# Definición de clases

Para definir una clase se usa la palabra reservada **class** y el nombre de la misma

class NombreClase {

...

}

UML (Diagrama de clases)

|  |
| --- |
| NombreClase |
| Atributos |
| Operaciones |

|  |
| --- |
| **Punto** |
| x  y |
| dibuja(Graphics g) |

### Ejemplo de definición de una clase

//Punto.java

import java.awt.\*;

class Punto {

int x = 0;

int y = 0;

public Punto ( ) { x=5; y=5; }

public Punto (int x, inty ) {

this. x=x;

this. y=y;

}

int obtenX ( ) { return x; }

int obtenY ( ) { return y; }

void ponX (int x) { this.x=x; }

void ponY(int y ) { this.y=y;}

public void mueve (int dx, int dy ){

x=x+dx;

y=y+dy:

}

public void dibuja (Graphics g ){

g.drawLine (x, y, x, y );

}

}

La palabra reservada this sirve para la resolucion de nombres

this es el objeto que se uso para llamar al metodo dentro del cual se encuentra this

import Punto;

import java.applet.\*;

import java.awt.\*;

public class DibujaPunto extends Applet {

Punto p1,p2;

public void init ( ) {

p1 = new Punto ( );

p2 = new Punto (10,10);

}

public void paint (Graphics g){

p1. dibuja (g);

p2. dibuja (g);

}

}

## Constructores

El constructor es un método especial que tiene las siguientes características:

1.-Tiene el mismo nombre que la clase.

2.-Sirve para inicializar los objetos de la clase.

3-Si hay mas de uno esta sobrecargado.

4.-No tiene Tipo de Retorno (ni siquiera void(

5.-Es llamado de manera indirecta usando el operador **new.**

|  |  |
| --- | --- |
| El operador **new** hace 2 cosas  **-**asigna memoria  -llama al constructor | Creacion de objetos  -asignar memoria  -inicializa el objeto |

## Superposición de métodos

class ClaseImprime {

int x=0;

int y=1;

void imprime () {

System.out.println (“x es “+x);

System.out.println (“y es “+y);

}

}

class SubclaseImprime extends ClaseImprime{

int z=3;

public static void main (String args[] ){

SubclaseImprime obj=new SubclaseImprime ();

obj.imprime();

}

}

x es 0

y es 1

class SubclaseImprime2 extends ClaseImprime {

int z=3;

public imprime (){

System.out.println (“x es “ + x );

System.out.println (“y es “ + y );

System.out.println (“z es “ + z ):

}

public static void main (String args[] ){

SubclaseImprime2 obj=new SubclaseImprime2 ();

obj.imprime ();

}

}

x es 0

y es 1

z es 3

// de la clase ClaseImprime

void imprime (){

System.out.println(“x es “ + x);

System.out.println(“y es “ + y);

}

// de la clase SubclaseImprime2

void imprime (){

super.imprime ();

System.out.println(“z es “ + z);

}

x es 0

y es 1

z es 3

**Clases abstractas**

Una clase abstracta no puede crear instancias directas. En java una clase abstracta es aquella que contiene al menos un método abstracto. Los métodos abstractos no necesitan implementación; se requiere que las subclases no abstractas los implementen.

public abstract class MiClaseAbstracta {

int unaVariable;

public abstract int unaSubclaseNoAbstractaDebeImplementarme();

public void hazAlgo(){

…

}

}

public class UnaSubclaseConcreta extends MiClaseAbstracta {

public abstract int unaSubclaseNoAbstractaDebeImplementarme (){

// debemos implementar este método

…

}

}

**Interfaces en java**

Una definición de interfaz es: una interfaz es una declaración de un conjunto de métodos. Esta no lleva información sobre implementación, así una interfaz no puede ser implementada directamente. Una clase puede implementar múltiples interfaces.

public interface unaInterfaz{

public static final int laRespuesta=42;//ok

public abstract int laVidaEnElUniverso();//ok

long contadorDelBingBang=0; //ok

long edadDelUniverso();//ok

protected int unaConstante;

private int obtenUnEntero();

}

### Ejemplo de definición y uso de una interfaz

inteface Imprimible {

void imprime();

}

class Lunes implements Imprimible{

void imprime(){

System.out.println(“Lunes”);  
}

}

class Martes implements Imprimible{

void imprime(){

System.out.println(“Martes”);

}

}

class Miercoles implements Imprimible{

void imprime(){

System.out.println(“Miercoles”);  
}

}

class Jueves implements Imprimible{

void imprime(){

System.out.println(“Jueves”);

}

}

class Viernes implements Imprimible{

void imprime(){

System.out.println(“Viernes”);

