



POLIMORFISMO	2
Operador instanceof	10
INTERFACES	11
Definición de una interfaz	11
Implementación de una interfaz	12
EXCEPCIONES	16
Clases de excepción	16
Los bloques try···catch···finally	19
Lanzamiento de una excepción	21
Métodos para el control de una excepción	22
Clases de Excepción personalizadas	22
ALMACENAMIENTO DE OBJETOS	24
Contenedores	24
Taxonomía de los Contenedores	25
Contenedores de tipo List	25
Contenedores de tipo Set (Conjunto)	26
Contenedores de tipo Map	26
Iteradores	27
La Interfaz COMPARABLE	28
Sobreescritura de <i>equals()</i> y <i>hashCode()</i>	29
Implementación de la Interfaz COMPARATOR	29
Notas finales	30



POLIMORFISMO

Para iniciar el tema de polimorfismo comenzaremos por analizar un concepto clave para comprender este tercer pilar de la POO, que es el concepto de clase abstracta.

Una **clase abstracta** es una clase en la que **alguno de sus métodos está declarado pero no definido**, o sea, se **especifica su nombre, parámetros y tipo de devolución pero no tiene código**. Este tipo de métodos reciben el nombre de **métodos abstractos**. En este tipo de métodos se desconoce cómo será su implementación; **las subclases de esa clase abstracta serán las responsables de darla forma a ese método, sobreescribiéndolo**, en este caso definiendo el código propio de esa subclase.

Veamos un ejemplo: sabemos que toda figura tiene un método para calcular su área, pero si tuviéramos una clase Figura no sería posible codificar ese método ya que cada figura presenta un cálculo diferente para obtener su área. Lo mismo ocurre en el caso de querer obtener su perímetro o dibujarla, cada uno de esos métodos se implementan de distinta manera según el tipo de figura. En ese caso dichos métodos se definen como abstractos en la clase Figura, dejando a las subclases (Rectángulo, Círculo, etc.) la codificación o sea su implementación.

La sintaxis para la creación de una clase abstracta es la siguiente:

```
public abstract class nombre_clase
{
    public abstract tipo nombre_metodo(argumentos);
    //otros métodos
}

public abstract class Figura {
    public abstract Double calcularElArea();
    public abstract Double calcularElPerimetro();
    public abstract void dibujarFigura();
    :
}
```

Tanto **en la declaración de la clase como en la del método abstracto se debe utilizar el modificador *abstract***. En el ejemplo se puede ver cómo los **métodos abstractos** son métodos **sin cuerpo, su declaración termina en un ";" y dado que no poseen código no llevan llaves {}**. Existen algunas cuestiones importantes a tener en cuenta al crear clases abstractas:

- Una clase abstracta puede tener métodos no abstractos. **La única condición para que una clase se convierta en abstracta es que posea por lo menos un método abstracto, independientemente de poseer otros métodos y atributos.**



- **No es posible crear objetos a partir de una clase abstracta.** En caso de intentar hacerlo daría como resultado un error de compilación. Eso se debe a que el principal objetivo de la creación de una clase abstracta es obrar como base de las futuras subclases, que a partir de la herencia se encargarán de sobrescribir el método abstracto, permitiendo de esta manera la implementación de los mismos.
- **Las subclases de una clase abstracta están obligadas a sobrescribir los métodos abstractos que heredan.** En caso de que no interese sobrescribir alguno de esos métodos, la subclase deberá ser declarada también como abstracta. En el momento en que un método abstracto es sobrescrito por una subclase, la palabra *abstract* desaparece de la definición del método.
- **Una clase abstracta puede tener constructores.** A pesar de no ser posible crear objetos a partir de una clase abstracta, la misma puede tener constructores, **dado que son heredadas por otras clases que sí podrán crear objetos.** Recordemos que cuando se crea un objeto de una subclase se ejecuta también el constructor de la superclase. De hecho, si no se incluye un constructor explícitamente en la clase abstracta el compilador añadirá uno por defecto (como sucede con cualquier clase estándar)

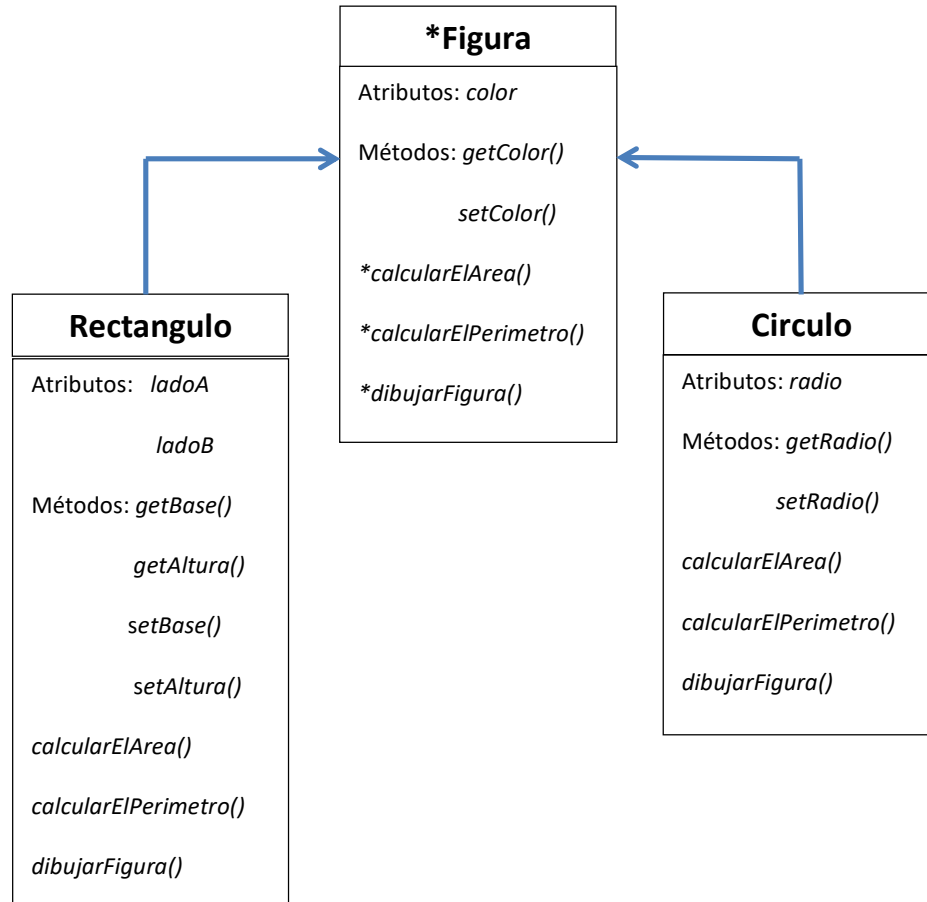
Veamos ahora en detalle el desarrollo de la clase abstracta *Figura*, con sus atributos: *color* en este caso, métodos abstractos: *calcularElArea()*, *calcularElPerimetro()* y *dibujarFigura()*; y métodos estándar *getter*, *setter* y *constructor*.

