Tema 3:

Programación orientada a objetos en Java

***Objetivos del tema**: este tema detalla la implementación en Java de la programación orientada a objetos estudiada en el tema anterior.*

INDICE:

1 Clases y herencia en Java 3

1.1 Definición de una clase 3
Sobrecarga de métodos 4
Constructores 5

1.2 Modificadores de métodos y variables 6
Modificadores de variables 6
Modificadores de un método 8
Volviendo al ejemplo del vector 9

1.3 Herencia 11

1.4 Creación y referencia a objetos 16

1.5 this 17

1.6 super 18

1.7 Un ejemplo de herencia con la clase Vector 19

2 Interfaces 20

3 Nuestro primer Programa orientado a objetos 24

4 Aprendiendo a usar los paquetes 31

4.1 Un ejemplo de código con paquetes 34

4.2 El ejemplo de los marcianos con paquetes 38

5 Excepciones 40

5.1 Ejemplo de excepciones: busca de raíces de una función 43

6 Ejercicios 47

1 Clases y herencia en Java

Una clase es la "plantilla" que usamos para crear los objetos. Todos los objetos pertenecen a una determinada clase. Un objeto que se crea a partir de una clase se dice que es una instancia de esa clase. Las distintas clases tienen distintas relaciones de herencia entre si: una clase puede derivarse de otra, en ese caso la clase derivada o clase hija hereda los métodos y variables de la clase de la que se deriva o clase padre. En Java todas las clases tienen como primer padre una misma clase: la clase Object.

Vamos a continuación a profundizar en todos estos conceptos y a explicar su sintaxis en Java.

1.1 Definición de una clase

La forma más general de definición de una clase en Java es:

```
[Modificador] class nombreClase [extends nombreClasePadre]
[implements interface] {
   Declaración de variables;
   Declaración de métodos;
}
```

Los campos que van entre corchetes son optativos. nombreClase es el nombre que le queramos dar a nuestra clase, nombreClasePadre es el nombre de la clase padre, de la cual hereda los métodos y variables. En cuanto al contenido del último corchete ya se explicará más adelante su significado.

Los modificadores indican las posibles propiedades de la clase. Veamos que opciones tenemos:

- **public**: La clase es pública y por lo tanto accesible para todo el mundo. Sólo podemos tener una clase public por unidad de compilación, aunque es posible no tener ninguna.
- **Ninguno**: La clase es "amistosa". Será accesible para las demás clases del package. Sin embargo, mientras todas las clases con las que estemos trabajando estén en el mismo directorio pertenecerán al mismo package y por ello serán como si fuesen públicas. Como por lo de ahora trabajaremos en un solo directorio asumiremos que la ausencia de modificador es equivalente a que la clase sea pública.
- **final**: Indicará que esta clase no puede "tener hijo", no se puede derivar ninguna clase de ella.
- **abstract**: Se trata de una clase de la cual no se puede instanciar ningún objeto.

Veamos un ejemplo de clase en Java:

```
class Animal{
    int edad;
    String nombre;

    public void saluda(){
        System.out.println("Hola");
    }
    public   void mostrarNombre(){
        System.out.println(nombre);
    }
   public void mostrarEdad(){
        System.out.println(edad);
   }
}
```

Sobrecarga de métodos

Java admite lo que se llama sobrecarga de métodos: puede haber varios métodos con el mismo nombre pero a los cuales se les pasan distintos parámetros. Según los parámetros que se le pasen, se invocará a uno u otro método:

```
class Animal{
    int edad;
    String nombre;

    public void saluda(){
        System.out.println("Hola");
    }
    public   void mostrarNombre(){
        System.out.println(nombre);
    }
    public void mostrarNombre(int veces){
        for(int i=0; i <veces; i++){
            System.out.println(nombre);
```

```
        }
    }
   public void mostrarEdad(){
        System.out.println(edad);
   }
}
```

Constructores

Constructores son métodos cuyo nombre coincide con el nombre de la clase y que nunca devuelven ningún tipo de dato, no siendo necesario indicar que el tipo de dato devuelto es void. Los constructores se emplean para inicializar los valores de los objetos y realizar las operaciones que sean necesarias para la generación de este objeto (crear otros objetos que puedan estar contenidos dentro de este objeto, abrir un archivo o una conexión de internet...).

Como cualquier método, un constructor admite sobrecarga. Cuando creamos un objeto (ya se verá más adelante como se hace) podemos invocar al constructor que más nos convenga.

```
class Animal{

    int edad;
    String nombre;

   public Animal(){
        this.edad = 0;
        this.nombre = "Rufo";
    }
    public Animal(int _edad, String _nombre){
    edad = _edad;
    nombre = _nombre;
    }
    public void saluda(){
        System.out.println("Hola");
    }
    public   void mostrarNombre(){
        System.out.println(nombre);
```

```
    }
    public void mostrarNombre(int veces){
        for(int i=0; i <veces; i++){
            System.out.println(nombre);
        }
    }
  public void mostrarEdad(){
        System.out.println(edad);
  }
}
```

Una de las funciones didácticas interesantes que proporciona BlueJ es la posibilidad de instanciar a través de una interfaz gráfica, y no de código Java, clase Java. En la Figura 1 podemos observar dos objetos de tipo Animal creados mediante BlueJ. El objeto que tiene el nombre animal4 dentro de BlueJ fue creado mediante el constructor Animal(int, String) pasándole como argumentos el entero 12 y el nombre "Tomás", mientras que el que tiene el nombre "animal3" fue creado con el mismo constructor, pero con los argumentos 2 y "Tobi".

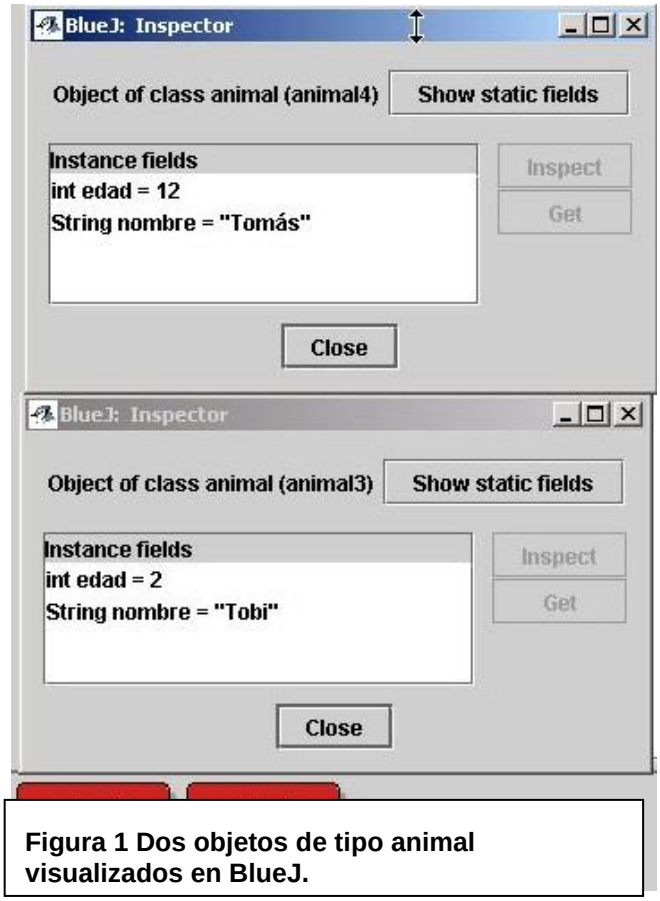


Figura 1 Dos objetos de tipo animal visualizados en BlueJ.

1.2 Modificadores de métodos y variables

Antes de explicar herencia entre clases comentaremos cuales son los posibles modificadores que pueden tener métodos y variables y su comportamiento:

Modificadores de variables

- **public**: Pública, puede acceder todo el mundo a esa variable.
- **Ninguno**: Es "amistosa", puede ser accedida por cualquier miembro del package, pero no por otras clases que pertenezcan a otro package distinto.

- **protected**: Protegida, sólo pueden acceder a ella las clases hijas de la clase que posee la variable y las que estén en el mismo package.
- **private**: Privada, nadie salvo la clase misma puede acceder a estas variables. Pueden acceder a ella todas las instancias de la clase (cuando decimos clase nos estamos refiriendo a todas sus posibles instancias)
- **static**: Estática, esta variable es la misma para todas las instancias de una clase, todas comparten ese dato. Si una instancia lo modifica todas ven dicha modificación.
- **final**: Final, se emplea para definir constantes, un dato tipo final no puede variar nunca su valor. La variable no tiene por qué inicializarse en el momento de definirse, pero cuando se inicializa ya no puede cambiar su valor. Es similar al modificador const de C.

```
class Marciano {
//esta variable será visible para los hijos y para las clases
//del mismo paquete
    boolean vivo;
//esta variable sólo la va a poder usar la propia clase
    private static int numeroMarcianos = 0;
//esta cadena de caracteres nunca podrá cambiar su valor
    final String Soy = "marciano";

    void quienEres(){
        System.out.println("Soy un " + Soy);
    }

    Marciano(){
        vivo = true;
        numeroMarcianos++;
    }
    void muerto(){
        if(vivo){
            vivo = false;
            numeroMarcianos--;
        }
    }
}
```

En la figura Figura 2 Podemos ver 4 objetos de tipo Marciano creados en BlueJ. Sobre el objeto "marciano3" se ha invocado el método *muerto*, lo cual ha provocado la muerte del marciano, es decir, la variable de tipo boolean vivo ha pasado a valer "false". Podemos observar también el resultado de la inspección de las variables estáticas de la clase, vemos que numeroMarcianos vale 3. Hemos creado 4 marcianos, con lo cual esa variable debiera tener el valor 4; hemos matado uno de los marcianos, con lo cual esa variable se decrementa y pasa a valer 3, como nos indica BlueJ.

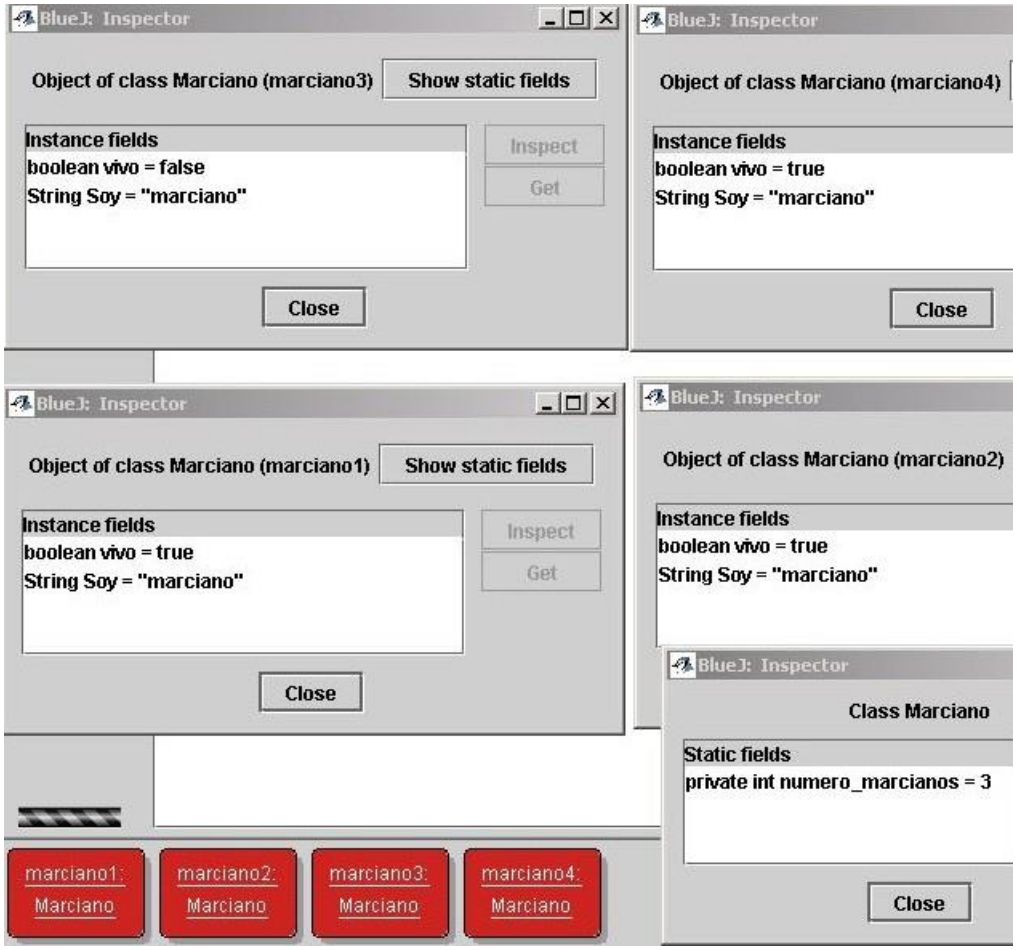


Figura 2 Cuatro objetos tipo Marciano creados e inspeccionados en BlueJ.

