II. JAVA

II.1. INTRODUCCIÓN:

JAVA es un lenguaje de programación orientado a objetos que se dio a conocer en 1995 y que comenzó como resultado de la búsqueda de un lenguaje para programar dispositivos empotrados. JAVA se define como un lenguaje:

- **Simple:** Los diseñadores de JAVA intentaron crear un lenguaje que un programador pudiera aprender con rapidez. Java utiliza muchas características similares a C y C++, más sin embargo los diseñadores de JAVA eliminaron otras tantas disponibles en esos lenguajes. Por ejemplo, JAVA no soporta la declaración *goto*, ni utiliza archivos de encabezado. Además los *struct y union* se han eliminado. También elimina las características de herencia múltiple de C++. La simplificación más importante es que JAVA no utiliza apuntadores.

- **Orientado a Objetos:** Significa que se debe poner atención especial a los datos de la aplicación y a los métodos que manipulan los datos y no se debe pensar estrictamente en términos de procedimientos. Se manejan por tanto las *clases* como una colección de datos y métodos que operan sobre dichos datos. Java posee un conjunto grande de clases ordenadas jerárquicamente por paquetes que se pueden utilizar en los programas.

- **Distribuido:** JAVA proporciona un soporte de alto nivel para redes. Existe una clase *URL* y un paquete *java.net* para trabajar fácilmente con un archivo o una fuente remota. Existen clases en JAVA para el manejo de la Invocación de Método Remoto o RMI que permiten que un programa de JAVA llame a métodos de objetos Java remotos, como si se tratara de objetos locales. Además Java también proporciona soporte de red tradicional, de bajo nivel, incluidos conexiones basadas en flujo por sockets.

- **Interpretado:** Java es un lenguaje interpretado pues el compilador de JAVA genera un *ByteCode* para la Máquina Virtual de Java (JVM-Java Virtual Machina) en vez de código nativo de máquina. Para ejecutar un programa de Java en forma real, se tiene que hacer uso del intérprete del lenguaje para ejecutar los *bytecode* compilados.

- **Robusto:** Java se ha diseñado para escribir software robusto y muy confiable ya que permite una verificación exhaustiva al tiempo de compilación, en busca de posibles problemas de no concordancia de letras. El manejo de excepciones en Java es una característica importante del lenguaje que contribuye a formar programas más robustos. Una excepción es una señal de que ha ocurrido algo excepcional, que no se puede evitar que ocurra más sin embargo si puede ser prevista.

- **Seguro:** Java se diseño con la seguridad en mente y proporciona varias capas de controles de seguridad en la red que protegen contra código malicioso. Estas capas permiten a los usuarios ejecutar con comodidad programas desconocidos como los Applets.

- **De Arquitectura Neutra:** Como los programas de Java se compilan en un formato de *bytecode* de arquitectura neutral, una aplicación de Java se puede ejecutar ne

cualquier sistema, siempre y cuando en dicho sistema se encuentre la máquina virtual de Java

- **Portable:** El hecho de que Java sea interpretado y defina un formato de *bytecode* estándar, de arquitectura neutra, es lo que le da la característica de ser portátil. Si se escribe una aplicación en Java, esta se podrá ejecutar en todas las plataformas que tengan instalada su máquina virtual (PC, Mac, y estaciones de trabajo UNIX o LINUX).

- **Multihilo:** Java es un lenguaje de hilos múltiples, es decir, proporciona soporte para varios hilos de ejecución que pueden manejar diferentes tareas "al mismo tiempo".

- **Dinámico:** En todo momento, una clase de JAVA se puede cargar en un intérprete de Java en ejecución. Dinámicamente se pueden crear casos (objetos) de estas clases cargadas.

II.1.1. MODELO DE UN PROGRAMA JAVA

Un programa en JAVA consiste en una o más definiciones de clase (donde una definición de clase se encuentra encerrada entre llaves como lo muestra la figura 1), y cada una de las clases ha sido compilada en su propios archivos *.class* de código objeto de la máquina virtual de Java. Una de estas clases habrá de definir el método[1] *main()*, que es donde el programa comienza a ejecutarse.