//Clase Figura

```
public abstract class Figura {  
  
    private String color;  
  
    public String getColor() {  
        return color;  
    }  
  
    public void setColor(String color) {  
        this.color = color;  
    }  
  
    public Figura(){  
        this.color = "Negro";  
    }  
  
    public abstract Double calcularElArea();  
    public abstract Double calcularElPerimetro();  
    public abstract void dibujarFigura();  
  
}
```



A partir de la clase abstracta *Figura* es posible implementar otras clases que sobrescriban los métodos abstractos, en nuestro caso las clases *Rectangulo* y *Circulo*. A continuación se muestra una representación gráfica de las clases previamente mencionadas y la relación entre las mismas:



A continuación se desarrolla el código completo de las clases *Rectangulo* y *Circulo*, con sus métodos abstractos *calcularElArea()*, *calcularElPerimetro()* y *dibujarFigura()* sobreescritos:

//Clase Rectangulo

```
public class Rectangulo extends Figura {

    private Double ladoA;

    private Double ladoB;

    public Rectangulo(Double ladoA, Double ladoB){

        this.ladoA = ladoA;

        this.ladoB = ladoB;

    }

}
```



```
@Override

public Double calcularElArea() {

    return (this.ladoA * this.ladoB);

}

@Override

public Double calcularElPerimetro() {

    return (this.ladoA*2) + (this.ladoB*2);

}

public Double getBase() {

    return ladoA;

}

public void setBase(Double base) {

    this.ladoA = base;

}

public Double getAltura() {

    return ladoB;

}

public void setAltura(Double altura) {

    this.ladoB = altura;

}

public void dibujarFigura(){

    System.out.println("Se dibuja un rectángulo
con un area de " + calcularElArea() + " y un perímetro de " +
calcularElPerimetro());

}

}
```



//Clase Circulo

```
import java.lang.Math;

public class Circulo extends Figura {

    private Double radio;

    public Circulo() {
        this.setRadio(new Double(0.00));
    }

    public Circulo(double radio) {
        this.setRadio(new Double(radio));
    }

    public Double getRadio() {
        return radio;
    }

    public void setRadio(Double radio) {
        this.radio = radio;
    }

    public Double calcularElArea() {
        Double area;
        area = (this.radio * this.radio * Math.PI);
        return area;
    }

    public Double calcularElPerimetro() {
        Double longitud;
        longitud = (2 * this.radio * Math.PI);

        return longitud;
    }
}
```



```
public void dibujarFigura(){  
    System.out.println("Se dibuja un circulo con  
un area de " + calcularElArea() + " y un perimetro de " +  
calcularElPerimetro());  
}  
}
```

De esta manera, estamos en condiciones de definir el concepto de Polimorfismo que, como se mencionara anteriormente, corresponde a uno de los tres pilares fundamentales de la POO. El polimorfismo es una de las principales aplicaciones de la herencia y el principal motivo de la existencia de las clases abstractas. Pero, antes de definir el polimorfismo es necesario detenernos en una circunstancia especial respecto de la asignación de objetos a variables.

En Java, es posible asignar un objeto de una clase a una variable de su superclase. Esto es aplicable incluso cuando la superclase es una variable abstracta.

Por ejemplo, dada una variable de tipo Figura:

```
Figura f;
```

Es posible asignar esta variable a un objeto Rectangulo:

```
f=new Rectangulo(...);
```

A partir de aquí puede utilizarse esta variable para invocar a aquellos métodos del objeto que también están definidos o declarados en la superclase, pero no aquellos que existan sólo en la clase a la que pertenece el objeto. Por ejemplo, puede utilizarse la variable *f* para invocar a los métodos *calcularElArea()*, *calcularElPerimetro()* y *getColor()* del objeto Rectangulo, pero no a los *getBase()* y *getAltura()* (por ser propios de la clase Rectangulo):

```
f.getColor(); //invoca a getColor() de Rectangulo  
f.calcularElArea(); //invoca al método de cálculo del área de Rectangulo  
f.getBase(); // error de compilación  
f.getAltura(), //error de compilación
```

De lo anterior se deduce que la misma instrucción *f.calcularElArea()* permite llamar a los métodos *calcularElArea()*, dependiendo del objeto almacenado en la variable *f*. En esto consiste el **polimorfismo**, o sea, en la **posibilidad de utilizar una misma expresión para invocar diferentes versiones de un mismo método**, determinando en tiempo de ejecución la versión del método a ejecutar.

La **principal ventaja del polimorfismo** es la **reutilización del código**, por el cual un objeto de la clase Rectangulo, un objeto de la clase Circulo y un objeto de la clase Triangulo pueden utilizar el mismo método, pero ajustado a las características de su subclase.



Recordemos que la sobreescritura de métodos de una clase implica que la nueva versión del método debe mantener el tipo de retorno definido por el método original. Sin embargo, a partir de la versión 5 de Java es posible modificar el tipo de retorno al sobreescribir un método, siempre y cuando el nuevo tipo sea un subtipo(subclase) del original. Por ejemplo, en la creación de la clase *Figura* se podría haber incluido un método abstracto *getNuevaFigura()* que devuelva una copia del objeto *Figura*:

```
abstract Figura getNuevaFigura();
```

Las subclases *Triangulo*, *Circulo* y *Rectangulo*, al heredar *Figura*, dispondrían también de este método y deberían sobreescribirlo. En el caso de la subclase *Circulo*, en versiones anteriores la sobreescritura podría haberse llevado a cabo de la siguientes forma:

```
public Figura getNuevaFigura(){  
    return new Circulo(radio, getColor());  
}
```

Suponiendo que *cir* es una variable que contiene una referencia al objeto *Circulo*, para obtener una copia del objeto *Circulo* deberíamos escribir:

```
Circulo cir2=Circulo(cir.getNuevaFigura());
```

De este modo, el método *getNuevaFigura()* devuelve un tipo *Figura* y es necesario realizar una conversión explícita al subtipo *Figura* con el que se está trabajando. Sin embargo, a partir de Java 5 el método *getNuevaFigura()* puede sobreescribirse en la clase *Circulo* de la siguiente manera:

```
public Circulo getNuevaFigura(){  
    return new Circulo(radio, getColor());  
}
```

Esto elimina la necesidad de realizar conversiones explícitas a la hora de obtener copias de objetos, como en este ejemplo:

```
Circulo cir2=cir.getNuevaFigura(); //Java 5
```

El uso de **polimorfismo** es una muy buena práctica de desarrollo para reutilizar funcionalidad, hacer el código escalable y de esta forma aprovechar las características de la programación orientada a objetos; pero a su vez **nos permite hacer uso del Casting**. El casteo o Casting es un **procedimiento para transformar una variable primitiva de un tipo a otro, o transformar un objeto de una clase a otra clase siempre y cuando haya una relación de herencia entre ambas**. Veamos un ejemplo

```
Figura figuraNueva=new Rectangulo(40.0,20.0);  
figuraNueva.calcularElArea();  
(Rectangulo)figuraNueva.setLadoA(5);  
figuraNueva.calcularElArea(); //100
```

Casteo o Casting

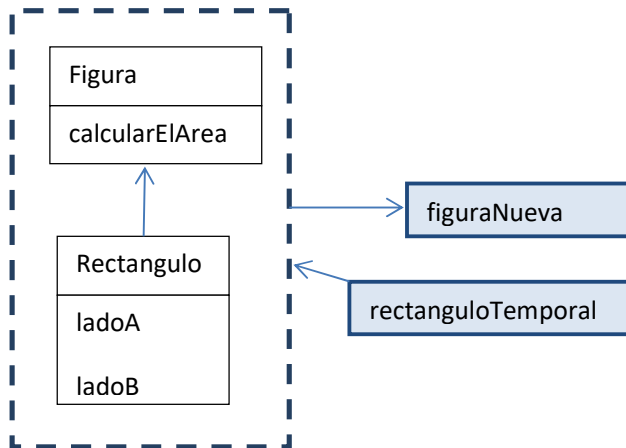


En la tercera línea se puede observar en `((Rectangulo)figuraNueva)` un casteo que avisa al compilador que `figuraNueva` es de tipo `Rectangulo`. Si se cambiara el tipo de figura, por ejemplo de `Rectangulo` a `Circulo` (o sea se cambia el casteo) daría un error en tiempo de ejecución, se produciría una excepción (concepto que se explicará más adelante). Otra forma de resolver la misma situación es la siguiente:

```
Rectangulo rectanguloTemporal=(Rectangulo) figuraNueva;  
rectanguloTemporal.setLadoA(5);  
figuraNueva.calcularElArea(); //100
```

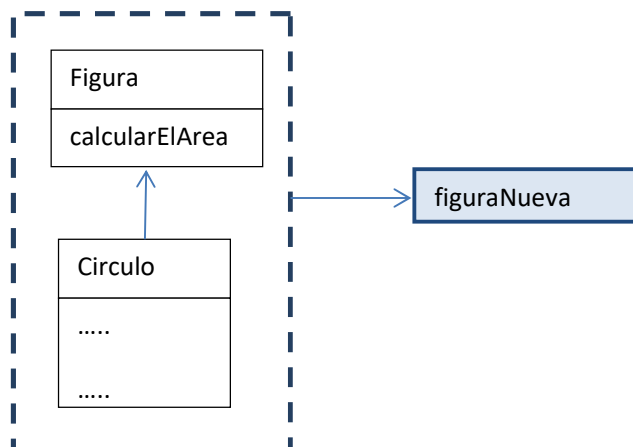
Casteo o Casting

En términos de eficiencia parecería que la primera opción de resolución es más eficiente que la segunda, dado que utiliza menos líneas de código; sin embargo, a nivel de bytecode ambas representan la misma eficiencia dado que en el primer caso se crea una variable anónima a nivel de la memoria.