Modificadores de un método

- **publ**ic: Público, puede acceder todo el mundo a este método.
- Ninguno: Es "amistoso", puede ser accedida por cualquier miembro del package, pero no por otras clases que pertenecen a otro package.
- **protected**: Protegido, sólo pueden acceder a él las clases hijas de la clase que posea el método y las que estén en el mismo package.

- **private**: Privada, nadie salvo la clase misma puede acceder a estos métodos.
- **static**: Estática, es un método al cual se puede invocar sin crear ningún objeto de dicha clase. Math.sin, Math.cos son dos ejemplos de métodos estáticos. Desde un método estático sólo podemos invocar otros métodos que también sean estáticos.
- **final**: Final, se trata de un método que no podrá ser cambiado por ninguna clase que herede de la clase donde se definió. Es un método que no se puede "sobrescribir". Más adelante se explicará que es esto.

A continuación, mostramos un ejemplo de una clase que define métodos estáticos. Estos métodos se podrían invocar, por ejemplo, como Mat.cuadrado (4).

```
class Mat{
    static int cuadrado(int i){
        return i*i;
    }
    static int mitad (int i){
        return i/2;
    }
} ///:~
```

Volviendo al ejemplo del vector

En este apartado mostramos cómo se implementaría en java una clase Vector que representa un vector en tres dimensiones y permita hacer sumas vectoriales y productos vectoriales. Se correspondería con la versión orientada a objetos del programa del cual ya vimos cuatro versiones en el Tema 2.

```
public class Vector {
    protected float x, y, z;

    public Vector(float _x, float _y, float _z) {
        x = _x;
        y = _y;
        z = _z;
    }
```

```
    public Vector sumaVectorial(Vector v) {
        Vector suma = new Vector(x + v.x, y + v.y, z + v.z);
        return suma;
    }

    public Vector multiplicacionVectorial(Vector v) {
        float xm = y * v.z - z * v.y;
        float ym = -x * v.z + z * v.x;
        float zm = x * v.y - y * v.x;
        Vector multiplicacion = new Vector(xm, ym, zm);
        return multiplicacion;
    }

     public String toString() {
        return "(" + "x=" + x + ", y=" + y + ", z=" + z +
")";
    }
}
```

Este sería un ejemplo de código que usase esta clase:

```
public class VectorMain {
   public static void main(String[] args) {
        Vector v1 = new Vector (1, 0, 0);
        Vector v2 = new Vector (0, 1, 0);
        Vector v3 = new Vector (0, 0, 1);

        Vector vs = v1.sumaVectorial(v2);
        System.out.println(vs);

        Vector vp = vs.multiplicacionVectorial(v3);
        System.out.println(vp);
    }
}
```

1.3 Herencia

Cuando en Java indicamos que una clase “extends” otra clase estamos indicando que es una clase hija de esta y que, por lo tanto, hereda todos sus métodos y variables. Este es un poderoso mecanismo para la reusabilidad del código. Podemos heredar de una clase, por lo cual partimos de su estructura de variables y métodos, y luego añadir lo que necesitemos o modificar lo que no se adapte a nuestros requerimientos. Veamos un ejemplo:

```
class Animal{
    protected int edad;
    protected String nombre;
    private boolean vivo;

    public Animal(){
        edad = 0;
        nombre = "Rufo";
        vivo = true;
    }

    public Animal(int _edad, String _nombre){
        edad = _edad;
        nombre = _nombre;
        vivo = true;
    }

    public void saluda(){
        if(vivo){
           System.out.println("Hola");
        }
    }

    public void mostrarNombre(){
        System.out.println(nombre);
    }
    //método sobrecargado
    public void mostrarNombre(int veces){
        for(int i=0; i <veces; i++){
            System.out.println(nombre);
        }
    }
```

```
    public void mostrarEdad(){
        System.out.println(edad);
    }
 //es habitual crear métodos de este tipo para acceder a
//variables privadas
    boolean isVivo(){
        return vivo;
    }
//es habitual crear mtodos de este tipo para modificar valor
//variables privadas
    void setVivo(boolean _vivo){
        vivo = _vivo;
    }
}
//La clase Perro extiende a Animal, heredando sus métodos y
//variables
public class Perro extends Animal{

    Perro(){
  //Esta sentencia invoca a un constructor de la clase padre.
        super(0,"Tobi");
    }

    Perro(int edad, String nombre){
  //Esta sentencia invoca a un constructor de la clase padre.
        super(edad,nombre);
    }
//Método estático que se puede llamar sin crear un objeto //como
Perro.esteMetodoNoTieneSentidoAqui()
    static void esteMetodoNoTieneSentidoAqui(){
        System.out.println("Solo para demostrar como funcionan los
metodos estaticos");
    }

    public void mostrarTodo(){
        //mostramos la edad que hemos heredado de Animal
        System.out.println(edad);
        //mostramos el nombre que hemos heredado de Animal
        System.out.println(nombre);
        //esto daria un error compilar porque la variables privada
```

```
        //System.out.println(vivo);
        //pero podemos acceder al mtodo que devuelve esa variable
        System.out.println(isVivo());

    }

    public static void main (String[] args){
//Creamos un objeto de tipo Perro
        Perro dog = new Perro(8,"Bambi");
        dog.mostrarTodo();
    }
}
```

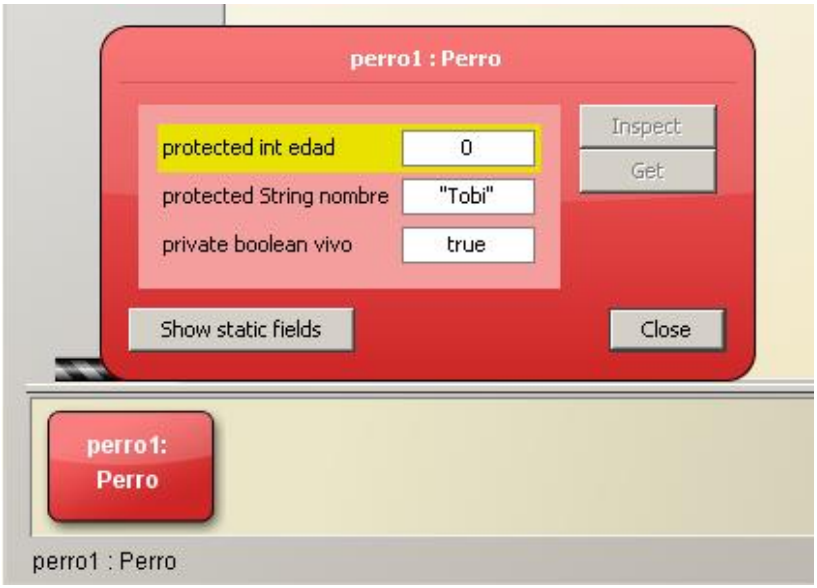
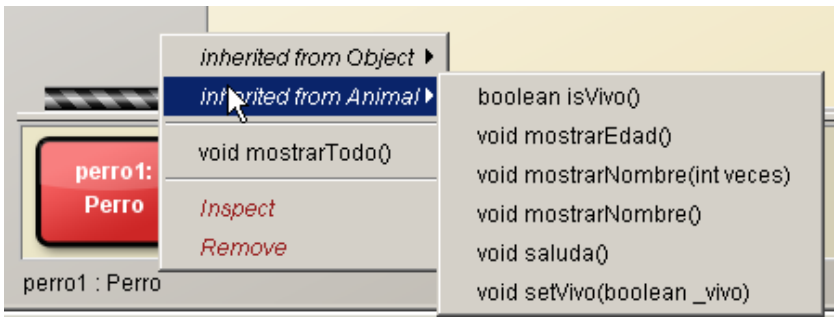


Figura 3 Inspección de un objeto tipo perro creado en BlueJ.

En la figura 3 podemos observar un objeto de tipo *Perro* creado invocando al constructor por defecto *Perro()*. Gracias a los mecanismos de inspección de BlueJ podemos observar como la variable *edad* se ha inicializado a 0, y la variable *nombre* a "Tobi", como se indicó en el constructor. La variable vivo ha sido inicializar true por el constructor de Animal. Así mismo vemos como el objeto Perro ha heredado todos los métodos definidos en la clase *Animal*.



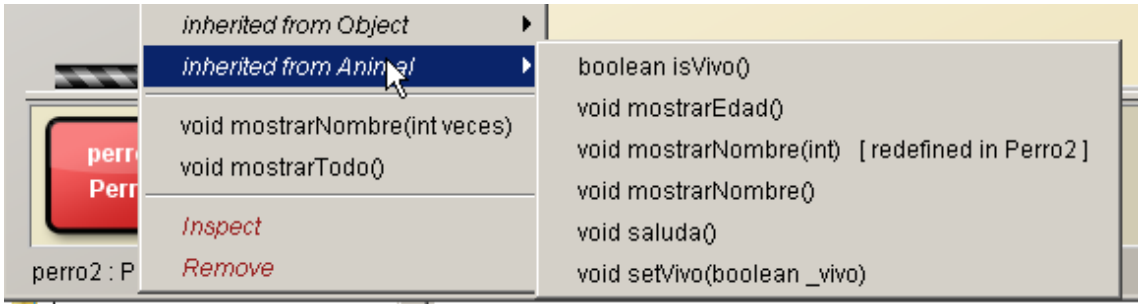
Si un método no hace lo que nosotros queríamos podemos sobrescribirlo (overriding). Bastará para ello que definamos un método con el mismo nombre y argumentos. Veámoslo sobre el ejemplo anterior:

```
public class Perro2 extends Animal{

    Perro2(){
  //Esta sentencia invoca a un constructor de la clase padre.
        super(0,"Tobi");
    }

    Perro2(int edad, String nombre){
  //Esta sentencia invoca a un constructor de la clase padre.
        super(edad,nombre);
    }
//Método estático que se puede llamar sin crear un objeto
//como Perro.esteMetodoNoTieneSentidoAqui()
    static void esteMetodoNoTieneSentidoAqui(){
        System.out.println("Solo para demostrar como funcionan los
metodos estaticos");
    }

    public void mostrarTodo(){
        //mostramos la edad que hemos heredado de Animal
        System.out.println(edad);
        //mostramos el nombre que hemos heredado de Animal
        System.out.println(nombre);
        //esto daria un error compilar porque la variables privada
        //System.out.println(vivo);
        //pero podemos acceder al mtodo que devuelve esa variable
        System.out.println(isVivo());

    }
    //método sobrescrito
    public void mostrarNombre(int veces){
        for(int i=0; i <veces; i++){
            System.out.println("Saludo numero "+i+": "+nombre);
        }
    }

    public static void main (String[] args){
//Creamos un objeto de tipo Perro
        Perro2 dog = new Perro2(8,"Bambi");
```

```
//Invocamos al método estático get1 pasándole el objeto // de tipo
Perro creado.
        dog.mostrarTodo();
        //invoca al método sobre escrito del hijo, no al del padre
        dog.mostrarNombre(5);
    }
}
```

En la figura podemos observar un objeto de tipo *Perro2*; esta vez ha sido creado invocando al constructor *Perro2(int, String)*, pasándole como argumentos el entero 6 y una cadena de caracteres con valor "Tobi_2".