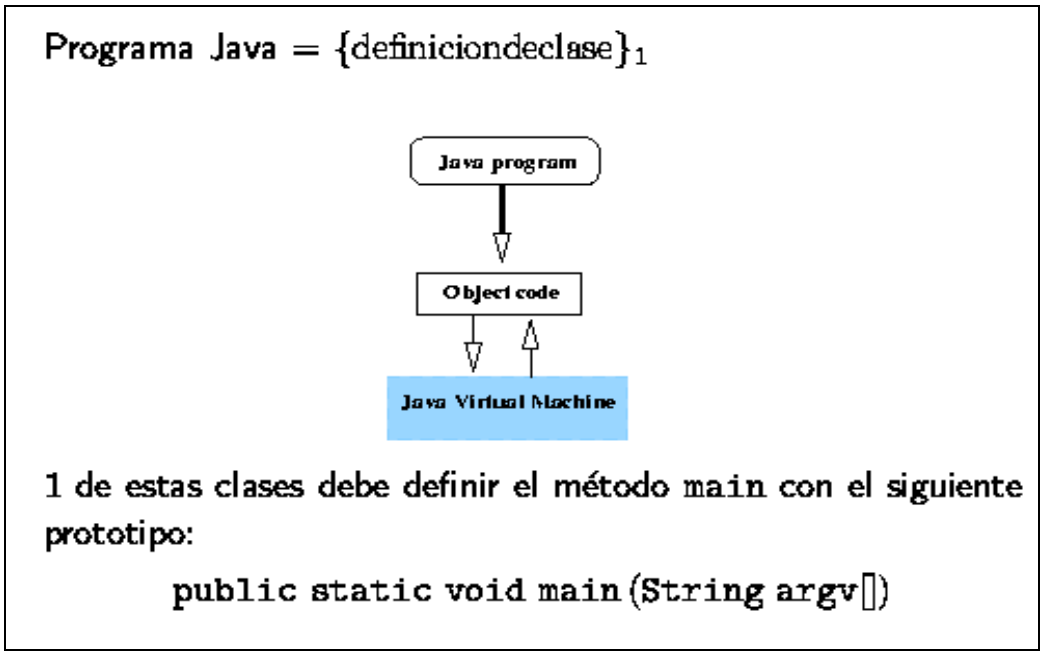


Figura 1. Modelo de un Programa Java

[1] Método es para Java lo que para C es una función

II.1.2. CONVENCIONES Y COMPILACION

Un archivo de código fuente en Java tiene la extensión *.java*. Consiste en una declaración *package* opcional, con un cierto número de declaraciones *import*[2], seguida por una o más definiciones de clase. Si en un archivo fuente de Java esta definida más de una clase, sólo una de ellas se puede declarar *public* (esto es, sólo una esta disponible fuera del paquete); además el archivo fuente debe tener el mismo nombre que la clase pública y la extensión *.java*.
Cada definición de clase en un archivo *.java* es compilada en un archivo separado. Estos archivos de bytecode de Java compilados se conocen como *"archivos de clase"* y deben tener el mismo nombre que la clase que definan, con la extensión *.class* anexada (ver figura 2).