En caso de modificar la figura, a nivel de memoria se vería así:

```
Figura figuraNueva=new Circulo(10.0);
```





Operador **instanceOf**

El operador **instanceOf** se utiliza para saber si un objeto pertenece a un determinado tipo e indica la relación “es una instancia de”. La forma de utilizarlo es:

```
referencia_Objeto instanceof Clase
```

Si el objeto pertenece a la referencia especificada o a una de sus subclases, **instanceOf** devuelve true, como en este ejemplo:

```
String s="Hola"

if (s instanceof String){

    System.out.println("Es una cadena");

}
```

En este caso mostrará por pantalla: Es una cadena. Otros ejemplos del uso de **instanceOf**:

```
public gestionarFigura(Figura figuraActual){


    If(figuraActual instanceof Triangulo){

    ...

public boolean equals(Object elQueQuieras)

    If(elQueQuieras instanceof Alumno){

    ...
```



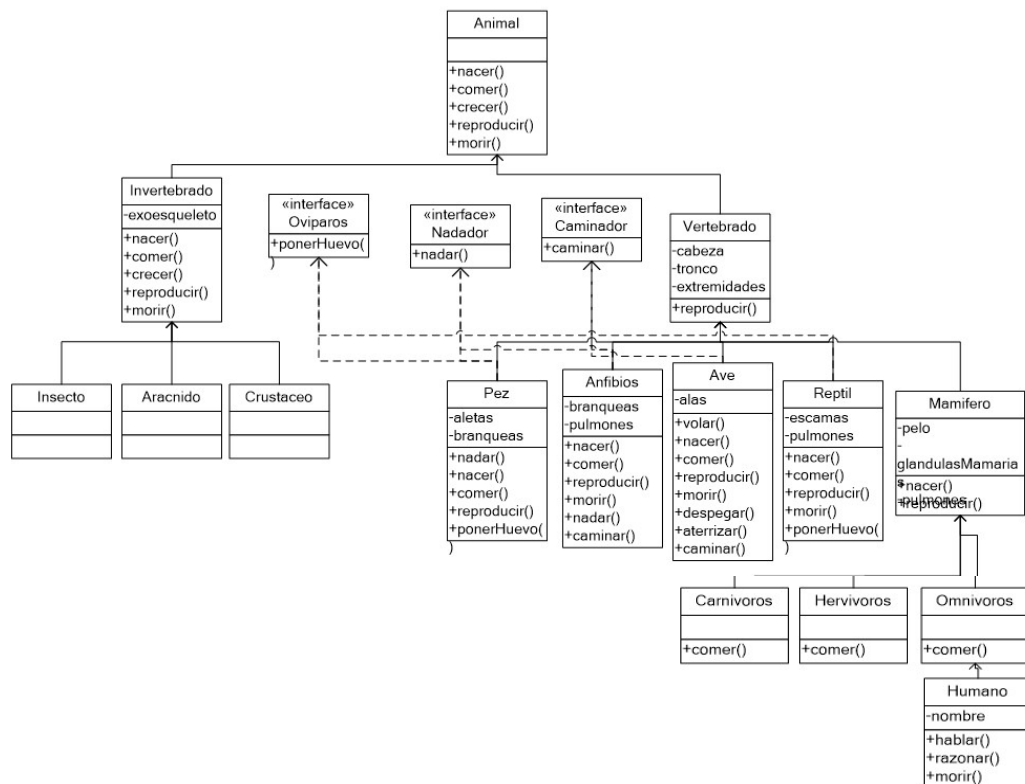
es una instancia de



INTERFACES

Como introducción al tema, podría decirse que las interfaces y las clases internas proporcionan formas más sofisticadas de organizar y controlar los objetos de un sistema. Una interfaz es un conjunto de métodos abstractos y de constantes públicas definidos en un archivo .java. Es similar a una clase abstracta, pero la diferencia de ella es que todos sus métodos son abstractos, pero a su vez, es más que una clase abstracta llevada al extremo, pues permite llevar a cabo una variación de la "herencia múltiple" que no es posible implementar de otra forma en Java. La finalidad de las interfaces es la de definir el formato de ciertos métodos que han de implementar determinadas clases, actúa como un patrón de diseño, como un contrato. La interfaz no determina qué es lo que el método tiene que hacer y cómo hacerlo, sino el formato (nombre, parámetros y tipo de devolución) que éste debe tener.

Una interfaz también puede contener campos, pero éstos son implícitamente estáticos y constantes. Una interfaz proporciona sólo la forma, pero no la implementación. Veamos el siguiente diagrama UML con la definición de interfaces y clases que las implementan.



Definición de una interfaz

Para crear una interfaz, se usa la palabra clave `interface` en vez de la palabra clave `class`. Por tanto, una interfaz se define mediante la palabra `interface`, utilizando la siguiente sintaxis:

```
public interface Nombre_interfaz{  
    tipo método1(argumentos);  
}
```



```
        tipo método2(argumentos);  
    }
```

Al igual que las clases, las interfaces se definen en archivos .java y **si la interfaz utiliza el modificador de acceso public el nombre de la interfaz deberá coincidir con el nombre del fichero .java donde se almacena**. Como resultado de la compilación de una interfaz, se genera un archivo .class. En el siguiente ejemplo, vemos el desarrollo de distintas interfaces del diagrama UML de la página anterior:

```
public interface Nadador {  
  
    static final boolean branquias=true;  
  
    public void nadar();  
  
}
```

Al crear una interfaz hay que tener en cuenta lo siguiente:

- **Todos los métodos definidos en una interfaz son públicos y abstractos, aunque no se lo indique explícitamente.** El uso de los modificadores abstract y public resulta entonces redundante, sin embargo su uso explícito no provoca errores.
- En una interfaz, **es posible definir constantes**. Además de los métodos, las interfaces pueden contener constantes de **tipo público y estático**. Los modificadores correspondientes a estas dos características pueden omitirse en la declaración de una constante en una interfaz. Como puede verse en el ejemplo anterior:

```
static final boolean branquias=true;
```

- **Una interfaz no es una clase, por lo cual no pueden contener métodos con código, constructores o variables, y obviamente, no es posible crear objetos a partir de ellas.**
- **Cuando una clase implementa una interfaz, está obligada a definir el código de todos los métodos existentes en la misma. De no implementar todos los métodos y dejar uno sin codificar la clase será declarada como abstracta.**

Implementación de una interfaz

Para hacer una clase que se ajuste a una interfaz particular (o a un grupo de interfaces), se usa la **palabra clave implements**. De alguna manera es como si se dijera: "La interfaz contiene la apariencia, pero a través de su implementación se dirá cómo funciona". En lo demás, se asemeja a la herencia. Para hacer que una clase defina el código para los métodos declarados en una interfaz, la clase deberá implementar la interfaz de la siguiente manera:

```
public class MiClase implements MiInterfaz{  
  
    ...  
  
}
```



Por otra parte, una clase puede heredar otra clase e implementar al mismo tiempo una o varias interfaces. Esto otorga una gran flexibilidad con respecto a las clases abstractas, ya que el implementar una interfaz no impide que la clase pueda heredar características de otras clases. La sintaxis utilizada para heredar una clase e implementar interfaces es:

```
public class MiClase extends SuperClase
    implements Interfaz1, Interfaz2,...{
    ...
}
```