}

}

Notación UML para interfaces

Nombre Interfaz

Clase

Implementadora

Clase

Implementadora

<<interfaz>>

Tipo

### Ejemplos

Imprimible

<<interfaz>>

Imprimible

Lunes

Lunes

Imprime():void

*Interfaces de elementos soportados por la clase java.awt.Component*

**public interface**EventListener {

}

public interface MouseListener extends EventListener {

public abstract void mouseClicked(MouseEvent e);

public abstract void mouseEntered(MouseEvent e);

public abstract void mouseExited(MouseEvent e);

public abstract void mousePressed(MouseEvent e);

public abstract void mouseReleased(MouseEvent e);

}

public interface ActionListener extends java.util.EventListener {

void actionPerformed (ActionEvent e);

}

si un objeto quiere recibir notificaciones de eventos del ratón debe de llamar el siguiente método

public void addMouseListener(MouseListener listener);

Si un objeto quiere recibir notificaciones de eventos acción se debe registrar llamando el siguiente método

public void addActionListener(ActionListener listener);

**Paquetes**

Los paquetes son la manera de Java para diseñar y organizar a gran escala. Además de las clases que nos permiten agrupar los campos y métodos, Java tiene una construcción llamada paquete que permite agrupar clases relacionadas. Clases seleccionadas de este paquete pueden ser usadas por otras clases que no estén en el paquete mediante el uso del comando import. Suponga que

quisiéramos crear un paquete de clases geométricas relacionadas que llamaremos el paquete forma. Podemos hacerlo usando un archivo por clase y poniendo una declaración package forma al principio de cada archivo.

//Forma.java

package forma;

class Forma {

...

}

---------------------------------------

//Punto.java

package forma;

class Punto extends Forma {

…

}

---------------------------------------

//Línea.java

package forma;

class Linea extends Forma {

…

}

---------------------------------------

//Triangulo.java

package forma;

class Triangulo extends Forma{

…

}

---------------------------------------

//Rectángulo.java

package forma;

class rectángulo extends Forma {

…

}

---------------------------------------

//Circulo.java

package forma;

class Circulo extends Forma{

…

}

---------------------------------------

//Elipse.java

package forma;

class Elipse extends Forma{

…

}

---------------------------------------

//Dibuja. Java

//importamos el paquete forma para usarlo

import forma.\*;

...

**Excepciones**

Una forma de manejar los errores en los lenguajes como C era regresar un valor para decirle al llamador de la función que un error ocurrió. Así siempre que una función tenia la posibilidad de regresar un código de error, tenia que checarse y una acción apropiada tenia que ser tomada, como en la siguiente pieza de código:

if(!escribirArchivo (algo)){

/\*error\*/

}

if(!escribirArchivo (algoMas)){

/\*error\*/

}

Las excepciones proveen una forma limpia para que un método aborte lo que sea que este haciendo y señale al llamador del método que algo salió mal. En contraste con las formas de manejo de error que devuelven códigos de error, el llamador puede ignorar la excepción que fue lanzada y dejar que el error se propague hacia arriba en la pila de llamadas hasta que alguno de los métodos maneje la excepción. Suponga que existe un método llamado escribe algo que usa retorno de códigos de error.

void escribeAlgo (String algo, String algoMás) {

if(! escribirArchivo(algo)){

/\*error\*/

return;

}

if(!escribirArchivo(algoMás)){

/\*error\*/

return;

}

}

Si se escribe dicho método usando excepciones y suponiendo que escribeArchivo lanzara una excepción.

void escribeAlgo(String algo, String algoMás){

try{

escribirArchivo(algo);

escribirArchivo(algomás);

}catch (excepción e ){

e. printStackTrace();

return;

}

}

Las instrucciones dentro del bloque try se escriben como si no ocurrieran excepciones. Si una excepción ocurre en cualquier parte del bloque try, las instrucciones del bloque catch se ejecutan.

Para lanzar una excepción, se usa el operador throw en Java. El argumento para este operador es una instancia de una subclase de la clase Exception.

void escribirArchivo (String algo){

if (archivoNoExiste){

throw new FileNotFoundException();

} else{

/\* escribir algo al archivo \*/

}

}

La instrucción throw aborta el método y lanza la excepción a lo largo de la pila de llamadas. Si hay una instrucción catch en el llamador que concuerde con la clase de la excepción o una de sus clases ancestros, entonces la excepción se detiene ahí y la instrucción catch manejara el problema. Sin embargo si el llamador no tiene una instrucción catch, la excepción continua viajando a lo largo de la pila hasta que encuentra una instrucción catch. El programa aborta si no hay ninguna instrucción catch.

escribirArchivo

excepción

escribeAlgo

escribeTodo

pila de

llamadas

manejador de

excepciones

Lo que pasa cuando se lanza una excepción.