Empleando los mecanismos de inspección de BlueJ podemos observar como la variable *edad* se ha inicializado a 6, y la variable *nombre* a "Tobi_2". Vemos como el objeto Perro2 ha heredado todos los métodos definidos en la clase *Animal*, y como BlueJ nos advierte de que el método getNombre(i) de la clase *Animal*, ha sido sobrescrito (redefined) en la clase hija, por lo que cuando se invoque será el código del hijo, y no el del padre, el que se ejecute



1.4 Creación y referencia a objetos

Aunque ya hemos visto como se crea un objeto vamos a formalizarlo un poco. Un objeto en el ordenador es esencialmente un bloque de memoria con espacio para guardar las variables de dicho objeto. Crear el objeto es sinónimo de reservar espacio para sus variables, inicializarlo es dar un valor a estas variables. Para crear un objeto se utiliza el comando new. Veámoslo sobre un ejemplo:

```
class Fecha{

   int dia,mes,ano;

    Fecha(){
        dia=1;
        mes = 1;
        ano = 1900;
    }

    Fecha (int _dia, int _mes, int _ano){
        dia= _dia;
        mes = _mes;
        ano = _ano;
    }
}
```

Ahora podemos hacer:

```
   Fecha hoy;
   hoy = new Fecha();
```

Con el primer comando hemos creado un puntero que apunta a una variable tipo fecha, como está sin inicializar apuntará a null. Con el segundo inicializamos el objeto al que apunta hoy, reservando espacio para sus variables. El constructor las inicializará, tomando el objeto hoy el valor 1-1-1900.

```
    Fecha unDia = new Fecha(8,12,1999);
```

Con esta sentencia creamos una variable que se llama unDia con valor 8-12-1999.

Una vez creado un objeto será posible acceder a todas sus variables y métodos públicos, así por ejemplo en el ejemplo anterior unDia.dia tomaría el valor 8. Si la variable fuese privada solo podrían acceder a ella sus instancias:

```
class Fecha{
    private int dia,mes,ano;

    Fecha(){
        dia=1;
        mes = 1;
        ano = 1900;
    }

    Fecha (int ndia, nmes, nano){
        dia= ndia;
        mes = nmes;
        ano = nano;
    }
}
```

De este modo no podríamos acceder a las variables de la clase fecha, para acceder a ella tendríamos que hacerlo mediante métodos que nos devolviesen su valor. A esto se le denomina encapsulación. Esta es la forma correcta de programar OOP: no debemos dejar acceder a las variables de los objetos por otro procedimiento que no sea paso de mensajes entre métodos.

1.5 this

Es una variable especial de sólo lectura que proporciona Java. Contiene una referencia al objeto en el que se usa dicha variable. A veces es útil que un objeto pueda referenciarse a si mismo:

```
class Cliente{
    public Cliente(String n){
//Llamamos al otro constructor. El empleo de this ha de ser
//siempre en la primera línea dentro del constructor.
        this(n, Cuenta.nuevo_numero());
        .....
```

```
    }
    public Cliente (String n, int a){
        nombre = n;
        numero_cuenta = a;
    }
    .....
}
```

Otro posible uso de this, que ya se ha visto en ejemplos anteriores es diferenciar entre variables locales de un método o constructor y variables del objeto. En los códigos correspondientes a los ejemplos Perro.java y Perro2.java el constructor de la clase Anima,l aunque realiza la misma función que los que se recogen en estos apuntos, son ligeramente diferentes:

```
    public Animal(int edad, String nombre){
   //this.edad = varaible del objeto Perro
   //edad = variable definida sólo dentro del constructor
    this.edad =edad;
    this.nombre=nombre;
    }
```

1.6 super

Del mismo modo que this apunta al objeto actual tenemos otra variable super que apunta a la clase de la cual se deriva nuestra clase. Uno de los principales usos de esta referencia es llamar desde la clase hija a código de la clase padre:

```
class Gato {
    void hablar(){
        System.out.println("Miau");
    }

}

class GatoMagico extends Gato {

    boolean gentePresente;
```

```
    void hablar(){
        if(gentePresente)
//Invoca al método sobrescrito de la clase padre
            super.hablar();

        else
            System.out.println("Hola");
    }
}
```

1.7 Un ejemplo de herencia con la clase Vector

En este apartado vamos a ver cómo podemos modificar la clase Vector para, apoyándonos en ella, representaron vector en un espacio de dos dimensiones. La suma de vectores en dos dimensiones debe ser un vector en dos dimensiones. Pero el producto vectorial de vectores en dos dimensiones debería ser un vector perpendicular, y por tanto en tres dimensiones. Y también sería deseable poder sumar vectores en dos dimensiones con vectores en tres dimensiones, obteniendo obviamente un vector de tres dimensiones.

```
//Vector2D es un Vector en 3D donde la z siempre es 0
public class Vector2D extends Vector{

    public Vector2D(float x, float y) {
        super(x , y, 0);
    }
    //Sobreescribimos el método de suma
    public Vector2D sumaVectorial(Vector2D v) {
        Vector2D suma = new Vector2D(x + v.x, y + v.y);
        return suma;
    }
    //Sobre escribimos la representación textual
     public String toString() {
        return "(" + "x=" + x + ", y=" + y + ")";
    }
}
```

Observa como hemos tenido que hacer menos trabajo al apoyarnos en la clase Vector. A continuación mostramos un código que opera con vectores 2D, que también combina operaciones con vectores en 2 y 3 dimensiones:

```
public class Vector2DMain {

    public static void main(String[] args) {
        Vector2D v1 = new Vector2D (1, 0);
        Vector2D v2 = new Vector2D (0, 1);
        Vector v3 = new Vector (0, 0, 1);

        Vector vs = v1.sumaVectorial(v2);
        System.out.println(vs);

   //Un Vector2D es un Vector así que puede multiplicarse con ellos
        //en esta ocasion el metodo que se invoca es el del padre
        Vector vp = vs.multiplicacionVectorial(v3);
        System.out.println(vp);
        //Y tambien podemos sumar un Vector2D con un Vector 3D
        //en esta ocasion el metodo que se invoca es el del padre
        Vector v4 = vs.sumaVectorial(vp);
        System.out.println(v4);
    }
}
```

2 Interfaces

En Java no está soportada la herencia múltiple, esto es, no está permitido que una misma clase pueda heredar las propiedades de varias clases padres. En principio esto pudiera parecer una propiedad interesante que le daría una mayor potencia al lenguaje de programación, sin embargo los creadores de Java decidieron no implementar la herencia múltiple por considerar que esta añade al código una gran complejidad (lo que hace que muchas veces los programadores que emplean lenguajes de programación que sí la soportan no lleguen a usarla).

Sin embargo para no privar a Java de la potencia de la herencia múltiple sus creadores introdujeron un nuevo concepto: el de interface. Una interface es formalmente como una clase, con dos diferencias: sus métodos están vacíos, no hacen nada, y a la hora de definirla en vez de emplear la palabra clave “class” se emplea “interface”. Veámoslo con un ejemplo:

```
interface Animal{

    public int edad = 10;
    public String nombre = "Bob";

    public void nace();
    public void mostrarNombre();
    void mostrarNombre(int veces);
}
```

Cabe preguntarnos cuál es el uso de una interface si sus métodos están vacíos. Bien, cuando una clase implementa una interface lo que estamos haciendo es una promesa de que esa clase va a implementar todos los métodos de la interface en cuestión. Si la clase que implementa la interface no "sobrescribiera" alguno de los métodos de la interface automáticamente esta clase se convertiría en abstracta y no podríamos crear ningún objeto de ella. Para que un método sobrescriba a otro ha de tener el mismo nombre, se le han de pasar los mismos datos, ha de devolver el mismo tipo de dato y ha de tener el mismo modificador que el método al que sobrescribe. Si no tuviera el mismo modificador el compilador nos daría un error y no nos dejaría seguir adelante. Veámoslo con un ejemplo de una clase que implementa la anterior interface:

```
interface Animal{

    public int edad = 10;
    public String nombre = "Bob";

    public void nace();
    public void mostrarNombre();
    void mostrarNombre(int veces);
}

public class Perro3 implements Animal{
    Perro3(){
        mostrarNombre();
        mostrarNombre(8);
    }
```

```
//Compruévese como si cambiamos el nombre del método a nac()
//no compila ya que no henos sobreescrito todos los métodos
//de la interfaz.
    public void nace(){
    System.out.println("hola mundo");
    }

    public void mostrarNombre(){
        System.out.println(nombre );
    }

    public void mostrarNombre(int i){
        System.out.println(nombre +" " +i);
    }

    public static void main (String[] args){
        Perro3 dog = new Perro3();
//Compruevese como esta línea da un error al compilar debido
//a intentar asignar un valor a una variable final
   //    dog.edad = 8;
    }
}
```

Las variables que se definen en una interface llevan todas ellas el atributo final, y es obligatorio darles un valor dentro del cuerpo de la interface. Además no pueden llevar modificadores private ni protected, sólo public. Su función es la de ser una especie de constantes para todos los objetos que implementen dicha interface.

Por último decir que aunque una clase sólo puede heredar propiedades de otra clase puede implementar cuantas interfaces se desee, recuperándose así en buena parte la potencia de la herencia múltiple.

```
interface Animal1{

    public int edad = 10;
    public String nombre = "Bob";
```

```
    public void nace();
}

 interface Animal2{

    public void mostrarNombre();
}

 interface Animal3{

    void mostrarNombre(int i);

}

public class Perro4 implements Animal1,Animal2,Animal3{
    Perro4(){
        mostrarNombre();
        mostrarNombre(8);
        //edad = 10; no podemos cambiar este valor
    }
    public void nace(){
    System.out.println("hola mundo");
    }

    public void mostrarNombre(){
        System.out.println(nombre );
    }

    public void mostrarNombre(int i){
        System.out.println(nombre +" " +i);
    }

    public static void main (String[] args){
        Perro4 dog = new Perro4();
    }
}
```

3 Nuestro primer programa orientado a objetos

Para intentar hacer este tema un poco menos teórico y ver algo de lo que aquí se ha expuesto se ha realizado el siguiente programilla; en él se empieza una ficticia guerra entre dos naves, una de marcianos y otra de terrícolas, cada uno de los cuales va disparando, generando números aleatorios, y si acierta con el número asignado a algún componente de la otra nave lo "mata". No pretende ser en absoluto ninguna maravilla de programa, simplemente se trata con él de romper el esquema de programación estructurada o clásica, en el cual el programa se realiza en el main llamando a funciones. En OOP un programa, digámoslo una vez más, es un conjunto de objetos que dialogan entre ellos pasándose mensajes para resolver un problema. Veremos cómo, ni más ni menos, esto es lo que aquí se hace para resolver un pequeño problema-ejemplo.