Observemos de qué manera se implementa una interfaz en el ejemplo del diagrama UML visto anteriormente (también puede observarse en este caso que la clase *Pez* hereda de su clase padre *Vertebrado*, sobrescribiendo sus métodos):

```
public class Pez extends Vertebrado implements Nadador {
    private String aletas;
    private String branquias;

    public Pez(String cabeza, String tronco, String extremidades) {
        super(cabeza, tronco, extremidades);
    }

    public Pez(String cabeza, String tronco, String extremidades,
        String branquias, String aletas) {
        super(cabeza, tronco, extremidades);
        this.aletas = aletas;
        this.branquias = branquias;
    }

    public void nadar(){
        System.out.println("Mover las aletas para desplazarse de
        un lugar a otro");
    }

    public void nacer(){
        System.out.println("Romper el huevo y salir al mundo");
    }

    public void comer(){
        System.out.println("A través de la boca, comer algas y
        otros peces");
    }
}
```



```
public void reproducir(Pez otroPez){

    System.out.println(this.toString() + "copula con" +
        otroPez + "y tienen pececitos");

    otroPez.ponerHuevo();

}

public void ponerHuevo(){

    System.out.println("Poner huevo. Luego de unos meses nace
        un nuevo pez");

}

public String toString(){

    return ("Pez con cabeza " + super.getCabeza() + " tronco "
        + super.getTronco() + ", extremidades "
        + super.getExtremidades() + " branquias " + branquias + " y
        aletas: " + aletas);

}

}
```

Una clase puede implementar más de una interfaz, para lo cual deberá implementar todos los métodos existentes en todas las interfaces.

```
public class MiClase implements Interfaz1, Interfaz2...{

:

}
```

Siempre siguiendo la línea del conjunto de animales, podemos observar cómo los anfibios implementan dos interfaces al mismo tiempo, ya que estos animales son capaces tanto de nadar (implementando la interfaz *Nadador*, ya vista) como de caminar (implementando la interfaz *Caminador*, descripta a continuación).

```
public interface Caminador {

    public void caminar();

}
```

He aquí el ejemplo de la clase Anfibio en forma completa:

```
public class Anfibio extends Vertebrado implements Caminador,
    Nadador {

    public Anfibio(String cabeza, String tronco, String
        extremidades){

        super(cabeza, tronco, extremidades);

    }

}
```



```
}

public void nacer() {
    System.out.println("Anfibio. Nacer");
}

public void nadar(){
    System.out.println("Anfibio. Nadar");
}

public void caminar(){
    System.out.println("Anfibio. Caminar");
}

@Override
public void comer() {
    System.out.println("Anfibio. Comer");
}

}
```

Una interfaz también puede heredar otras interfaces, aunque en realidad no se trata de una herencia en sentido estricto ya que lo único que adquiere la subinterfaz son los métodos abstractos existentes en la superinterfaz. La sintaxis usada es la siguiente:

```
public interface MiInterfaz extends Interfaz1, Interfaz2, ...{
    ...
}
```

Por todo lo dicho, el principal objetivo que persiguen las interfaces es la definición de un formato común de los métodos, cumpliendo de esta manera con el principio de polimorfismo. Una variable de tipo interfaz puede almacenar cualquier objeto de las clases que la implementan, pudiendo utilizar esta variable para invocar a los métodos de los objetos que han sido declarados en la interfaz e implementados en la clase.

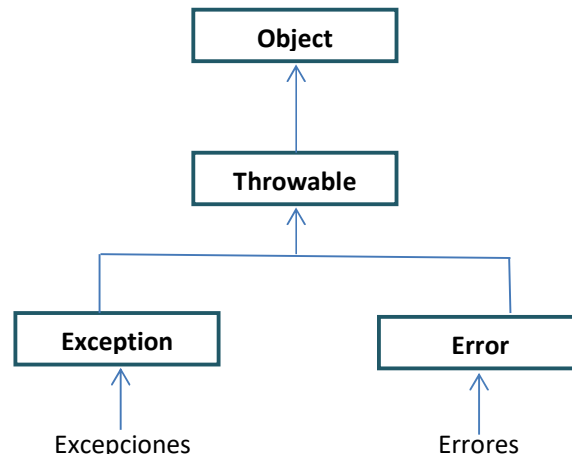


EXCEPCIONES

El momento ideal para capturar un error es en tiempo de compilación, antes incluso de intentar ejecutar el programa. Sin embargo, no todos los errores son detectados en tiempo de compilación, siendo necesario manejarlos en tiempo de ejecución, mediante alguna formalidad que pueda hacerse cargo de manipular la dificultad de manera adecuada. Mediante la captura de excepciones, Java proporciona un mecanismo que permite al programa sobreponerse a estas dificultades, de modo que el programador pueda decidir qué acciones llevar a cabo para cada tipo de excepción que pueda ocurrir. Se considera excepción a una situación anómala, tal como el intento de una división por cero, un acceso a posiciones por afuera de los límites de un array, etc. Si bien en algunos lenguajes de programación no es necesario hacer manejo de excepciones, Java obliga a hacerlo.

Además de las excepciones, en un programa desarrollado en Java pueden producirse otro tipo de errores, como un fallo de la máquina virtual o memoria llena, situaciones que están más allá del control del programador. Estos errores no son recuperables, o sea no es posible volver atrás.

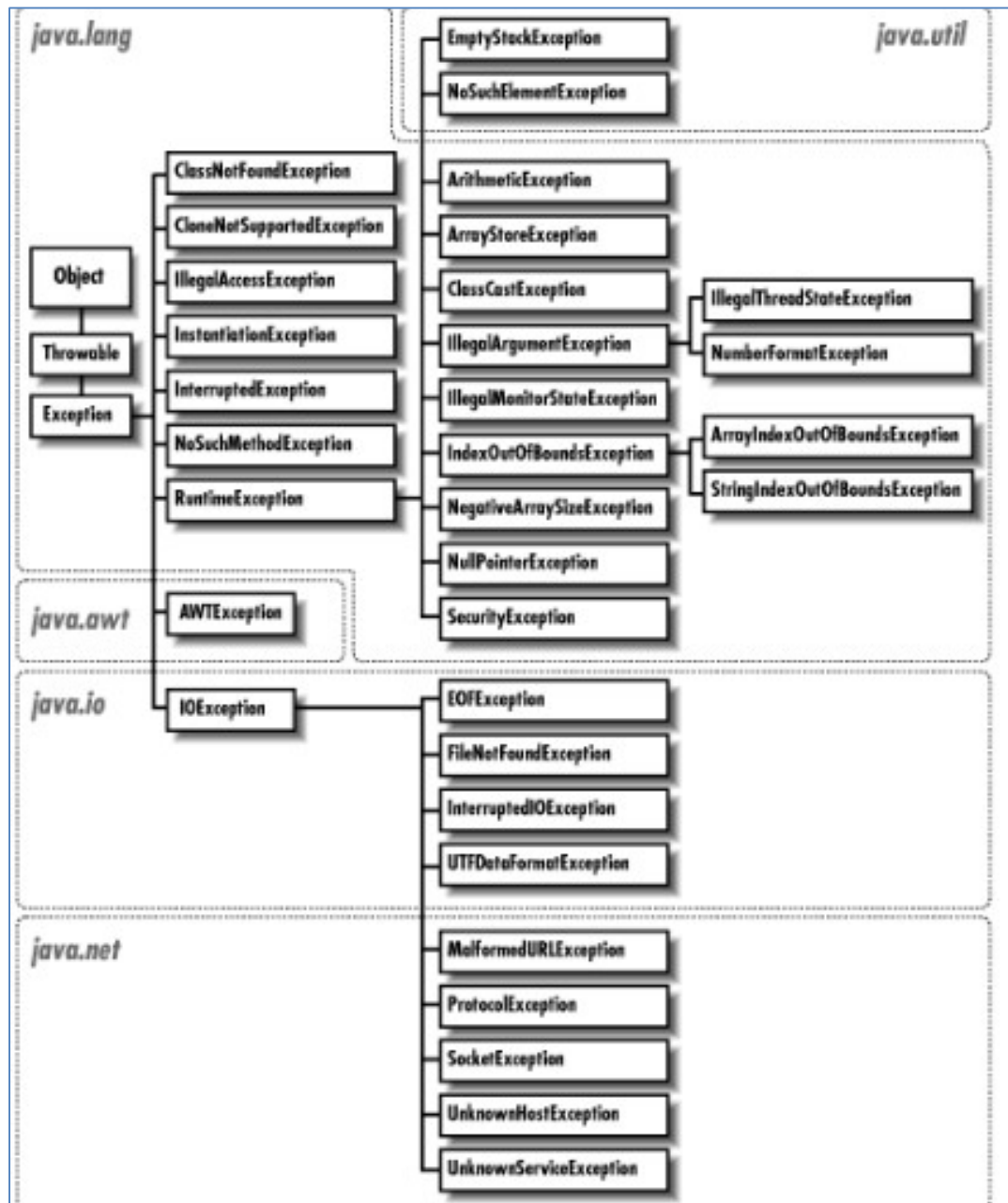
Cada tipo de error está representado por una subclase **Exception**, mientras que los errores son subclases de la clase **Error**. Ambas clases son subclases de la clase **Throwable**.