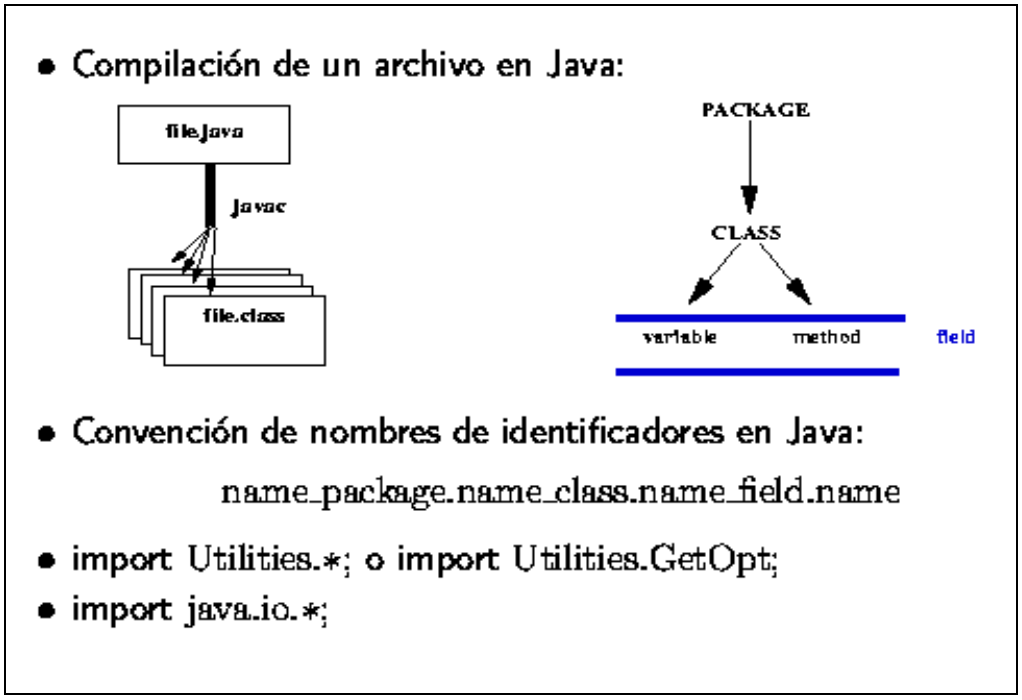


Figura 2. Convenciones y Compilación

Por convención el nombre de una clase en Java siempre empieza con una letra mayúscula seguida de cualquier cantidad de letras en minúscula. Si el nombre de una clase es una conjunción de dos o más palabras, por convención las palabras van juntas sin espacio y cada una de ellas empieza con su primera letra escrita en mayúscula. Esta misma convención se puede aplicar también a nombre de variables y/o métodos.

[2] La palabra reservada *import* es para Java lo que la palabrareservada *include* es para C

El siguiente código concreto de JAVA muestra el programa quizás más simple del lenguaje:

```
// La aplicación Hola Mundo
public class HolaMundo
   {
    public static void main(String[] args)
      {
        System.out.println(“Hola Mundo. . .“);
      }
   }
```

El programa define una clase pública que contiene el método main() que es el punto de entrada principal para todas las aplicaciones de JAVA. El cuerpo de main() consiste de sólo una línea que imprime la salida **“Hola Mundo”**.

El programa debe ser guardado como ya se mencionó, en un archivo con el mismo nombre que la clase pública, más la extensión *.java*. La sintaxis de compilación es:

javac *Nom_arch.java*

donde *javac* es el nombre del compilador del lenguaje Java y *Nom_arch* es el nombre del archivo que contiene el código fuente Java.

El compilador genera el archivo *nom_arch.class* dentro del directorio actual. Para la ejecución del programa se utilizará el intérprete del lenguaje mediante la sintaxis:

java *Nom_arch*

donde *java* es el nombre del intérprete del lenguaje Java y *Nom_arch* es el nombre del archivo *.class* que se creó después de la compilación si no hubo ningún error de sintaxis y que es donde se encuentra la codificación a bytecode de nuestro código fuente.

Observaciones:

- La primera línea es un comentario. Todo lo que viene después de la secuencia // hasta el fin de línea es un comentario. Java también acepta comentarios de la forma: /* ... */

- Luego viene la definición de una clase llamada HolaMundo:

  public class HolaMundo { ... }

- Dentro de la clase HolaMundo se define el método *main*:

  public static void main (String[] args) { ... }

  En una clase se definen uno o más métodos.

- Las palabras *public* y *static* son atributos del método que discutiremos más tarde.

- La palabra *void* indica que el método *main* no retorna ningún valor.

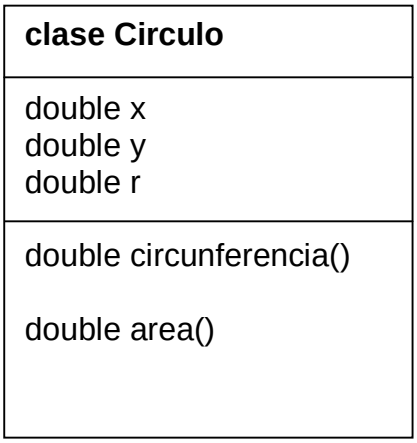
- La forma *(String[] args)* es la definición de los argumentos que recibe el método *main*. En este caso se recibe un argumento. Los corchetes [ ] indican que el argumento es un arreglo y la palabra *String* es el tipo de los elementos del arreglo (en particular String es el nombre de una clase de Java que define la semántica de cómo se trabajan las cadenas en el lenguaje).

  Por lo tanto *main* recibe como argumento un arreglo de objetos cadenas que corresponden a los argumentos con que se invoca el programa.