Antes de pasar al código, veamos la representación UML del diagrama de clases que componen nuestro problema visualizado en BlueJ:

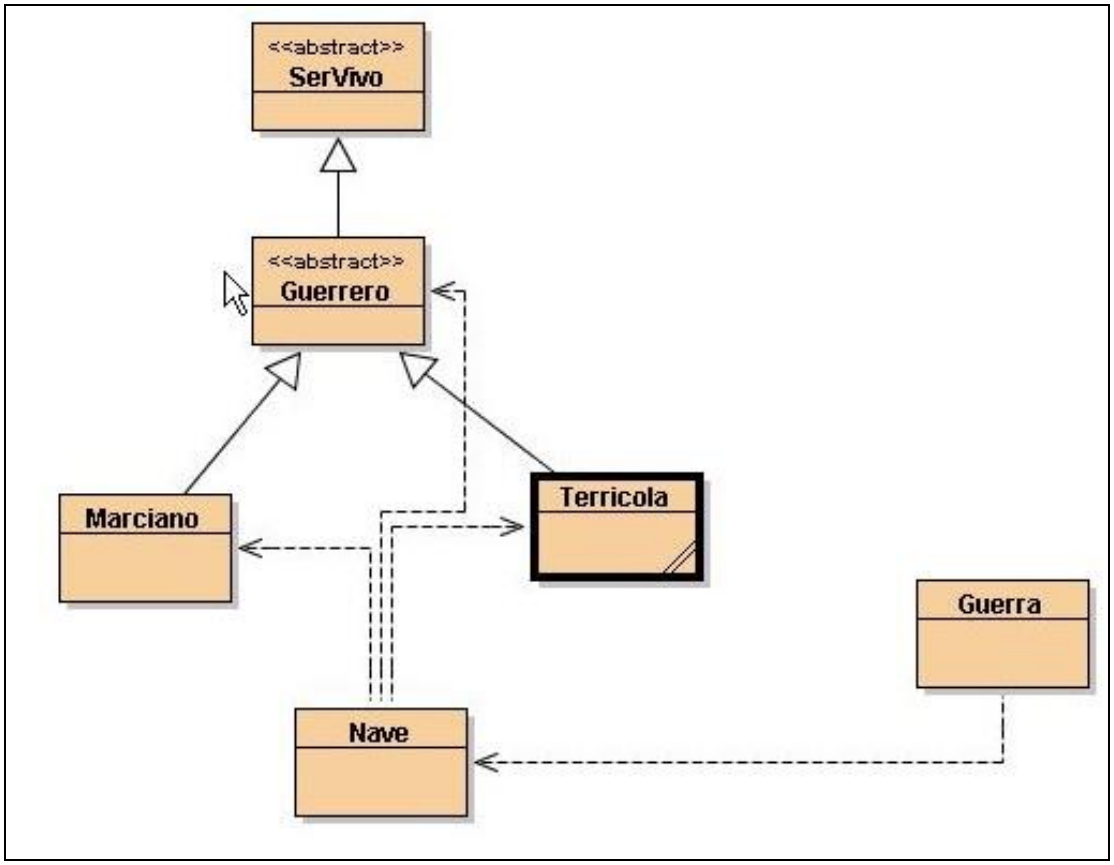


Figura 4.- Diagrama de clases de nuestro problema visualizado en BlueJ. Contamos con una clase Abstracta SerVivo, que representa a un ser vivo. Guerrero es nuevamente una clase abstracta que hereda de SerVivo (un guerrero es un ser vivo); en ella se sitúa el comportamiento de atacar. Tenemos dos clases, Terrícola y Marciano, que representarán a terrícolas y marcianos. La clase Nave actúa como un contenedor cuya tripulación puede ser un conjunto de terrícolas o de marcianos. Por último Guerra es una clase auxiliar encargada de la creación de las naves y de la simulación de la guerra.

La clase SerVivo simplemente representa a un ser que está vivo:

```
public abstract class SerVivo {
    protected boolean vivo = true;
    public boolean isVivo() {
        return vivo;
    }
}
```

La clase Guerrero representa un ser vivo con capacidad de combatir; tiene una variable donde se guarda un número aleatorio entre 0 y 10; si alguien acierta con ese número el guerrero se morirá. Además, tiene capacidad para disparar, esto es, generar números aleatorios que representan un disparo a un adversario (método dispara()). También tiene un método que se invoca cuando alguien le ha disparado a él y comprueba si tiene o no que morirse (si han acertado o no con su número aleatorio). Observa como también hay un método privado; este método privado forma parte de un detalle de implementación de la clase y ninguna otra clase necesita conocerlo.

```
public abstract class Guerrero extends SerVivo {

    protected int blancoAAdivinar;
    protected final String soy;
    public static final int MAX_BLANCO = 10;

    public Guerrero(String soy) {
        blancoAAdivinar = generaBlancoAAdivinar();
        this.soy = soy;
    }

    public int dispara() {
        if (vivo) {
            int disparo = ((int) (Math.random() * (MAX_BLANCO+1)));
            System.out.println(soy + " dispara n " + disparo);
            return disparo;
        } else {
            return Integer.MIN_VALUE;
        }
    }

    public boolean recibeDisparo(int disparo) {
        boolean seMuere = false;
        if (vivo && blancoAAdivinar == disparo) {
            vivo = false;
```

```
            seMuere = true;
            System.out.println(soy + " muerto por disparo n " + disparo);
        }
        return seMuere;
    }

    public int getBlancoAAdivinar() {
        return blancoAAdivinar;
    }

    private int generaBlancoAAdivinar() {
        return ((int) (Math.random() * (MAX_BLANCO+1)));
    }
}
```

La clase Marciano representa a los marcianos. Los marcianos emplean una variable estática (totalMarcianosVivos) para contar cuántos marcianos quedan vivos en cada momento. Esta variable estática se incrementa en su constructor. Su constructor también llama al constructor de Guerrero y le pasa la cadena de caracteres que contiene el texto informativo de qué tipo de Guerrero se trata (un marciano en este caso). Observa como además Marciano modifica la forma en la que los guerreros reciben los disparos: los marcianos son más duros de matar y hay que dispararles un número de veces igual a DISPAROSQUEAGUANTA para conseguir matarlo. Una vez que se le ha disparado ese número de veces, el método recibeDisparo() del Marciano llama al método recibeDisparo() de Guerrero, que es el que realmente se encarga de ejecutar el código necesario para "matar" al Marciano y mostrar el mensaje correspondiente en pantalla.

```
public class Marciano extends Guerrero {

    private static int totalMarcianosVivos = 0;
    private int disparosRecibidos = 0;
    public static final int DISPAROS_QUE_AGUANTA = 3;

    Marciano(String soy) {
        super(soy);
        totalMarcianosVivos++;
    }

    public boolean recibeDisparo(int disparo) {
        boolean seMuere = false;
        if (vivo && blancoAAdivinar == disparo) {
            disparosRecibidos++;
            //Los marcianos aguantan varios disparos
```

```
            if (disparosRecibidos == DISPAROS_QUE_AGUANTA) {
                totalMarcianosVivos--;
                super.recibeDisparo(disparo);
                seMuere = true;
            }
        }
        return seMuere;
    }

    public static int getTotal() {
        return totalMarcianosVivos;
    }
}
```

La clase Terrícola también tiene una variable estática que se emplea para contar cuántos terrícolas quedan vivos en cada momento (totalTerricolas). Observa como se ha sobrescrito el método recibeDisparo(); en este caso no estamos realmente cambiando el comportamiento del método de la clase padre (Guerrero); simplemente estamos haciendo otra cosa adicional: decrementar el número total de terrícolas que quedan vivos. Pero los terrícolas se mueren igual que los guerreros. Sin embargo, los terrícolas no disparan igual que los guerreros. Los terrícolas son más inteligentes a la hora de disparar, y llevan cuenta de los "disparos" realizados; esto es, de los números aleatorios que han generado. Cuando generan un número aleatorio un número de veces igual a Marciano.DISPAROS_QUE_AGUANTA entonces no vuelven a generar nunca más ese número aleatorio; generar de nuevo ese número aleatorio es inútil porque todos los marcianos que se pueden "matar" con ese número ya se han muerto. Por tanto, si ese número aleatorio se vuelve a generar "repiten el disparo".

Es interesante observar en las clases Terrícola y Marciano como ambos pueden "disparar" y "recibir disparos", y ambas lo hacen a través de la misma interfaz (que está definida en la clase Guerrero), pero ambas lo hacen de modo diferente. Los terrícolas disparan de un modo más inteligente, pero se mueren con un solo disparo. Los marcianos disparan de un modo más tonto (pueden repetir números aleatorios) pero son más resistentes y requieren tres disparos para morirse.

```
import java.util.Arrays;

public class Terricola extends Guerrero {

    private static int totalTerricolas = 0;
    private static int[] disparosHechos = new int[Guerrero.MAX_BLANCO+1];
```

```
    Terricola(String soy) {
        super(soy);
        totalTerricolas++;
        Arrays.fill(disparosHechos, 0);
    }

    public boolean recibeDisparo(int disparo) {
        boolean seMuere = super.recibeDisparo(disparo);
        if (seMuere) {
            totalTerricolas--;
        }
        return seMuere;
    }

    public int dispara() {
        int disparo;
        if (vivo) {
            do {
                disparo = ((int) (Math.random() * (MAX_BLANCO+1)));
                //Nosotros disparamos modo más inteligente
            }
while(disparosHechos[disparo]>=Marciano.DISPAROS_QUE_AGUANTA);

            disparosHechos[disparo]++;
            System.out.println(soy + " dispara n " + disparo +"; van " +
disparosHechos[disparo] + " disparos de este numero");
            return disparo;
        } else {
            return Integer.MIN_VALUE;
        }
    }

    public static int getTotal() {
        return totalTerricolas;
    }
}
```

La clase Nave hace de contenedor y contiene una tripulación de guerreros, que pueden ser o bien marcianos o bien terrícolas. Observa cómo el constructor de la clase nave, en función de la cadena de caracteres que se le pasa como argumento, decide si tiene que crear Marcianos o Terrícolas para guardar en el array de Guerreros. Las naves también son las que generan los disparos; para ello le piden a un tripulante que genere el disparo (generaDisparo(int tripulante)). También son ellas las que reciben inicialmente el disparo (recibeDisparo(int disparo)), y notifican a todos los tripulantes de que han

recibido un disparo para que éstos decidan si tienen o no que hacer algo (morirse si han adivinado su número aleatorio y se trata de un terrícola, o decrementar su contador de disparos recibidos si se trata de un marciano). Observa cómo el objeto Nave fundamentalmente delega en su tripulación para todas sus actividades; es la tripulación la que dispara, y es la tripulación la que recibe los disparos.

```
public class Nave {

    private Guerrero[] tripulacion;
    private String somos;

    public Nave(String somos, int tripulantes) {
        this.somos = somos;
        tripulacion = new Guerrero[tripulantes];

        if (somos.equals("Terricolas")) {
            for (int i = 0; i < tripulantes; i++) {
                tripulacion[i] = new Terricola(somos);
            }
        } else {
            for (int i = 0; i < tripulantes; i++) {
                tripulacion[i] = new Marciano(somos);
            }
        }
        System.out.println( "Creada nave de " + somos + " con " +
cuantosQuedan() + " tripulantes");
    }

    public void recibeDisparo(int disparo) {
        for (int j = 0; j < tripulacion.length; j++) {
            if (somos.equals("Terricolas")) {
                ((Terricola) (tripulacion[j])).recibeDisparo(disparo);
            } else {
                ((Marciano) (tripulacion[j])).recibeDisparo(disparo);
            }
        }
    }

    public int generaDisparo(int tripulante) {
        return tripulacion[tripulante].dispara();
    }

    public int cuantosQuedan() {
        if (somos.equals("Terricolas")) {
            return Terricola.getTotal();
```

```
        } else {
            return Marciano.getTotal();
        }
    }
}
```