Cuando se lanza una excepción, ocurren varias cosas. En primer lugar, se crea el objeto excepción de la misma forma en que se crea un objeto Java: con `new`. Después, se detiene el cauce normal de ejecución (el que no se podría continuar) y se lanza la referencia al objeto excepción desde el contexto actual. En este momento se interpone el mecanismo de gestión de excepciones que busca un lugar apropiado en el que continuar ejecutando el programa. Este lugar apropiado es el gestor de excepciones, cuyo trabajo es recuperarse del problema de forma que el programa pueda, o bien intentarlo de nuevo, o bien simplemente continuar.

Clases de excepción

Al producirse una excepción en un programa, como dijimos se crea un objeto de la subclase **Exception** a la que pertenece la excepción. Este objeto será luego utilizado por el programa durante el tratamiento de la excepción. Veamos la distribución de las clases de excepciones dentro del siguiente esquema:



Dentro de la clase **Exception** pueden encontrarse diversas subclase como **IOException**, **SQLException**, **RuntimeException**, etc. Dentro de esta última podemos encontrar clases derivadas como **ArithmeticException**, **NullPointerException**, **IndexOutOfBoundsException**, **ClassCastException**, etc. Desde el punto de vista del tratamiento de una excepción dentro de un programa pueden definirse dos grandes tipos:

- Excepciones verificadas o marcadas
- Excepciones no verificadas o no marcadas



a) Las excepciones verificadas o marcadas son aquellas cuya captura es obligatoria y se producen al invocar métodos de determinadas clases y son lanzadas desde el interior de dichos métodos en caso de fallo de ejecución de los mismos. Todas las clases de excepciones, excepto ***RuntimeException*** y sus subclases pertenecen a este grupo. Un ejemplo de excepción marcada es ***IOException***. Esta excepción es lanzada por el método ***readLine()*** de la clase ***BufferedReader***, cuando se produce un error en la operación de lectura que obliga al programa a capturar la excepción. Si en un bloque de código se invoca a algún método que puede generar una excepción marcada y ésta no se captura, el programa no compilará. O sea, que cada vez que usemos un método que tenga declaradas excepciones, es necesario capturar esas excepciones.

Para declarar una excepción se utiliza la palabra reservada ***throws***, seguida de la lista de excepciones que el método puede provocar

```
public String readLine() throws IOException

public void service throws ServletException, IOException
```

b) Las excepciones no verificadas o no marcadas son aquellas excepciones en tiempo de ejecución, o sea ***RuntimeException*** y todas sus subclases. No es obligatorio capturar una excepción no marcada, dado que la gran mayoría es fruto de una mala programación. Las excepciones de este tipo que se recomienda capturar son las de tipo ***ArithmeticException***. Otro caso de este tipo de excepciones son las correspondientes a errores de casteo (***ClassCastException***). Veamos un ejemplo donde se verifica este tipo de excepción:

```
public class TestClassCastException {

    @Test

    public void testClassCastException(){

        Animal unVertebrado = TestNaturaleza.crearAnimales();

        String cabeza = ((Vertebrado)unVertebrado).getCabeza();

        System.out.println("La cabeza del vertebrado creado es:" +
        cabeza);

    }

}
```

En este ejemplo, la excepción tira un error de casteo cuando el animal no es un vertebrado, pero no ocurre un tratamiento especial del error, ya que es una excepción no verificada o no marcada (perteneciente al grupo de las excepciones que derivan de la clase ***RunTime Exception***).

En el momento en que se produce una excepción en un programa, se crea un objeto de la clase de excepción correspondiente y se “lanza” a la línea de código donde ocurrió la excepción. Esto permite delinear las diferentes acciones a realizar según la clase de excepción producida. En el caso de los errores, como representan fallos del sistema, si bien se pueden capturar al igual que las excepciones, es recomendable no hacerlo.



Los bloques **try...catch...finally**

Ahora bien, veamos de qué manera se manipula una excepción. Para ello analicemos el siguiente ejemplo:

```
public class TestLanzarExcepcion {

    private void mostrarContenidoDelZoologico(Animal zoologico[])
    throws NullPointerException, IOException{

        for(Animal a:zoologico){

            if(a==null){

                throw new IOException();

            }

            System.out.println(a.toString());

        }

    }

    @Test

    public void testLanzarExcepcion(){

        //Lanzando la Excepción

        Animal zoologicoDeLujan[];

        zoologicoDeLujan = TestNaturaleza.ingresarAnimales();

        System.out.println("Los animales en este zoologico son:");

        try{

            mostrarContenidoDelZoologico(zoologicoDeLujan);

        } catch(IOException E){

            System.out.println("**** El zoologico no está
            completo ****");

        }

    }

}
```

Los bloques **try...catch...finally** constituyen la forma de captar excepciones de programas en Java. El bloque **try** delimita las instrucciones donde puede producirse una excepción;

```
try{

    mostrarContenidoDelZoologico(zoologicoDeLujan);
```



luego el control del programa se transfiere al bloque *catch* que corresponde al tipo de excepción producida (en la cual se pasa como parámetro la excepción lanzada).

```
} catch(IOException E){  
  
    System.out.println("**** El zoologico no está  
    completo ****");  
  
}
```

En forma opcional se puede incluir el bloque *finally* en el que se definen un grupo de instrucciones que se ejecutan obligatoriamente (como por ejemplo cerrar archivos abiertos, o conexiones a base de datos, etc, la idea es mantener el sistema estable). En este ejemplo no se ha incluido el bloque *finally*.

Un bloque *catch* define las instrucciones que deben ejecutarse en caso en que se produzca una determinada excepción. A propósito del bloque *catch* deben tenerse en cuenta las siguientes consideraciones:

- Se pueden definir los bloques *catch* que sean necesarios, cada uno servirá para tratar un tipo diferentes de excepción. **No pueden existir dos bloques *catch* con la misma clase de excepción.**
- Un bloque *catch* sirve para capturar cualquier excepción que se corresponda con el tipo declarado o cualquiera de sus subclases. El bloque *catch*:

```
catch(RuntimeException e){  
    :  
}
```

se ejecutaría el producirse cualquier excepción de tipo *NullPointerException*, *ArithmeticException*, etc. De esta manera, una excepción podría ser tratada en diferentes bloques *catch*, por ejemplo la excepción *NullPointerException* podría ser tratada en un *catch* que capture esa excepción y en uno que capture *RuntimeException*.
- Aunque se incluyan varios bloques *catch*, sólo uno de ellos será captado cuando éste se produzca. La ejecución de los distintos bloques *catch* se realiza en forma secuencial; una vez terminada la ejecución del mismo, el control del programa se transfiere al bloque *finally* o a la instrucción siguiente al último bloque *catch*, si la instrucción *finally* no existiese. Tras la ejecución de un bloque *catch*, **el control del programa nunca vuelve al lugar donde se ha producido la excepción.**
- En el caso en que existan varios *catch* cuyas excepciones están relacionadas por la herencia, los *catch* más específicos deben preceder a los más generales. En caso de no respetar este orden se produce un error de compilación dado que los bloques *catch* más específicos nunca se ejecutarían.
- Si se produce una excepción no marcada para la que no se ha definido un bloque *catch*, esta será propagada por la pila de llamadas hasta encontrar algún punto donde se trate la excepción. En caso de no existir un tratamiento para la misma, la Máquina Virtual cancelará la ejecución del programa.



- Los bloques *catch* son opcionales en caso de que exista un bloque *finally*, pero si no existe un bloque *finally*, entonces es obligatorio disponer por lo menos, de un bloque *catch*.