- La instrucción *System.out.println(...)* despliega un *string* en la consola. Java no posee una sintaxis abreviada para desplegar *strings*.

II.2. CLASES Y OBJETOS:

Una clase es una colección de datos, y métodos que operan sobre esos datos. Los datos y los métodos en conjunto sirven para definir algún tipo de objeto. Por ejemplo la clase Círculo:




```
public class Circulo
   {
     // Variables de Instancia

     public double x,y;
     public double r;


     // Métodos de Instancia

     public double Circunferencia()
        {
         return 2*3.14159*r;
        }

     public double area()
        {
         return 3.14159*r*r;
        }
   }
```

II.2.1. Instancias de clase:

Al definir la clase Círculo en JAVA se ha creado un nuevo tipo de dato, ya que variables de ese tipo pueden ser declaradas como:

```
Circulo c1,c2,c3,c4;
```

Pero estas variables son sólo nombres que hacen referencia a objetos círculo pero no son objetos en sí. En JAVA todo objeto se crea dinámicamente con la palabra *new*. Por ejemplo:

```
Circulo c1,c2,c3,c4;

c1= new Circulo();
c2= new Circulo();
c3= new Circulo();
c4= new Circulo();

o bien:

Circulo c1= new Circulo();
Circulo c2= new Circulo();
Circulo c3= new Circulo();
Circulo c4= new Circulo();
```

Se dice entonces que c1, c2, c3 y c4 son variables de tipo Círculo que contienen instancias de la clase Circulo (es decir, objetos Círculo).

II.2.2. Acceso a los datos del Objeto:

Una vez creado un objeto se pueden utilizar sus campos de datos o variables de instancia de la siguiente forma a manera de ejemplo:

```
Circulo c1= new Circulo();

// Objeto Circulo c1 con coordenadas (2,2) y radio de valor 1
c1.x=2.0;
c1.y=2.0;
c1.r=1.0;
```

II.2.3. Uso de métodos del Objeto:

Así mismo se pueden utilizar los método de instancia del objeto sobre si mismo (a través de sus datos) de la siguiente forma a manera de ejemplo:

```
Circulo c= new Circulo();
double a;

c.x=2.0;
c.y=2.0;
c.r=2.5;

a=c.area();
```

II.2.4. El Constructor (Creación de un objeto):

Toda clase en JAVA tiene por lo menos un método constructor (con el mismo nombre que la clase), cuyo propósito es efectuar toda la inicialización necesaria para el nuevo objeto. En el ejemplo de los círculos:

```
Circulo c= new Circulo();
```

JAVA proporciona un constructor por omisión que ni tima argumentos ni efectúa una inicialización especial. A continuación, un ejemplo de un constructor para la clase Círculo definido por el programador:

```
public class Circulo
   {
     // Variables de Instancia

     public double x,y;
     public double r;


     //Método Constructor

     public Circulo(double x, double y, double r)
        {
          this.x=x; //El argumento implícito llama “this”
          this.y=y; // y hace referencia a “este” objeto
          this.r=r; //o variable
        }
```

```
    // Métodos de Instancia

    public double Circunferencia()
       {
        return 2*3.14159*r;
       }

    public double area()
       {
        return 3.14159*r*r;
       }
  }
```

ANTES	DESPUES

`Circulo c;`
`c=new Circulo();`
`c.x=1.414;`
`c.y=-1.0;`
`c.r=0.25;`
	`Circulo c= new Circulo(1.414,-1.0,0.25)`

IMPORTANTE:

- El nombre del constructor es siempre el mismo que el de la clase

- El tipo de retorno es implícitamente una instancia de la clase. Un constructor no debe usar una declaración *return* para regresar un valor, ni se emplea la palabra clave *void* para definir el tipo de retorno ya que implícitamente es “this”.