La última clase es Guerra; esta será la clase que cree las dos naves que van a combatir. Y es la que hace que las dos naves "se peleen". En esta guerra sólo hay dos naves, y la una dispara a la otra. Pero sería trivial extender este código para tener una flota de naves de terrícolas y de marcianos que participasen en una misma guerra.

```
public class Guerra {

    private Nave naveTerricolas, naveMarcianos;
    private int numeroDeTerricolas, numeroDeMarcianos;

    public Guerra(int numeroDeTerricolas, int numeroDeMarcianos) {
        this.numeroDeTerricolas = numeroDeTerricolas;
        this.numeroDeMarcianos = numeroDeMarcianos;
        naveTerricolas = new Nave("Terricolas", numeroDeTerricolas);
        naveMarcianos = new Nave("Marcianos", numeroDeMarcianos);
    }

    public void empiezaGuerra() {
        do {
            int tamanoMaxTripulaciones =
                    numeroDeTerricolas > 20 ? numeroDeTerricolas :
numeroDeMarcianos;

            for (int i = 0; i < tamanoMaxTripulaciones &&
quedanVivosEnAmbasNaves(); i++) {
                if (i < numeroDeTerricolas) {

naveMarcianos.recibeDisparo(naveTerricolas.generaDisparo(i));
                }

                if (i < numeroDeMarcianos) {

naveTerricolas.recibeDisparo(naveMarcianos.generaDisparo(i));
                }
            }
        } while (quedanVivosEnAmbasNaves());
```

```
        if (naveTerricolas.cuantosQuedan() > 0) {
            System.out.println("GANARON LOS TERRICOLAS!!!!!");
        } else if (naveMarcianos.cuantosQuedan() > 0) {
            System.out.println("GANARON LOS MARCIANOS");
        }
    }

    private boolean quedanVivosEnAmbasNaves() {
        return naveTerricolas.cuantosQuedan() > 0 &&
naveMarcianos.cuantosQuedan() > 0;
    }

    public static void main(String[] args) {
        Guerra guerra = new Guerra(30, 10);
        guerra.empiezaGuerra();
    }
}
```

4 Aprendiendo a usar los paquetes

A estas alturas deberías tener claro que una clase tiene una parte privada que oculta a los demás y que no es necesario conocer para poder acceder a la funcionalidad de la clase. Si hacemos cambios a la parte privada de la clase, mientras se respete la parte pública, cualquier código cliente que emplee la clase no se dará cuenta de dichos cambios.

Imagínate que tú y un compañero vais a construir en un programa complejo juntos. Os repartís el trabajo entre los dos y cada uno de vosotros implementa su parte como un montón de clases Java. Cada uno de vosotros en su código va a emplear parte de las clases del otro. Por tanto, os ponéis de acuerdo en las interfaces de esas clases. Sin embargo, cada uno de vosotros para construir la funcionalidad de esas clases probablemente se apoye en otras clases auxiliares. A tu compañero le dan igual las clases auxiliares que tú emplees. Es más, dado que el único propósito de esas clases es servir de ayuda para las que realmente constituyen la interface de tu parte del trabajo sería contraproducente que él pudiese acceder a esas clases que son detalles de implementación: tú en el futuro puedes decidir cambiar esos detalles de implementación y cambiar esas clases, modificándolas o incluso eliminándolas.

Dada esta situación ¿no sería interesante poder "empaquetar" tu conjunto de clases de tal modo que ese "paquete" sólo dejase acceder a tu compañero a las clases que tú quieras y oculte las demás?. Esas clases a las que se podría acceder serían la interface de ese "paquete". Serían "públicas". Dentro del paquete tú puedes meter cuantas más clases quieras. Pero esas no serán vistas por tu compañero y podrás cambiarlas en cualquier momento sin que él tenga que modificar su código. Es la misma idea que hay detrás de una clase pero llevada a un nivel superior: una clase puede definir cuáles de sus partes son

accesibles y no accesibles para los demás. El paquete permitiría meter dentro cuantas clases quisieras pero mostraría al exterior sólo aquellas que considere adecuado.

Esta es precisamente la utilidad de los *package* en Java. Empaquetar un montón de clases y decidir cuáles serán accesibles para los demás y cuáles no. Para empaquetar las clases simplemente debemos poner al principio del archivo donde definimos la clase, en la primera línea que no sea un comentario, una sentencia que indique a qué paquete pertenece:

```
package mipaquete;
```

Una clase que esté en el paquete "mipaquete" debe situarse dentro de un directorio con nombre "mipaquete". En Java los paquetes se corresponden con una jerarquía de directorios. Por tanto, si para construir un programa quiero emplear dos paquetes diferentes con nombres "paquete1" y "paquete2" en el directorio de trabajo debo crear dos subdirectorios con dichos nombres y colocar dentro de cada uno de ellos las clases correspondientes. En la figura, el directorio de trabajo desde el cual deberíamos compilar y ejecutar la aplicación es "paquetes". En cada uno de los dos subdirectorios colocaremos las clases del paquete correspondiente.

Cuando una clase se encuentra dentro de un paquete el nombre de la clase pasa a ser "nombrepaquete.NombreClase". Así, la clase "ClasePaquete1" que se encuentra físicamente en el directorio " paquete1" y cuya primera línea de código es:

```
package paquete1;
```

tendrá como nombre completo "paquete1.ClasePaquete1". Si deseamos, por ejemplo, ejecutar el método main de dicha clase debemos situarnos en el directorio "Paquetes" y teclear el comando:

```
java paquete1.ClasePaquete1
```

Para todos los efectos, el nombre de la clase es " paquete1.ClasePaquete1". Cuando en una clase no se indica que está en ningún paquete, como hemos hecho hasta ahora en todos los ejemplos de este tutorial, esa clase se sitúa en el "paquete por defecto" (default package). En ese caso, el nombre de la

clase es simplemente lo que hemos indicado después de la palabra reservada class sin precederlo del nombre de ningún paquete.

Es posible anidar paquetes; por ejemplo, en el directorio "paquete1" puedo crear otro directorio con nombre "paquete11" y colocar dentro de él la clase "OtraClase". La primera línea de dicha clase debería ser:

```
package paquete1.paquete11;
```

y el nombre de la clase será "paquete1.paquete11.OtraClase".

¿Cómo indico qué clases serán visibles en un paquete y qué clases no serán visibles?. Cuando explicamos cómo definir clases vimos que antes de la palabra reservada class podíamos poner un modificador de visibilidad. Hasta ahora siempre hemos empleado el modificador public. Ese modificador significaría que la clase va a ser visible desde el exterior, forma parte de la interfaz del paquete. Si no ponemos el modificador public la clase tendrá visibilidad de paquete, es decir, no será visible desde fuera del paquete pero sí será visible para las demás clases que se encuentren en el mismo paquete que ella. Aunque hay más opciones para el modificador de visibilidad de una clase, para un curso básico como éste estas dos son suficientes.

Por tanto, poniendo o no poniendo el modificador public podemos decidir qué forma parte de la interfaz de nuestros paquetes y qué no.

¿Y cómo hacemos para emplear clases que se encuentren en otros paquetes diferentes al paquete en el cual se encuentra nuestra clase?. Para eso es precisamente para lo que vale la sentencia import. Para indicar que vamos a emplear clases de paquetes diferentes al nuestro. Así, si desde la clase "MiClase" que se encuentre definida dentro de "paquete1" quiero emplear la clase "OtraClase" que se encuentra en "paquete2" en "MiClase" debo añadir la sentencia:

```
import paquete2.OtraClase;
```

A partir de ese momento, si OtraClase era pública, podré acceder a ella y crear instancias. El importar una clase sólo será posible si dicha clase forma parte de la interfaz pública del paquete. También podemos escribir la sentencia:

```
import paquete2.*;
```

Que haría accesibles todas las clases públicas que se encuentren en "paquete2", y no sólo una como el ejemplo anterior.

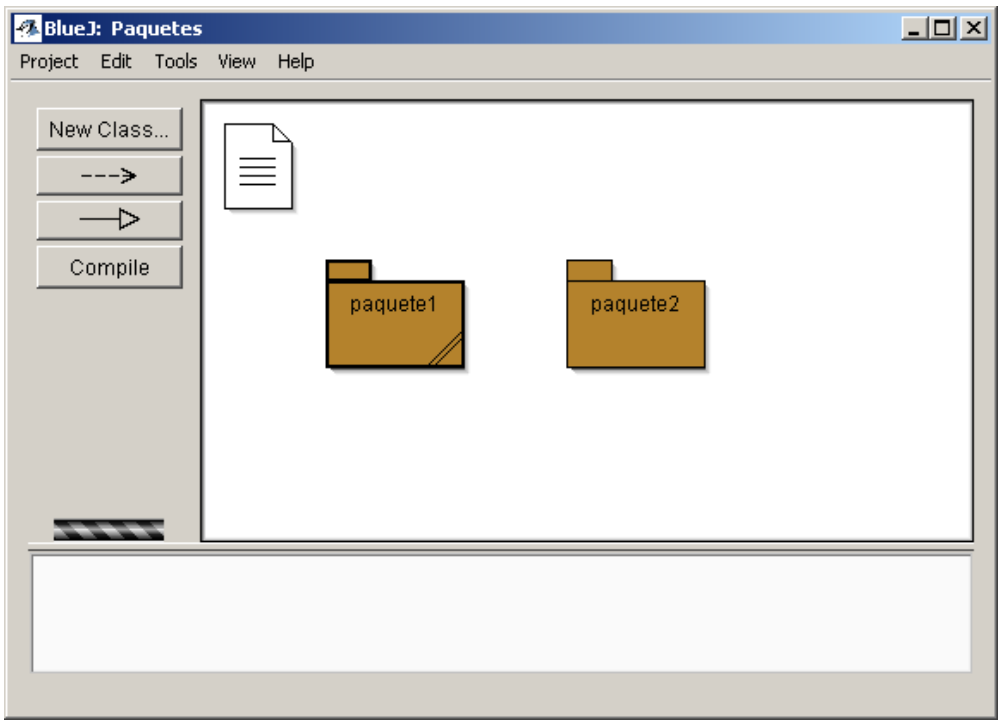
Una opción alternativa a emplear la sentencia import es emplear el nombre completo de la clase cuando vayamos a acceder a ella para crear un objeto o para invocar uno de sus métodos estáticos. Así, si no hemos importado las clases del paquete2, para crear un objeto de una de sus clases deberemos escribir:

```
paquete2.OtraClase objeto = new paquete2.OtraClase ();
```

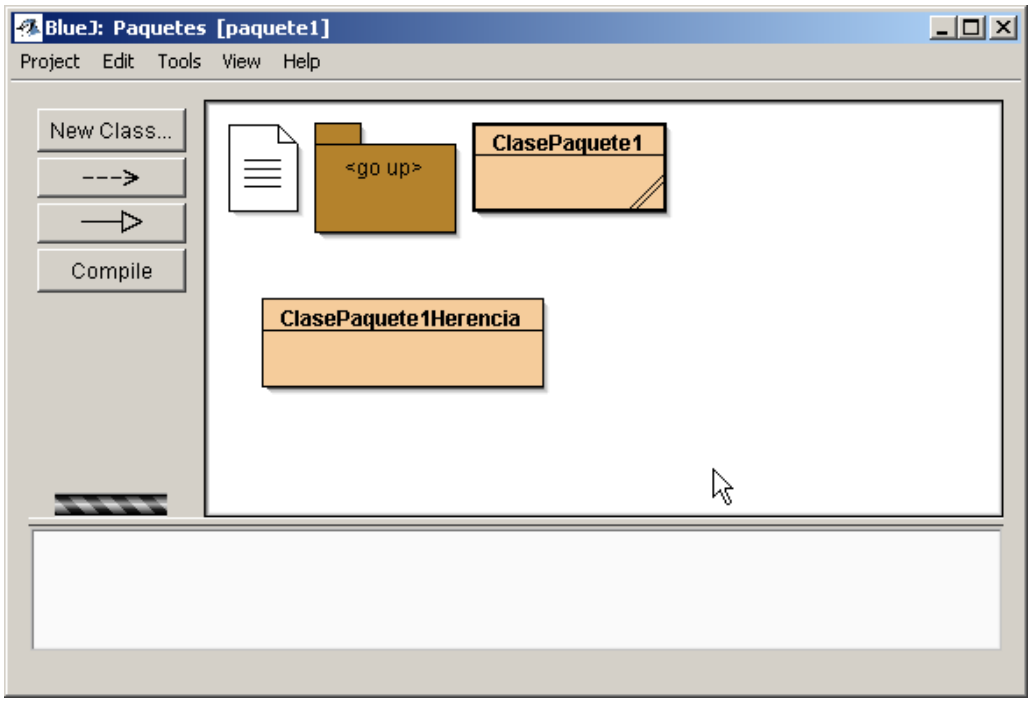
Es posible que las clases que estén dentro de un paquete hereden de clases que forman la parte pública de otro paquete. En este caso, se aplican las normas que ya hemos presentado para la herencia: la clase hija podrá acceder a la parte pública, protegida y de visibilidad de paquete de la clase padre.