Como se ha mencionado anteriormente su uso es opcional. Si existe, el bloque *finally* es ejecutado tanto si se produce una excepción o no. Si se produce una excepción, el bloque *finally* se ejecuta después del *catch* para el tratamiento de la excepción. En caso en que no hubiese ningún bloque *catch* para el tratamiento de la excepción, el bloque *finally* se ejecutaría antes de propagar la excepción. Si no se produce excepción dentro del bloque *try*, el bloque *finally* se ejecuta tras la última instrucción del *try*. **Aún existiendo una instrucción para la salida del método (por ejemplo un *return*), el bloque *finally* se ejecutará antes que esto suceda.**

Lanzamiento de una excepción

En algunos casos puede resultar útil generar y lanzar una excepción desde un determinado método, para enviar un aviso a otra parte del programa, indicándole que algo está sucediendo y que no es posible continuar con la normal ejecución del método. Para lanzar una excepción desde código se utiliza la expresión:

```
throw objeto_excepcion;
```

donde *objeto_excepción* es un objeto de alguna subclase de *Excepción*. En nuestro ejemplo anterior:

```
private void mostrarContenidoDelZoologico(Animal zoologico[]) throws
NullPointerException, IOException{
    for (Animal a:zoologico){
        if(a==null){
            throw new IOException();
        }
        System.out.println(a.toString());
    }
}
```

Cuando se lanza una excepción marcada o verificada desde un método esta debe ser declarada en la cabecera del método y debe ser tratada, dado que el compilador así lo exige. Si se trata de una excepción no marcada o no verificada (la clase *RuntimeException* y sus subclases) puede no darse tratamiento a la excepción, o hacerlo desde otro método. Si durante la ejecución de un programa se produce una excepción y no es capturada, la Máquina Virtual provoca la finalización del programa, enviando a la consola la pila con los datos de la excepción. Gracias a los volcados de pila el programador puede detectar fallos de programación.



Métodos para el control de una excepción

Todas las clases de excepción heredan una serie de métodos de la clase `Throwable` y pueden ser utilizados en el interior de los bloques `catch` para completar las acciones de tratamiento de la excepción. Los métodos principales son:

- `String getMessage()`: Devuelve un mensaje de texto asociado a la excepción, dependiendo del tipo de objeto de excepción sobre el que se aplique.
- `void printStackTrace()`: Envía a la consola el volcado de la pila asociado a la excepción. Es muy útil en la fase de desarrollo de la aplicación para ayudar a detectar errores de programación causantes de excepciones.
- `void printStackTrace(PrintStream s)`: Permite enviar el volcado de la pila aun objeto `PrintStream` cualquiera, por ejemplo un fichero `log`.

Veamos la siguiente creación de excepciones en el ámbito de la interface `Animal`: con la instrucción **`e.printStackTrace()`**; lo que se está imprimiendo es la traza de la pila de ejecución asociada a la excepción. Si es necesario, se puede mostrar alguna información adicional.

```
public void morir() {  
    try {  
        this.finalize();  
    } catch (Throwable e) {  
        // TODO Auto-generated catch block  
        e.printStackTrace();  
    }  
}
```

Clases de Excepción personalizadas

Cuando se necesita lanzar una excepción para notificar una situación anómala, puede ocurrir que las clases de excepción existentes no se adaptan a las características de la situación que se quiere informar. En estos casos se puede definir una clase de excepción personalizada, subclase de *Exception*, que se ajuste más a las características de la excepción que se piensa tratar. Veamos la siguiente creación de excepción personalizada:

```
public class ExceptionProgramacionBasica2 extends Exception {  
    public ExceptionProgramacionBasica2() {}  
  
    public ExceptionProgramacionBasica2(String msg) {  
        super(msg);  
    }  
}
```

Y su clase de prueba correspondiente:



```
public class TestExcepcionPropia {

    public void f() throws ExceptionProgramacionBasica2{

        System.out.println("Lanzando excepción sencilla");

        //throw new ExceptionProgramacionBasica2("No es un error,
        sino una prueba");

        throw new ExceptionProgramacionBasica2();

    }

    @Test

    public void testManejoExcepcion(){

        TestExcepcionPropia exc = new TestExcepcionPropia();

        try{

            exc.f();

        }catch(ExceptionProgramacionBasica2 e){

            e.printStackTrace();

            System.err.println("Esta es una prueba del lanzamiento de
            una excepción");

        }finally{

            // Bloque de código que se ejecuta siempre.

        }

    }

}
```



ALMACENAMIENTO DE OBJETOS

Existen distintas formas de almacenar objetos: a través de *arrays*, como ya se ha visto y a través de contenedores, que son clases que permiten almacenar objetos sin limitación de tamaño, ya que en estas clases se va recalculando el tamaño a medida que se van agregando los elementos. Hay dos aspectos que distinguen a los *arrays* de los contenedores: la eficiencia y el tipo. El array es la forma más eficiente que proporciona Java para almacenar ya que permite acceder al azar a una secuencia de objetos; por otra parte en los *arrays* se define el tipo de dato que se guarda; esto es distintivo de los *arrays* en Java, a diferencia de las colecciones que no necesitan definir el tipo. Sin embargo, a partir de Java 2.0 las colecciones también se tipifican, con lo cual esta característica distintiva de los *arrays* se desdibuja.

Las otras clases contenedoras genéricas *List*, *Set* y *Map*, manipulan los objetos como si no tuvieran tipo específico (un mismo contenedor puede almacenar elementos de cualquier tipo) dado que los tratan como de tipo **Object**, la clase raíz de todas las clases de Java. De esta manera es necesario construir sólo un contenedor, y cualquier objeto Java entrará en ese contenedor.

Por tanto, la primera y más eficiente selección a la hora de mantener un grupo de objetos debería ser un *array*, especialmente si lo que se desea guardar es un conjunto de datos primitivos. Pero, en el caso en el que no se sabe en el momento de escribir el programa cuántos objetos se necesitarán o si se necesitará una forma más sofisticada de almacenamiento de los objetos es necesario recurrir a la biblioteca de clases contenedoras para solucionar este problema, en la que destacan los tipos básicos *List*, *Set* y *Map*.

Contenedores

Los contenedores en Java 2 se ocupan de "almacenar objetos" y lo divide en dos conjuntos distintos:

1. **Colección** (Collection): grupo de elementos individuales, a los que generalmente se aplica alguna regla. Una lista (List) debe contener elementos en una secuencia concreta, y un conjunto (Set) no puede tener elementos duplicados.
2. **Mapa** (Map): grupo de pares de objetos clave-valor (por ejemplo clave o id: país; valor o elemento: capital). Los Mapas, al igual que los *arrays* pueden extenderse de manera sencilla a múltiples dimensiones sin añadir nuevos conceptos: simplemente se construye un Mapa cuyos valores son Mapas (y el valor de esos Mapas pueden ser Mapas, etc.).

La categoría Colección sólo mantiene un elemento en cada posición e incluye la Lista, que guarda un conjunto de elementos en una secuencia específica, donde los elementos se encuentran encadenados entre sí, que puede contener elementos de cualquier tipo y el Conjunto o Set, que sólo permite la inserción de un elemento de cada tipo con control de duplicidad (o sea, no se pueden ingresar elementos duplicados). La lista de Arrays (*ArrayList*) es un tipo de Lista, y el conjunto *HashSet* es un tipo de Conjunto. Para añadir elementos a cualquier Colección, se utiliza el método **add()**.

El Mapa guarda pares de valores clave, de manera análoga a una mini base de datos. Para añadir elementos a un Mapa se utiliza el método **put()** que toma una clave y un valor



como argumentos. Los métodos sobrecargados *fill()* rellenan Colecciones y Mapas respectivamente. Un uso habitual de *Map* podría constituir un archivo de configuración de determinadas propiedades y su estado.