Se puede inicializar un objeto de distintas maneras usando constructores múltiples. Por ejemplo:

```
public class Circulo
   {
     // Variables de Instancia

     public double x,y;
     public double r;
```

```
    //Métodos Constructores
    public Circulo(double x, double y, double r)
       {
         this.x=x; //El argumento implícito llama “this”
         this.y=y; // y hace referencia a “este” objeto
         this.r=r; //o variable
       }

    public Circulo(double r)
       {
         x=0.0;
         y=0.0;
         this.r=r;
       }

    public Circulo(Circulo c)
       {
         x=c.x;
         y=c.y;
         r=c.r;
       }

    public Circulo()
       {
         x=0.0;
         y=0.0;
         r=1.0;
  }

    // Métodos de Instancia
    public double Circunferencia()
       {
        return 2*3.14159*r;
       }

    public double area()
       {
        return 3.14159*r*r;
       }
  }
```

IMPORTANTE:

Todos los constructores tienen el mismo nombre. En JAVA un método se distingue mediante su nombre, el número, el tipo y posición de sus argumentos.

El hecho de definir métodos con el mismo nombre y distintos tipos de argumentos se llama ***SOBRECARGA DE METODOS***.

II.2.5. Variables de Clase:

En la definición de la clase Círculo, se declararon 3 variables de instancia: *x,y,r*. Cada instancia de la clase, es decir, cada Círculo tiene su propia copia de estas tres variables. Sin embargo en ocasiones se requiere una variable de la cual sólo haya una copia; algo así como una variable global aunque JAVA no permite variables globales.

JAVA utiliza la palabra clave *"static"* para indicar que una variable particular es una *variable de clase* y no una variable de instancia. Es decir, sólo hay una copia de la variable asociada con la clase. A este tipo de variable también se le denomina *variable estática*. Por ejemplo:

```
public class Circulo
   {
     // Variable de clase
     static int num_circulos=0; //Cuenta el no. circulos
                                // creados...
     // Variables de Instancia
     public double x,y;
     public double r;

     //Métodos Constructores
     public Circulo(double x, double y, double r)
        {
          this.x=x; this.y=y; this.r=r;
          num_circulos++;
        }

     public Circulo(double r) // Hace referencia al
        { this(0.0,0.0,r); }  // constructor de arriba...

     public Circulo(Circulo c)
        { this(c.x,c.y,c.r); }

     public Circulo()
        { this(0.0,0.0,1.0); }




     // Métodos de Instancia
```

```
     public double Circunferencia()
        {
         return 2*3.14159*r;
        }

     public double area()
        {
         return 3.14159*r*r;
        }
   }
```

Para accesar a las variables de clase se utiliza el siguiente formato:

```
Nombre_de_clase.Variable_de_clase
```

Por ejemplo:

```
System.out.println("Num. de Circulos creados:"
                                    +Circulo.num_circulos);
```

II.2.6. Constantes (Otro tipo de variables de clase):

Cuando una variable de clase además de ser declarada con la palabra *static* se declara también con la palabra clave *final*, dicha variable se considera como si fuera una constante. Por ejemplo

```
public class Circulo
   {
     public static final double PI= 3.14159265358979323846;
     public double x,y,r;
     public static int num_circulos=0;
     .
     .
     .
   }
```

Circulo.PI=4; ß Error

double a= 2*Circulo.PI*r; ß Correcto

II.2.7. Métodos de clase:

Los métodos de clase se asemejan a las variables de clase en cuanto a que:

- Los métodos de clase se declaran con la palabra clave *static*
- Los métodos de clase se denominan métodos estáticos
- Los métodos de clase se invocan mediante la clase y no con una instancia (objeto).

Por ejemplo; si queremos utilizar el método de JAVA que se emplea para calcular la raíz cuadrada de un número; escribiríamos:

```
Math.sqrt(número);
```

Esto es así ya que *sqrt()* es un método estático o un método de clase, en este caso de la clase *Math* que es donde se encuentra definido.

En el ejemplo siguiente se muestran 2 definiciones sobrecargadas de un método para la clase círculo. Uno es un método de instancia y el otro un método de clase:

```
public class Circulo
   {
     public double x,y,r;

    // Método de instancia. Regresa el más grande de dos
    // círculos...
    public Circulo masgrande(Circulo c)
       {
         if (c.r > r) return c;
         else return this;
       }

    // Un método de clase. Regresa el más grande de dos
    // círculos...
   public static Circulo masgrande(Circulo a, Circulo b)
      {
         if (a.r > b.r)
     return a;
          else return b;
      }
     .
     .
   }
```

La invocación del método de instancia sería la siguiente:

```
Circulo a= new Circulo(2.0);
Circulo b= new Circulo(3.0);
```

```
Circulo c= a.masgrande(b);
```