4.1 Un ejemplo de código con paquetes

Vamos a ver un código en el que se ponen un uso los conceptos que estamos presentando aquí. Este código se encuentra en el directorio "Paquetes". Dicho directorio es un proyecto de BlueJ. El proyecto contiene dos paquetes, como se muestra en la imagen.

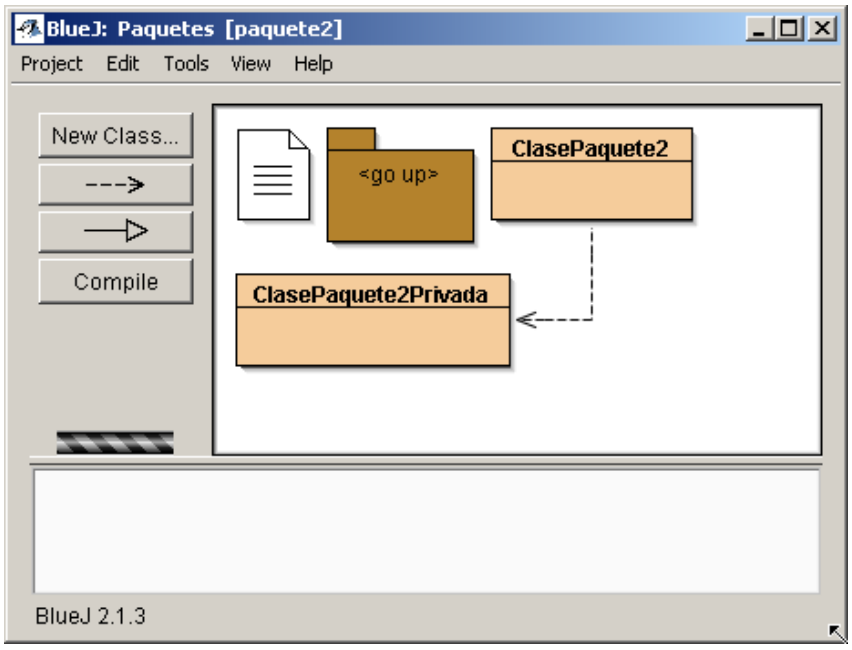


Para crear un paquete desde BlueJ se emplea el menú de edición, en el cual hay una entrada que permite crear paquetes. Para "meternos dentro de un paquete" hacemos doble clic sobre él y podemos ver su contenido. Por ejemplo, el contenido de "paquete1" es:



El icono con forma de paquete y con el texto "go up" permite subir un nivel en el anidamiento de los paquetes. En nuestro caso, permitirá ir al nivel raíz que contiene los dos paquetes. Cuando estamos visualizando el contenido de un paquete si creamos una nueva clase en BlueJ dicha clase se creará dentro de ese paquete. Las clases de este primer paquete son las que van a usar las clases del segundo paquete. En concreto, la clase "ClasePaquete1" empleará a una clase del segundo paquete (creará a una instancia de ella e invocará métodos) y la clase "ClasePaquete1Herencia" heredará de una clase del otro paquete.

El contenido del segundo paquete es:



Cómo podrás deducir a partir de los nombres de las clases, una de ellas es accesible desde fuera y por tanto será la interfaz de "paquete2", mientras que la otra no lo es. Eso sí, como también se muestra en la imagen, la clase accesible desde fuera del paquete emplea a la clase no accesible desde fuera. Esa clase es "un detalle de implementación" de este paquete. Si en el futuro la modificamos, la eliminamos, creamos más clases para repartir sus responsabilidades... ningún código que emplee "paquete2" se dará cuenta de dichos cambios ya que nunca conoció la existencia de esa clase.

Veamos ahora el código de cada una de las clases:

```
   package paquete2;
   //esta clase tienen visibilidad de paquete, formará parte de los
detalles de implementación de este paquete.
   class ClasePaquete2Privada{

        public void visibilidadPublica(){
              System.out.println("Mensaje del método con visibilidad
pública de la clase con visibilidad de paquete");

        }

        void visibilidadPaquete(){
              System.out.println("Mensaje del método con visibilidad de
paquete de la clase con visibilidad de paquete");
      }

        protected void visibilidadProtegida(){
              System.out.println("Mensaje del método con visibilidad
protegida de la clase con visibilidad de paquete");
      }

          private void visibilidadPrivada (){
              System.out.println("Mensaje del método privado de la clase
con visibilidad de paquete");
             }

   }
```

```
   package paquete2;

   /*Esta clase es pública, por lo tanto formará parte de la interfaz del
paquete*/
   public class ClasePaquete2{
```

```
          public void saludar(){
                System.out.println("Hola");
          //por supuesto, la clase puede acceder a sus métodos privados
                this.privado();
                //protegidos
                this.visibilidadProtegida();
                //y de paquete
                this.visibilidadPaquete();
           //aquí usamos los "detalles de implementación" del paquete
              ClasePaquete2Privada objeto2 = new ClasePaquete2Privada();
              //por supuesto, pueda acceder a su parte pública
              objeto2.visibilidadPublica();
   //y, como estoy en el mismo paquete, a la parte comisaría de paquete
              objeto2.visibilidadPaquete ();
   //tambien a la parte protegida porque estamos en el mismo paquete
              objeto2.visibilidadProtegida ();
//pero no la privada; descomentar esta linea daria un error de compilacion
   //         objeto2.visibilidadPrivada ();
        }

        void visibilidadPaquete(){
              System.out.println("Mensaje del método con visibilidad de
paquete");
      }

        protected void visibilidadProtegida(){
              System.out.println("Mensaje del método con visibilidad
protegida");
      }

          private void privado (){
              System.out.println("Mensaje del método privado");
          }
   }
```

Ahora veamos el contenido de "paquete1". Sus clases tienen métodos main y, por tanto, se pueden ejecutar. Las dos clases de este paquete emplean la clase pública del paquete anterior, o bien porque crea un objeto de ella o bien porque heredan de ella.

```
   package paquete1;

   import paquete2.*;
```

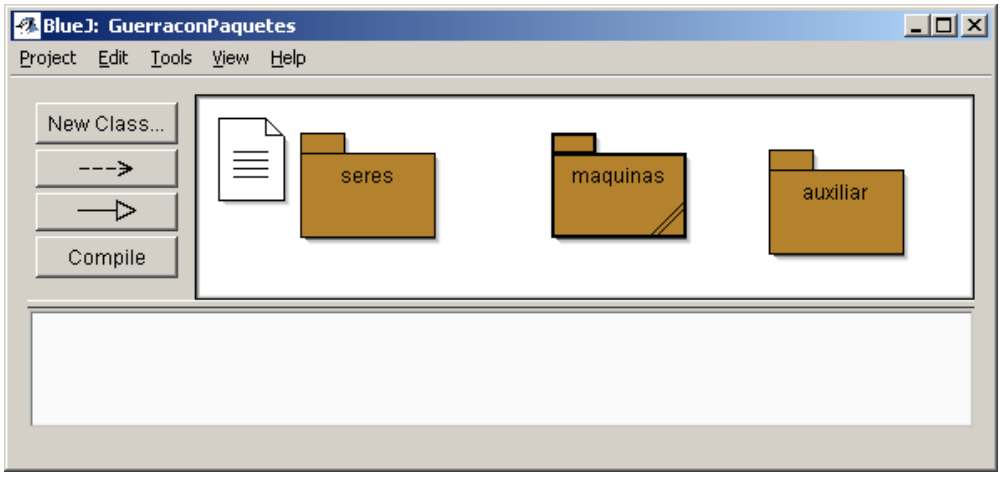
```
   public class ClasePaquete1{
          public static void main (String[] args){
              ClasePaquete2 objeto = new ClasePaquete2();
              //A la siguiente otra clase no podemos acceder; descomentar
//esta linea daria un error de compilacion
   //         ClasePrivadaPaquete2 objeto = new ClasePrivadaPaquete2();
           objeto.saludar();
           //no puedo acceder al método privado:
   //        objeto.privado ();
           //ni al que tiene visibilidad de paquete
   //        objeto.visibilidadPaquete ();
      }
   }
```

```
   package package paquete1;

   import paquete2.*;
   //la clase hereda de ClasePaquete2, por tanto va a poder acceder a sus
//partes protegidas
   //y con visibilidad de paquete adems de, por supuesto, a la parte
//pública
   public class ClasePaquete1Herencia extends ClasePaquete2{

           public static void main (String[] args){
           ClasePaquete1Herencia objeto = new ClasePaquete1Herencia();
           //no puedo acceder al método privado:
   //        objeto.visibilidadPrivada()
           //ni al que tiene visibilidad de paquete
   //        objeto.visibilidadPaquete();
           //pero si al protegido
           objeto.visibilidadProtegida();
           //y al publico
           objeto.saludar();
      }
   }
```

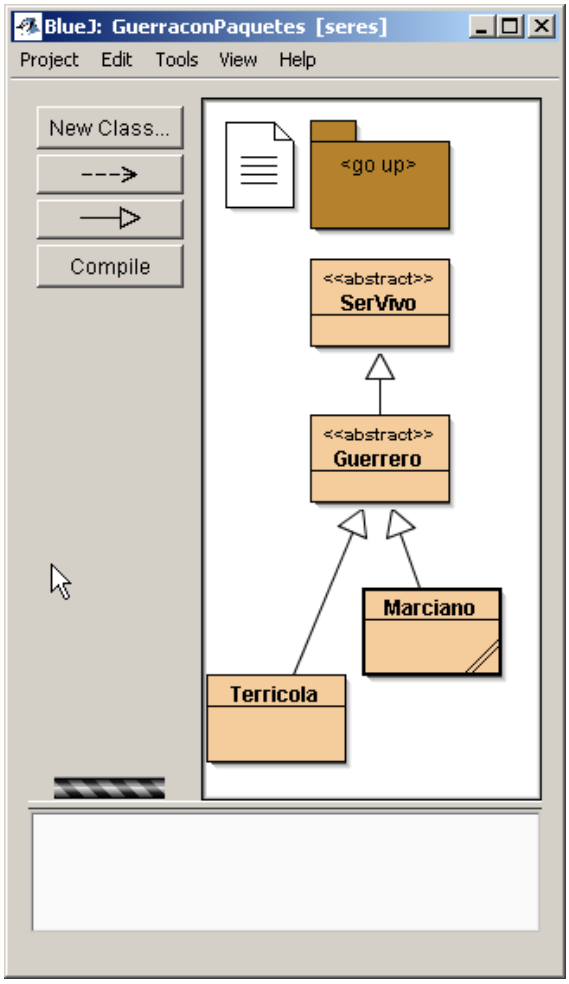
4.2 El ejemplo de los marcianos con paquetes

