deberá constar de una serie de palabras simples siempre en minúsculas
  - Se recomienda usar el nombre de nuestra DNS al revés

    especialistajee.org → org.especialistajee.prueba
- Colocar las clases interdependientes, o que suelan usarse juntas, en un mismo paquete
- Separar clases volátiles y estables en paquetes diferentes
- Hacer que un paquete sólo dependa de paquetes más estables que él
- Si creamos una nueva versión de un paquete, daremos el mismo nombre a la nueva versión sólo si es compatible con la anterior

Otras características

- Imports estáticos

```
import static java.lang.Math;
…
double raiz = sqrt(1252.2);
```

- Argumentos variables

```
public void miFunc(String param, int... args) {  for(int i: args) { … }
}
```

- Anotaciones (metainformación)
  - P.ej., @deprecated

Modificadores de acceso

- Las clases y sus elementos admiten unos modificadores de acceso:
  - *privado*: el elemento es accesible sólo desde la clase en que se encuentra
  - *protegido*: el elemento es accesible desde la propia clase, desde sus subclases, y desde clases del mismo paquete
  - *público*: el elemento es accesible desde cualquier clase
  - *paquete*: el elemento es accesible desde la propia clase, o desde clases del mismo paquete.

Modificadores de acceso

- *private* se utiliza para elementos PRIVADOS
- *protected* se utiliza para elementos PROTEGIDOS
- *public* se utiliza para elementos PUBLICOS
- No se especifica nada para elementos PAQUETE

```
public class MiClase {
    private int n;
    protected void metodo() { ... }
```

- Todo fichero Java debe tener una y solo una clase pública, llamada igual que el fichero (más otras clases internas que pueda tener)

Modificadores de acceso

Visibilidad	Public	Protected	Default	Private
Desde la misma Clase	SI	SI	SI	SI
Desde cualquier Clase del mismo Paquete	SI	SI	SI	NO
Desde una SubClase del mismo Paquete	SI	SI	SI	NO
Desde una SubClase fuera del mismo Paquete	SI	SI, a través de la herencia	NO	NO
Desde cualquier Clase fuera del Paquete	SI	NO	NO	NO

Otros modificadores

- *abstract*: para definir clases y métodos abstractos
- *static*: para definir elementos compartidos por todos los objetos que se creen de la misma clase
  - miembros que no pertenecen al objeto en si, sino a la clase
  - dentro de un método estático sólo podemos utilizar elementos estáticos, o elementos que hayamos creado dentro del propio método
- *final*: para definir elementos no modificables ni heredables

```
public abstract class MiClase {
    public static final int n = 20;
    public abstract void metodo();
    ...
```

Otros modificadores

- *volatile* y *synchronized*: para elementos a los que no se puede acceder al mismo tiempo desde distintos hilos de ejecución
  - *volatile* no proporciona atomicidad pero es más eficiente

```
volatile int contador;
contador++; //puede causar problemas, son 3 operaciones
    diferentes
```

  - *synchronized* se usa sobre bloques de código y métodos

```
synchronized(this){
      contador++;
}
```

Otros modificadores

- *native*: para métodos que están escritos en otro lenguaje, por ejemplo en C++, utilizando JNI (Java Native Interface)
- *transient*: para atributos que no forman parte de la persistencia de objeto, para evitar que se serialicen
- *strictfp*: evitar que se utilice toda la precisión de punto flotante que proporcione la arquitectura. Usar el estándar del IEEE para float y double. No es aconsejable a menos que sea necesario.