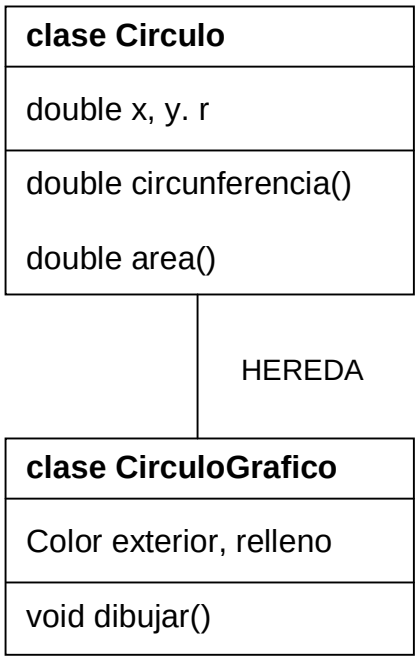
La invocación del método de clase sería la siguiente:

```
Circulo a= new Circulo(2.0);
Circulo b= new Circulo(3.0);

Circulo c= Circulo.masgrande(a,b);
```

II.3. SUBCLASES Y HERENCIA:

La herencia es un mecanismo para compartir y delegar tanto conocimiento como comportamiento de entidades software en sistemas complejos, permitiendo a nuevas clases reutilizar parte de otras clases que han sido previamente definidas.

Por ejemplo, podríamos definir la clase CirculoGrafico (que dibuja un circulo en pantalla) como una extensión o subclase de la clase Círculo de la manera siguiente:




```
public class CirculoGrafico extends Circulo
   {
      // Automáticamente se heredan las variables y métodos
de Círculo
```

```
    Color exterior, relleno;

     public void dibujar(Dibujaventana dw)
    {
      dw.dibujarCirculo(x,y,r,exterior,relleno);
       // x,y,r estan definidas en Circulo...
    }
    .
    .
    .
  }
```

La palabra *extends* dice a JAVA que CirculoGrafico es una subclase de Circulo y que hereda los campos y métodos de dicha clase, o bien en otras palabras; la clase Circulo es la superclase de la clase CirculoGrafico y por tanto esta última es una extensión de la primera. Por ejemplo:

```
CirculoGrafico gc= new CirculoGrafico();

double area= gc.area();

Circulo c=gc;   // Todo objeto CírculoGrafico es un objeto
                // Circulo)
```

II.3.1. Clases Finales:

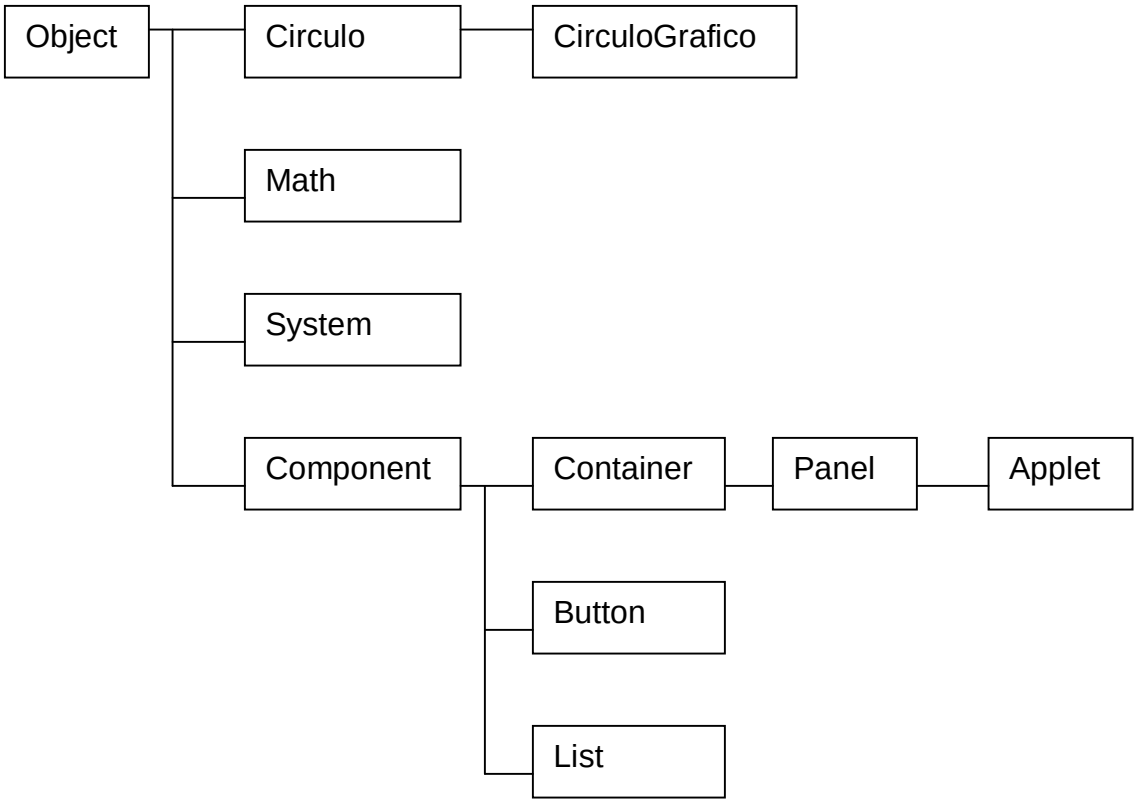
Cuando una clase se declara con el modificador *final*, no es posible extender ni formar una subclase de la misma.