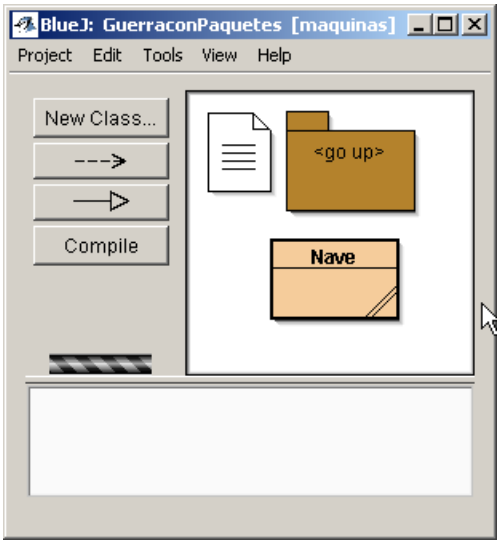
En esta sección vamos a reorganizar el código de la guerra entre marcianos y terrícolas para emplear paquetes. La principal funcionalidad de los paquetes es precisamente ayudar a organizar programas complejos. El cómo organizar un código en paquetes depende completamente del propósito de dicho código y de cómo se quiera diseñar el software. Con el tiempo irás ganando experiencia e irás aprendiendo a organizar tus clases en paquetes de modo adecuado.

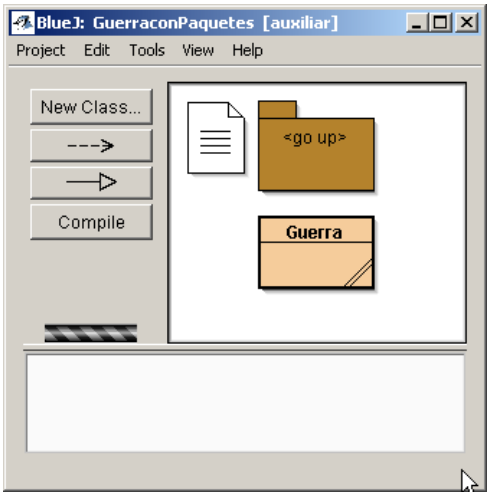
Para un ejemplo tan trivial como éste podría perfectamente argumentarse que con emplear un único paquete es más que suficiente. No obstante, vamos a reorganizar el código empleando tres paquetes. Uno de los paquetes lo llamaremos "seres" porque en él será donde vayan todos los objetos del universo que consideramos "seres". En nuestro caso, irán Marciano, Terrícola, Guerrero y SerVivo. Crearemos un segundo paquete llamado "máquinas" donde irían todas las máquinas que implementásemos. En nuestro caso, la única máquina en el problema es Nave. Por último, creamos un paquete "auxiliar" en el cual colocamos la clase que contiene el método main.



Los únicos cambios que será necesario hacer en el código es incluir las sentencias que indican a qué paquete pertenece cada clase, colocar cada clase en el directorio adecuado y, cuando una clase emplee clases de otros paquetes diferentes al suyo, añadir la sentencia import correspondiente. Puedes consultar el código fuente en los códigos de ejemplo que vienen con los apuntes; aquí vamos a admitirlo porque es prácticamente idéntico al ejemplo ya presentado. La apariencia de los tres paquetes que forman el programa será:







5 Excepciones

A estas alturas te habrás dado cuenta que gestionar los posibles errores que pueden suceder en un programa es una de las tareas más tediosas y complicadas que debe realizar un programador. En C la única forma que tenemos de comunicar que ha sucedido un error en un módulo o una función es devolver un código de error; esto es, hacer que la función en vez de devolver un dato válido devuelva un dato imposible que será interpretado como un error por el código que la invoca. Por ejemplo, si tenemos una función que debe devolver el tamaño de un array dinámico y esta función se invoca antes de que el array haya sido inicializado la función podría devolver "-1".

Este mecanismo de gestión de errores es extremadamente débil y limitado. ¿Qué pasaría si la función pudiese devolver cualquier valor? Por ejemplo, si una función devuelve la suma de dos números ¿Cómo podría la función indicarle al código que le invoca que se ha producido un overflow?. Por otro lado,

ahora que estamos programando en Java ¿qué pasa si algo va mal dentro de un constructor? Los constructores, por definición, nunca devuelven nada. No es posible chequear el valor devuelto. Además, el hecho de que devolvamos un código de error no garantiza que quien ha invocado al método correspondiente le preste atención. Por último, si sólo contamos con este mecanismo de gestión de errores cuando nos enteramos de que ha sucedido un error a veces es demasiado tarde para hacer nada. Por ejemplo, imaginemos que intentamos hacer una reserva de memoria dinámica de un tamaño superior a la memoria disponible empleando el operador new. Sucederá un error que hará a que termine el programa sin que podamos gestionar de ningún modo este fallo en nuestro código fuente; esto es, no se nos va a dar la opción a comprobar un valor de retorno.

La gestión de errores en base a códigos de retorno es, sin duda, insuficiente en muchos escenarios. De ahí que en Java se creasen las excepciones. Las excepciones son objetos que se "lanzan" desde los métodos o constructores para indicar que algo ha ido mal. Para lanzar excepciones se emplea la palabra reservada throw. Por ejemplo:

```
if (condiciónError) throw new Exception();
```

En cuanto se lanza la excepción se termina la ejecución del método y se devuelve el control al código que lo invocó. En Java cuando se invoca a un método pueden suceder dos cosas: que se ejecute correctamente y, posiblemente, nos devuelva un valor o que se lance una excepción.

Cuando se invoca a una función que puede lanzar excepciones podemos hacer dos cosas con las excepciones: gestionarlas o ignorarlas. Si una excepción se ignora produce la inmediata terminación del método que ha ignorado la excepción y ésta continúa "escalando" en el stack de llamadas a métodos. Si la excepción recorre todo el stack sucederá un error en el proceso en el cual se ejecutaba el programa y el sistema operativo lo terminará. Si en algún momento una función gestiona el error el programador tiene la posibilidad de solucionarlo o bien arreglando el problema que se ha producido o, al menos, mostrando un mensaje de error al usuario.

Cuando sabemos que en un fragmento de código se puede producir un error y queremos gestionarlo debemos rodear dicho fragmento de código con un bloque try-catch:

```
try { // bloque de código en el cual se puede lanzar una
excepción
...
}
catch (Exception ex) { // bloque que maneja las posibles
excepciones
...
}
```

El código que puede generar excepciones debe ir dentro del bloque try. Si se produce una excepción en cualquier línea de ese bloque se detiene inmediatamente la ejecución y se pasa control al bloque catch.

Entre los paréntesis del bloque catch debe indicarse el tipo de dato al que pertenece la excepción que se desea capturar. En un mismo bloque de código try podrían lanzarse muchos tipos diferentes de excepciones y puede que en el bloque catch correspondiente sólo se pueda gestionar un subconjunto de esas excepciones.

Es posible que un único bloque try tenga varios manejadores para poder gestionar distintos tipos de excepciones. Si se produce una excepción en ese bloque se buscará el manejador más adecuado para el tipo de dato de la excepción generada.

Los bloques try-catch pueden llevar también un bloque finally; en este caso el código de este último bloque se ejecutará tanto si el código del try se ha ejecutado correctamente, como si se ha lanzado una excepción y nos hemos ido al catch. Los bloques finally se emplean a menudo para ejecutar código que debe ejecutarse tanto por el camino de ejecución normal como excepcional. Por ejemplo, código que cierre un archivo que hemos abierto para escritura; tanto si se produce algún error al escribir como si no tendremos que cerrarlo.

```
try { // bloque de código en el cual se puede lanzar una
excepción
...
}
catch (Exception ex) { // bloque que maneja las posibles
excepciones
...
}
 finally{
...
}
```

Algunas excepciones, que se denominan uncheked, no tienen que gestionarse de modo necesario, y el compilador no nos obliga a gestionarlas, aunque sea posible que en nuestro código se produzcan excepciones de este tipo. Un ejemplo es la NullPointerException. Si queremos construir una excepción de este tipo haremos que nuestra excepción herede de RuntimeException. Hay otras excepciones, que se denominan cheked, que suelen representar problemas con el hardware que se considera que el programador debe siempre tener en mente cuando está construyendo su programa. Estas excepciones estamos obligados a gestionarlas; las unchecked podemos gestionarlas si queremos o si no podemos ignorarlas. Cuando invoquemos a código (constructores o métodos) que potencialmente pueden lanzar una excepción cheked debemos colocar ese código dentro de un bloque try-catch. Para construir una excepción checked haremos que nuestra excepción herede de Exception.

5.1 Ejemplo de excepciones: busca de raíces de una función

Veamos un ejemplo del uso de excepciones. Vamos a construir un programa que encuentre una raíz de una función empleando el método de las divisiones sucesivas. Para ello emplearemos un método estático de una clase que va a recibir con argumentos la función para la cual queremos encontrar la raíz y el principio y el fin del intervalo donde buscar la raíz. El principio y el fin primero deben verificar que el principio sea anterior al fin del intervalo, y que la función tiene signo diferente al principio y al final del intervalo. En caso de que no se cumplan estas condiciones lanzaremos una excepción del siguiente tipo:

```
package rootsolver;

public class NonValidIntervalException extends RuntimeException {

    public enum ErrorType {
        NOT_AN_INTERVAL, INTERVAL_NOT_VALID};

    private FunctionInterface function;
    private float start, end;
    private ErrorType error;

    NonValidIntervalException(FunctionInterface function, float start,
float end, ErrorType error) {
        this.function = function;
        this.start = start;
        this.end = end;
        this.error = error;
    }

    public String toString() {
        switch (getError()) {
            case NOT_AN_INTERVAL:
                return "No es un intervalo valido; el inicio es posterior
al final";

            case INTERVAL_NOT_VALID:
                return "No es un intervalo valido; la funcion al principio
vale "  + getFunction().evaluate(getStart()) + " y al final vale "  +
getFunction().evaluate(getEnd());
            default:
                return "Se produjo algún error";
        }
    }

    public FunctionInterface getFunction() {
        return function;
    }

    public float getStart() {
        return start;
    }
```

```
    public float getEnd() {
        return end;
    }

    public ErrorType getError() {
        return error;
    }
}
```

En la excepción hemos sobrescrito el método toString; éste es el método que se emplea para convertir un objeto Java a una cadena de caracteres cuando, por ejemplo, lo concatenamos con una cadena de caracteres. Observa además como hemos guardado en atributos de la clase toda la información que puede resultar útil a alguien que quiera gestionar esta excepción.