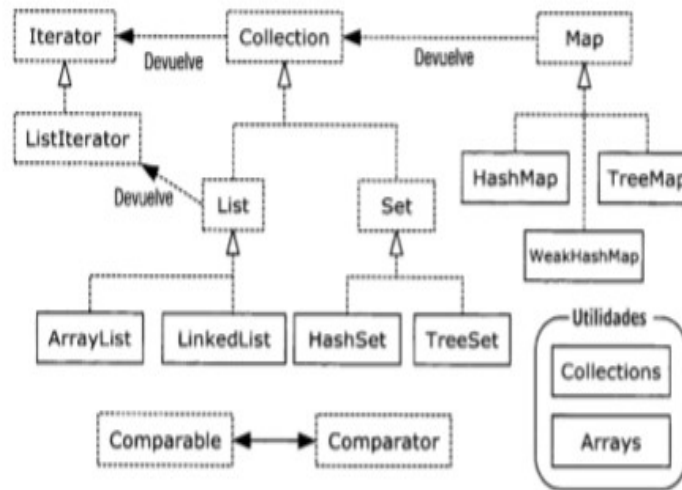
Una Colección siempre se indica entre corchetes, separando cada elemento por comas. Un *Mapa* se escribe entre llaves, con cada clave y valor asociados mediante un signo igual (claves a la izquierda, valores a la derecha). Por ejemplo:

[perro, perro, gato] // Ejemplo de Lista

[gato, perro] //Ejemplo de Conjunto o Set

{gato=Rags, perro=Spot} //Ejemplo de Mapa

Taxonomía de los Contenedores



En este diagrama puede observarse que realmente sólo hay tres componentes contenedores: *Map*, *List* y *Set*, y sólo hay dos o tres implementaciones de cada uno (habiendo una versión preferida, generalmente).

Contenedores de tipo List

Dentro del contenedor *List*, pueden observarse *ArrayList* y *LinkedList*, que son contenedores de tipo *List* que difieren entre sí en la forma en cómo agregan, buscan y eliminan sus elementos.

El *ArrayList* es similar a un array: se crea, se introducen objetos usando *add()* y posteriormente se extraen con *get()* haciendo uso del índice. *ArrayList* también tiene un método *size()* que permite saber cuántos elementos se han añadido evitando así generar una excepción. Es recomendado para el acceso rápido a elementos, pero es lento si se desea insertar o eliminar elementos del medio de una lista. Veamos un ejemplo:

```
import java.util.Collection;

import java.util.List;
```



```
public class ManejoContenedores {  
  
    public Collection rellenar(Collection c){  
  
        c.add("perro");  
  
        c.add("gato");  
  
        c.add("perro");  
  
  
        return c;  
  
    }  
}
```

En el ejemplo anterior, el elemento “perro” se encuentra repetido; para evitar que quede guardado en la colección habría que sobrescribir los métodos ***equals()*** y ***toString()*** (la sobrescritura de estos métodos constituye una buena práctica de programación, en particular ***toString()*** es un método de todo objeto no primitivo que es invocado en situaciones especiales cuando el compilador desea obtener un objeto como cadena de caracteres)

Por otra parte, el *LinkedList* no es tan eficiente en términos de rapidez en la búsqueda, pero sí lo es a la hora de actualizar la lista, dado que proporciona acceso secuencial óptimo, con inserciones y borrados en la parte central de la *List*. Tiene métodos como ***addFirst()***, ***addLast()***, ***getFirst()***, ***getLast()***, ***removeFirst()***, y ***removeLast()*** que permiten que se la trate de una pila o una cola o una lista doblemente enlazada.

Contenedores de tipo Set (Conjunto)

Set tiene exactamente la misma interfaz que *Collection*. Pero gracias a los principios de herencia y polimorfismo, es capaz de expresar un comportamiento distinto. En este caso, un *Set* es un objeto *Collection* que no guarda valores duplicados (esa es la diferencia en su comportamiento). Cada elemento que se añada al objeto *Set* debe ser único; si no es así, *Set* no añade el elemento duplicado. Los Objects añadidos a un *Set* deben implementar ***equals()*** para establecer la unicidad de los objetos. La interfaz *Set* no almacena sus elementos en ningún orden particular.

Dentro de los contenedores de tipo Conjunto o *Set* se encuentran el *HashSet* que suelen definir un ***HashCode()*** para mejorar la eficiencia en las búsquedas; y el tipo *TreeSet*, que es un *Set* ordenado respaldado por un árbol que permite extraer una secuencia ordenada de objeto *Set*, dado que *TreeSet* implementa la interfaz *SortedSet*.

Contenedores de tipo Map

Un objeto *ArrayList* permite hacer una selección a partir de una secuencia de objetos usando un número, por lo que en cierta forma asigna números a los objetos, pero si lo que se quiere es buscar objetos a partir de otros objetos se debe implementar una interfaz *Map*. La misma permite crear contenedores que mantengan asociaciones clave-valor (pares), de forma que se pueda buscar un valor usando una clave. El método ***put()***(Object clave, Object valor) añade un valor (el elemento deseado), y le asocia una clave (el elemento con el que buscar). ***get()***(Object clave) devuelve el valor a partir de la clave correspondiente. En caso en que las claves se repitan guarda una sola clave, y asocia el último valor ingresado. Veamos un ejemplo:

```
import java.util.Collection;
```



```
import java.util.Map;

public Map rellenar(Map m){

    m.put("Homer", "perro");

    m.put("Ramon", "gato");

    m.put("Felipe", "perro");

    return m;

}
```

La biblioteca estándar de Java tiene dos tipos diferentes de objetos *Map*: *HashMap* y *TreeMap*. Ambos implementan la misma interfaz, pero difieren en la forma de hacerlo.

En el caso del contenedor *HashMap*, la implementación basada en una tabla de tipo hash, cuyo rendimiento se puede ajustar mediante constructores que permiten especificar la capacidad y el factor de carga de la tabla de tipo hash.

En el caso del contenedor *TreeMap*, la implementación está basada en un árbol. Cuando se vean las claves de los pares, se ordenarán (en función de Comparable o Comparator). Esta es la característica fundamental de *TreeMap*: se logran resultados en orden. *TreeMap* es el único objeto *Map* con un método ***subMap()***, que permite devolver un fragmento de árbol.

Iteradores

En cualquier clase contenedora, hay que tener una forma de introducir y extraer elementos dado que esta es la funcionalidad primaria de un contenedor: almacenar elementos. Ahora bien, ¿es posible escribir un fragmento de código genérico independiente del tipo de contenedor con el que trabaje para recorrer una colección? Aquí aparece el concepto de iterador.

En programación, un iterador es un objeto cuyo trabajo es moverse a lo largo de una secuencia de objetos y seleccionar cada objeto de esa secuencia sin que el programador cliente tenga que saber u ocuparse de la estructura subyacente de esa secuencia. El ***Iterator*** de Java es un ejemplo de un iterador con cierto tipo de limitaciones. Para trabajar con un elemento iterador es necesario conocer ciertos aspectos cuando se solicita a un contenedor que proporcione un iterador utilizando un método denominado ***iterador()***.

1. Devuelve el primer elemento de la secuencia en la primera llamada al método ***next()***
2. Consigue el siguiente objeto de la secuencia con ***next()***
3. Verifica si hay más objetos en la secuencia con ***hasNext()***
4. Elimina el último elemento devuelto por el iterador con ***remove()***



Observemos un ejemplo donde un iterador recorre los valores que devuelve la colección veterinaria (en este caso el *ArrayList* se encuentra tipificado indicando el tipo de objeto Animal indicándolo entre símbolos <>: <Animal>)

```
@Test

public void testRecorrerUnaLista(){

    //List coleccion = new ArrayList();

    ArrayList<Animal> veterinaria = new ArrayList<Animal>();

    veterinaria.add(new Pez("cabeza", "tronco", "extremidades"));

    for(int i=0; i<100; i++){

        veterinaria.add(TestNaturaleza.crearAnimales());

    }

    veterinaria.add(null);

    veterinaria.add(new Reptil("Cabeza agregada despues del
    null", "", ""));

    Iterator<Animal> mascota = veterinaria.iterator();

    while(mascota.hasNext()) {

        try{

            System.out.println((mascota.next()).toString());

        }catch(NullPointerException e){

            System.err.println("Se escapó la tortuga");

        }

    }

}
```

La Interfaz COMPARABLE

La interfaz Comparable es una interfaz de apoyo para colecciones que forman parte del paquete java.lang. Proporciona un método llamado **compareTo()**, que permite a una clase especificar el criterio de comparación entre los objetos de la misma, para poder ordenar colecciones. Para que un array o colección de objetos sea ordenado según un orden natural de los objetos se debe definir el criterio de ordenación.