II.3.2. Superclases y Jerarquías de Clases:

Toda clase que se defina en JAVA tendrá necesariamente una superclase, aun cuando no se especifique la superclase con la claúsula *extends* en cuyo caso la superclase será la clase *Object* que es la clase sobre la cual cuelgan todas las demás clases inmediatas, siendo ésta la másimportante debido a que:

- La clase Object es la única clase sin una superclase
- Los métodos definidos en cla clase Object pueden ser llamados por cualquier objeto de JAVA.

Un ejemplo de una jerarquía de clases puede ser la siguiente:



II.3.3. Constructores de Subclase:

Generalmente dentro de los constructores de sublase se utiliza la palabra reservada *super* que sirve para invocar el método constructor de una superclase y sólo se puede utilizar de esta manera y debe aparecer como la primera instrucción dentro del constructor. Por ejemplo:

```
public class CirculoGrafico( double x, double y, double r,
                             Color exterior, Color relleno)
   {
     this.x=x;
     this.y=y;
     this.r=r;
     this.exterior=exterior;
```

```
     this.relleno=relleno;

     // En este constructor se duplica el código del
     // constructor Circulo, y si x,y,r fueron declaradas
     // como private en Circlo, no podrían ser inicializadas
     // de esta forma
   }
```

```
public class CirculoGrafico( double x, double y, double r,
                             Color exterior, Color relleno)
   {
     super(x,y,r);
     this.exterior=exterior;
     this.relleno=relleno;
   }
```

Si un constructor no invoca ningún constructor de su superclase, JAVA lo hace por implicación. Pero si una clase se declara sin constructor alguno, JAVA agrega implícitamente un constructor a la clase de la forma:

```
public nom_clase_constructor() { super(); }
```

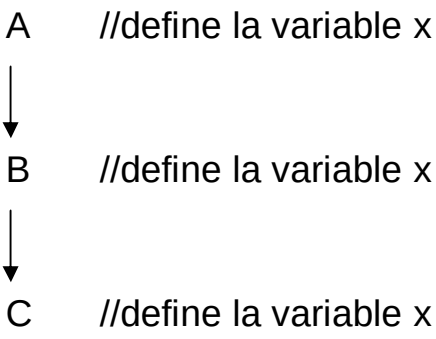
Por ejemplo:

```
class A {
          int i;

          public A() {
                      // Aquí va una llamada implícita a
                      //  super()

                      i=3;
                     }
        }


class B extends A {
                    // Constructor por omision:
                    // public B() { super(); }
                  }
```

II.4. ENSOMBRECIMIENTO E INVALIDACIÓN

II.4.1. Variables Ensombrecidas:

Si suponemos la siguiente jerarquía de clases, donde cada una de ellas define una misma variable *"x"***:**



Suponiendo que C es una subclase de B y ésta última a su vez es una subclase de A. Se dice entonces que la clase C ensombrece la variable x definida también en las clases A y B y de igual manera la clase B ensombrece la variable x definida también en A. Si suponemos todo esto y además damos por hecho que estamos trabajando en la clase C, las siguientes expresiones son válidas excepto la última:

```
x                   // Es la variable de la clase C

this.x              // Es la variable de la clase C

super.x             // Es la variable de la clase B

((B)this).x         // Es la variable de la clase B

((A)this).x         // Es la variable de la clase A

super.super.x       // Es un error sintáctico
```