La clase que busca la raíz lo primero que hace es comprobar si el intervalo es sólo un intervalo válido, y en caso negativo lanza una excepción:

```
package rootsolver;

public class RootSolver {

    private static float tolerance = 0.001F;

    public static float solve(FunctionInterface function, float start,
float end)
            throws NonValidIntervalException {
        validateInterval(end, start, function);
        while (differentSign(function, start, end) && (end - start) >
getTolerance()) {

            float middle = (start + end) / 2;

            if (differentSign(function, start, middle)) {
                end = middle;
            } else if (differentSign(function, end, middle)) {
                start = middle;
            } else {
                return middle;
            }
        }

        return (start + end) / 2;

    }

    private static void validateInterval(float end, float start,
FunctionInterface function)
            throws NonValidIntervalException {
        if ((end - start) <= 0) {
```

```
            throw new NonValidIntervalException(function, start, end,
                    NonValidIntervalException.ErrorType.NOT_AN_INTERVAL);
        }

        if (!differentSign(function, start, end)) {
            throw new NonValidIntervalException(function, start, end,

NonValidIntervalException.ErrorType.INTERVAL_NOT_VALID);
        }
    }

    private static boolean differentSign(FunctionInterface function, float
i, float f) {
        return function.evaluate(i) * function.evaluate(f) < 0;
    }

    public static float getTolerance() {
        return tolerance;
    }

    public static void setTolerance(float aTolerance) {
        tolerance = aTolerance;
    }
}
```

Ambas clases trabajan con las funciones a través de una interfaz:

```
package rootsolver;

public interface FunctionInterface {

    float evaluate (float x);
}
```

En el paquete rootsolver.functions hay varias implementaciones de esta interfaz; aquí puedes ver una:

```
package rootsolver.functions;

import rootsolver.FunctionInterface;

public class ERisedToXMinusX implements FunctionInterface {

    private float a, b;

    public ERisedToXMinusX(float a, float b) {
        this.a = a;
        this.b = b;
```

```
    }

    @Override
    public float evaluate(float x) {
        return (float) (a * Math.exp(x) + b * x);
    }
}
```

Finalmente, vamos a ver un ejemplo de código fuente que emplea el resolvedor de funciones y gestiona las excepciones; la última llamada al resolvedor emplea un intervalo que no es válido; la función no tiene ninguna raíz en dicho intervalo. Por tanto se va a lanzar una excepción.

```
package example;

import rootsolver.NonValidIntervalException;
import rootsolver.RootSolver;
import rootsolver.functions.ERisedToXMinusX;
import rootsolver.functions.Linear;
import rootsolver.functions.Parabola;

public class Example {

    public static void main(String[] args) {
        float solution;
        try {
            solution = RootSolver.solve(new Linear(1, 1), -10, 10);
            System.out.println("La solucion es " + solution);

            solution = RootSolver.solve(new Parabola(1, 0, -4), 0, 10);
            System.out.println("La solucion es " + solution);

            solution = RootSolver.solve(new ERisedToXMinusX(1, 1), -10,
10);
            System.out.println("La solucion es " + solution);

            //No hay raiz en este intervalo
            solution = RootSolver.solve(new ERisedToXMinusX(1, 1), 1, 10);
            System.out.println("La solucion es " + solution);
        } catch (NonValidIntervalException ex) {

            System.out.println("Error: " + ex);
            ex.printStackTrace();
        }
    }
}
```

Y esto es lo que muestra el código anterior en consola:

```
La solucion es -1.000061
La solucion es 1.9998169
```

```
La solucion es -0.5673218
Error: No es un intervalo valido; la funcion al principio vale
3.7182817 y al final vale 22036.465
No es un intervalo valido; la funcion al principio vale 3.7182817 y
al final vale 22036.465
   at rootsolver.RootSolver.validateInterval(RootSolver.java:35)
   at rootsolver.RootSolver.solve(RootSolver.java:9)
   at example.Example.main(Example.java:24)
```

6 Ejercicios

1. Escribe un programa que cree una clase para representar un objeto punto en tres dimensiones. Proporcionar un constructor que inicialice los valores del punto al origen de coordenadas y otro que permita especificar las coordenadas del punto. Sobrescribe su método toString() para que muestre información sobre los puntos. Usa la clase en un programa donde crees objetos que representen los puntos (12, 13, 18) y (8, 14, 0) y los muestres por consola.
2. Crea una clase fecha que almacene el día, el mes y el año de una fecha. Proporciona funciones miembro para acceder a estos atributos (getDia(), getMes() y getAño() y para modificarlos (setDia(int dia), setMes(int mes) y setAño(int año)). Sobreescribe su método toString(). Crea la fecha 20/10/2018. Muéstrala por pantalla. Después cambia el año 2019. Muéstrala por pantalla.
3. Crear una clase que represente un número racional que permita, al menos, sumar, multiplicar y simplificar números racionales. Proporcionar un constructor por defecto, un constructor de copia (esto es, un constructor al que se le pasa una instancia de la clase número racional y crea otro número racional idéntico), y otro que permita indicar los valores del numerador y del denominador. Usando esta clase, crea una calculadora que permita operar con números racionales, seleccionando las operaciones de un menú.
4. Repetir el ejercicio anterior, pero creando una clase que represente a un número complejo.
5. Haz una clase llamada Persona con atributos: nombre, edad, DNI, sexo (usa una enumeración), peso y altura. Crea métodos para acceder y modificar todos los atributos.

Por defecto, todos los atributos menos el DNI tendrán valores por defecto según su tipo (0 números, cadena vacía para String, etc.). Sexo será mujer por defecto. La clase deberá tener los siguientes constructores constructores:

- Un constructor por defecto.
- Un constructor con el nombre, edad y sexo, el resto por defecto.
- Un constructor con todos los atributos como parámetro, menos el DNI.

La clase deberá tener los siguientes métodos:

- calcularIMC(): calcula el índice de masa corporal de la persona (peso en kg/(altura^2 en m))
- valorarPesoCorporal() devuelve un -1 si está por debajo de su peso ideal, un 0 si está en su peso ideal y un 1 si tiene sobrepeso. Sobrepeso se define como IMC > 25 y se considera que se está por debajo del peso ideal si IMC <18.
- esMayorDeEdad(): indica si es mayor de edad, devuelve un booleano.
- toString(): devuelve toda la información de la persona como una cadena de caracteres.
- generaDNI(): genera un numero aleatorio de 8 cifras que será el DNI de la persona. Este método no será visible desde el exterior. Este método deberá invocarse desde cualquier constructor para generar el DNI.
- Métodos set de cada parámetro, excepto de DNI.

Ahora, crea una clase ejecutable que haga lo siguiente:

- Pide por teclado el nombre, la edad, sexo, peso y altura.
- Crea 3 objetos de la clase anterior, el primer objeto obtendrá las anteriores variables pedidas por teclado, el segundo objeto obtendrá todos los anteriores menos el peso y la altura y el último por defecto, para este último utiliza los métodos set para darle a los atributos un valor.
- Para cada objeto, se deberá comprobar si está en su peso ideal, tiene sobrepeso o por debajo de su peso ideal con un mensaje.
- Indicar para cada objeto si es mayor de edad.
- Por último, mostrar la información de cada objeto.

6. Haz una clase llamada Password que tenga los atributos longitud y contraseña . Por defecto, la longitud será de 8. Los constructores serán los siguiente:

- Un constructor por defecto que tendrá como contraseña "password".
- Un constructor con la longitud que nosotros le pasemos. Generará una contraseña aleatoria con esa longitud.

Los métodos de esta clase serán:

- esFuerte(): devuelve un booleano si es fuerte o no, para que sea fuerte debe tener más de 2 mayúsculas, más de 1 minúscula y más de 5 números.
- generarPassword(): genera la contraseña del objeto con la longitud que tenga.
- Método get para contraseña y longitud.
- Método set para longitud.

Ahora, crea una clase clase ejecutable:

- Cree un array de Passwords con el tamaño que tú le indiques por teclado.
- Cree un bucle que cree un objeto para cada posición del array. Indica por teclado la longitud de cada password.
- Crea otro array de booleanos donde se almacene si el password del array de Password es o no fuerte (usa el bucle anterior).
- Al final, muestra la contraseña y si es o no fuerte (usa el bucle anterior). Usa este simple formato:

contraseña1 valor_booleano1

contraseña2 valor_bololeano2

7. Usando las clases del código de ejemplo de los marcianos, construye una guerra donde combatan 5 naves de los marcianos y 10 naves de los terrícolas.
8. Usando las clases del código de ejemplo del resolvedor de raíces, añade una nueva función que permita representar un polinomio de grado n. Crea esta función en tu propio paquete. Utilizando el resolvedor sin modificar ninguna línea de código, busca raíces de tu polinomio.
9. Crea un nuevo método en la clase resolvedor tal que uno pueda especificar una función, y no sea necesario indicar el intervalo inicial para buscar la raíz.

El propio método va a tratar de buscar un intervalo adecuado. Trata de idear una estrategia adecuada para encontrar ese intervalo.

10. En una empresa todos los trabajadores tienen un sueldo base de 1000 €. Los jefes tienen un suplemento de 500 € por cada año que hayan sido jefe de la empresa, y los viajantes además del sueldo base cobran 300 € por viaje realizado. Crear una clase empleado de la cual deriven las clases jefe y viajante. Crear una plantilla de una empresa con dos jefes, cinco viajantes y 15 empleados, e imprimir por consola sus respectivos salarios. Para generar el número de viajes de los viajantes y la antigüedad de los jefes puedes generar números aleatorios entre 0 y 10. Emplea el polimorfismo de herencia.
11. Crea la clase cuenta bancaria, que deberá tener como atributos un nombre de titular, una fecha de apertura, un número de cuenta y un saldo (puedes emplear un número real. La cuenta deberá tener un método para retirar dinero, otro para ingresar dinero y otro para transferir dinero a otra cuenta. De una cuenta no se podrá retirar nunca dinero (ni transferir) si la cantidad de dinero a retirar o transferir es mayor que el saldo. Crea una cuenta a plazo fijo, en la cual cuando se retira dinero de algún modo antes de una fecha de vencimiento (que será otro atributo de esta clase) además del dinero a retirar se penaliza con un 5% adicional. Crea además una cuenta Vip, que tendrá un atributo adicional que es el saldo negativo máximo que puede tener. En las cuentas Vip uno podrá tener saldo negativo siempre que no supere este valor. A continuación construye un main que permita crear los tres tipos de cuentas, y transferir dinero de unas a otras, ingresar dinero y retirar dinero. Almacena las cuentas en un array. Emplea polimorfismo de herencia.
12. Crea una clase fecha que almacene el día, el mes y el año de una fecha. Proporciona funciones miembro para acceder a estos atributos (getDia(), getMes() y getAño() y para modificarlos (setDia(int dia), setMes(int mes) y setAño(int año)). Sobreescribe su método toString(). La clase debe asegurarse de que los valores introducidos para sus miembros, tanto a través de los constructores como de los métodos modificadores, se corresponden con una fecha válida (no es necesario tener en cuenta años bisiestos). Para ello lanzará una excepción en caso de que los datos no sean válidos. Crea la

fecha 31/02/2015 y verifica que se lanza la excepción correspondiente. Verifica que esto también sucede al invocar el método setDia (35).

13. Escribir un resolvedor que busque raíces de una función empleando el algoritmo de Newton-Raphson para encontrar una raíz de una función concreta. Este método parte de una estimación inicial de la raíz, x0, y va calculando aproximaciones sucesivas al valor de la misma utilizando la fórmula:

$$x_{i+1}=x_i-f(x_i)/f'(x_i)$$

Al ser un método iterativo, es necesario tener un criterio para que termine. Puede establecerse como criterio de terminación que la diferencia entre f(x) y 0 sea menor que un valor e pequeño. Además, por seguridad, se debe establecer un número máximo de iteraciones para que el algoritmo termine aunque no converja a una solución (de lo contrario, al ejecutarlo en el ordenador éste podría quedarse bloqueado).