La definición de esta interfaz es:

```
interface Comparable <T>{

    int compareTo(T obj); //obj es el objeto contra el que se realiza
    la comparación

}
```

A fin de notificar el resultado de la comparación el método **compareTo()** devuelve un número entero menor que 0 (<0) cuando el objeto con el que se compara es mayor, igual a 0



(=0) si ambos objetos son iguales y mayor que 0 (>0) cuando el objeto que se compara es menor. Tanto String como las clases de envoltorio implementan la interfaz *Comparable*, no así las clases StringBuffer ni StringBuilder.

Sobreescritura de *equals()* y *hashCode()*

En caso de querer realizar comparación de referencias a objetos es necesario sobreescribir el método *equals()* ya que de lo contrario heredará la versión del método definida en la clase Object que devuelve true sólo en caso en que las referencias apunten al mismo objeto. Por otra parte, en las colecciones de tipo *Map*, donde se trabaja con claves únicas, los objetos de clases que no se sobreescriben el método *equals()* no podrán ser utilizados como claves. A la hora de sobreescribir *equals()*, es importante definir qué es lo que se entiende por igual teniendo en cuenta que el objeto pasado como parámetro sea una instancia de la clase a la que pertenece el objeto con el que se compara. La sintaxis se expresa de la siguiente forma:

```
public boolean equals(Object obj){  
    if(obj instanceof Clase) &&...
```

Más allá del criterio que se defina, la sobreescritura de *equals()* debe responder a los siguientes principios:

- Reflexividad: a.equals(a) debe devolver true
- Simetría: a.equals(b) debe devolver el mismo resultado que b.equals(a)
- Transitividad: si a.equals(b) devuelve true y b.equals(c) devuelve true, entonces a.equals(c) debe devolver true.
- Dada una referencia a no nula, a.equals(Null) debe devolver false. Si a es una referencia nula, se producirá una excepción NullPointerException.

En cuanto al método *hashCode()*, éste devuelve un valor asociado al objeto. Si se sobreescribe *equals()* también debe sobreescribirse *hashCode()*, ya que debemos tener en cuenta que dos objetos considerados iguales utilizando el método *equals()* deben tener idénticos valores de *hashCode()*. El valor hashCode de un objeto es utilizado por las colecciones para saber dónde colocar internamente cada objeto. El valor hashCode es utilizado por las colecciones HashTable, HashSet, HashMap, LinkedHashSet y LinkedHashMap.

Cuando se sobreescribe el método *hashCode()* hay que tener en cuenta que éste debe cumplir con ciertas propiedades:

- El valor devuelto por *hashCode()* no tiene porqué ser el mismo de una ejecución a otra en la misma aplicación.
- Si dos objetos son iguales de acuerdo a *equals()*, las llamadas a sus métodos *hashCode()* deben dar el mismo resultado.

Implementación de la Interfaz COMPARATOR

Si necesitamos ordenar conjuntos de objetos cuyas clases no implementan la interfaz Comparable, podemos recurrir a otra técnica que procede agregando una clase adicional donde se establecen los criterios de comparación, esta clase deberá implementar la interfaz Comparator. La interfaz Comparator se encuentra en el paquete java.util y proporciona un



método llamado ***compare()*** que proporciona un método de comparación entre objetos de una determinada clase, su formato es:

```
int compare (T ob1, T ob2);
```

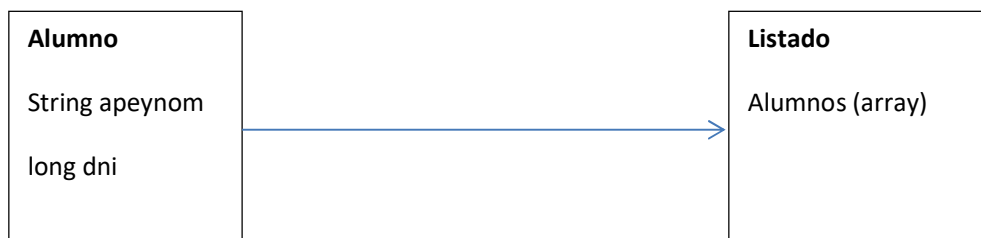
donde ob1 y ob2 son los objetos a comparar y T la clase a la cual pertenecen. En cuanto al valor devuelto por el método, también devuelve un número entero menor que 0 (<0) cuando el objeto 1 es menor que el objeto 2; igual a 0 (=0) cuando los objetos son iguales; y mayor que 0 (>=) cuando el objeto 1 es mayor que el objeto 2.

La utilización de la interfaz *Comparator* presenta una ventaja respecto de la interfaz *Comparable* ya que, de ser necesario definir varios objetos de ordenación para una clase, esto es posible creando varias clases que implementen la interfaz *Comparator*, cada una de ellas con su correspondiente criterio de comparación.

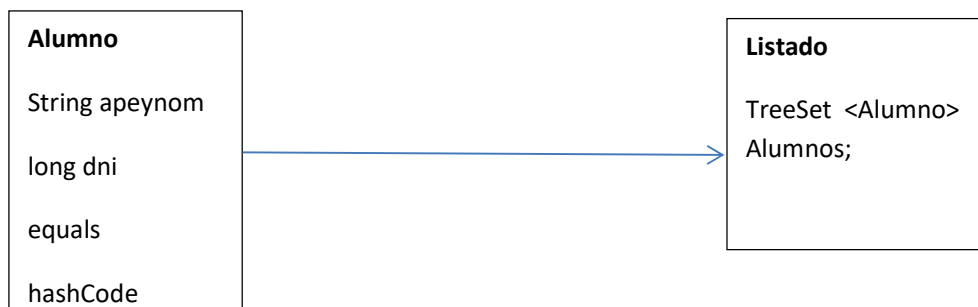
Notas finales

Veamos en un ejemplo práctico la implementación de la interfaz *Comparable*: Imaginemos la siguiente situación: se desea generar un listado de alumnos de un curso que han sido inscriptos según su DNI. Ahora bien, los alumnos son ingresados por DNI, dado que cada alumno debe ser único e irrepetible, sin embargo se desea ordenar la lista por apellido, utilizando al mismo como orden natural.

Tanto alumno como listado constituyen dos clases diferentes. Sabemos que existe otra forma de reutilizar la funcionalidad del código además de la herencia. Se trata de crear objetos de una clase existente dentro de otra clase. A esto se le llama ***composición***, donde una clase hace uso de objetos de clases existentes. Nuevamente se está reutilizando la funcionalidad del código. Desde este punto de vista podrían crearse dos clases: la clase Alumno y la clase Listado. El listado “tiene un” conjunto de alumnos (un array de tipo Alumno).



Esta relación de composición se resuelve con contenedores pero es necesario hacer una correcta selección de los mismos: en este caso, dado que los alumnos no han de repetirse y además su orden natural debe ser el ordenamiento por el apellido (atributo `apeynom`) lo más conveniente es seleccionar una colección de tipo `TreeSet`.





Cada vez que se agrega un alumno al *TreeSet* se ejecuta el método ***.add()***. Para que el método ***add()*** pueda ordenar elementos necesita de una interfaz que permita la comparación entre los dos elementos. Esta interfaz es *Comparable*; el método de comparación que utiliza la interfaz *Comparable* es ***compareTo()***, que va a estar dentro de un objeto y va a recibir como parámetro otro objeto. La interfaz *Comparable* en este caso se implementa en *Alumno*, ya que lo que se comparan son alumnos, así que en esa clase es donde se incluye el método ***compareTo()***, que debe determinar desde el punto de vista del apellido qué alumno va antes del otro. Recordemos que lo que devuelve el método ***compareTo()*** es un entero donde su valor puede ser menor, mayor o igual (en realidad que sean iguales no significa que los objetos sean iguales, sino a los fines del ordenamiento). Cuando son iguales devuelve 0, si el objeto *this* es mayor al objeto pasado por parámetro devuelve un número mayor a 0. Cuando el parámetro sea mayor a *this*, o sea *this* menor que el objeto pasado por parámetro devuelve un valor menor que 0. El método ***compareTo()*** de la interfaz *Comparable* permite que un contenedor de tipo *TreeMap* pueda agregar elementos a partir de la ejecución del método ***add()*** en forma ordenada (a partir del criterio de ordenamiento del método ***compareTo()***)