II.4.2. Invalidación de Métodos:

Cuando una clase define un método que usa el mismo nombre, el mismo tipo de retorno y los argumentos como un método en su subclase. El método en dicha clase invalida el método en la superclase (No confundir con la sobrecarga de métodos). Así, cuando se invoca el método para un objeto de la clase, se llama a la nueva definición del método (el de la subclase) y no a la definición antigua de la superclase. Por ejemplo:

```
class A {
```

```
          int i=1;
          int f() { return i; }
        }

class B extends A {
                  int i=2;
                  int f() { return –i }
                }

public class prueba_invalidacion
   {
    public static void main(String args[])
     {
      B b= new B();
      System.out.println(b.i);     // i=2
      System.out.println(b.f());   // regresa -2

      A a= (A)b;    // Modela b como instancia de A
      System.out.println(a.i);     // i=1
      System.out.println(a.f());   // regresa -2
     }                          // Sigue referencia a B.f()
   }
```

NOTA: No se pueden invalidar métodos estáticos, privados y finales. Tampoco se pueden invalidar los métodos de una clase final.

II.4.2.1. Como invocar a un método invalidado:

```
class A {

          int i=1;
          int f() { return i; }
        }

class B extends A {
                  int i;
                  int f() {
                            i=super.i+1 //representa A.i
                            return super.f()+i
                                      //invoca a A.f()
                          }
                }
```

II.5. OCULTAMIENTO Y ENCAPSULACION DE DATOS:

A la técnica de la Orientación a Objetos de ocultar datos dentro de la clase para dejarlos disponibles sólo mediante los métodos se le conoce como **encapsulamiento**. En

otras palabras, cuando todas las variables de una clase están ocultas, los métodos de la clase definen las únicas operaciones que se pueden efectuar en los objetos de dicha clase para procesar dichas variables.

Si se quieren ocultar variables y/o métodos, bastará con declararlos como privados (con la palabra private). Ejemplo:

```
public class LavanderiaAutomatica {          // El usuario puede usar esta clase
      private Lavanderia[] sucio, limpio;     // El usuario no puede ver estas vars. internas
      public void lavar() { . . . }       // Sin embargo, si puede utilizar estos métodos
      public void secar() { . . .}        // para manipular dichas variables…
      .
      .
      .
      }
```

II.5.1. Modificadores de Visibilidad:

a) *private* (visible solo para los métodos definidos dentro de la clase)
b) *private protected* (visible dentro de su clase y subclases)
c) *protected* (visibilidad de private protected más visibilidad en el package al que pertenece la clase)
d) *public* (visible en todas partes)
e) *nivel de visibilidad por omisión* (dentro de su clase y de las clases que pertenecen al mismo paquete)

II.5.2. Ejemplo de Ocultamiento de variables en la clase Circulo

```
public class Circulo
   {
    protected double x,y;
    protected double r;
    private static final double MAXR= 100.0;

    private boolean checar_radio(double r)
     { return (r <=MAXR); }

    // Constructores públicos
    public Circulo(double x, double y, double r)
     {
       this.x=x; this.y=y;
      if (checar_radio(r)) this.r=r;
      else this.r= MAXR;
     }

    public Circulo(double r)
```

```
     { this(0.0,0.0,r); }

    public Circulo()
     { this(0.0,0.0,1.0); }

    // métodos públicos de acceso a datos
    public void mover_a(double x, double y)
     { this.x=x;     this.y=y; }

    public void mover(double dx, double dy)
     { x+=dx;   y+=dy; }

    public void ponerRadio(double r)
     { this.r= (checar_radio(r)) ? r : MAXR; }

    // métodos triviales
   public double conseguirX() { return x; }
   public double conseguirY() { return y; }
   public double conseguirRadio() { return r; }
   }
```

II.6. CLASES Y METODOS ABSTRACTOS

Las clases y los métodos abstractos se definen mediante la palabra clave "abstract". Un método abstract no tiene cuerpo sino solo una definición de señal o firma seguida de un punto y coma. Algunas reglas de los métodos y clases abstract son:

- Cualquier clase con un método abstract se vuelve automáticamente abstract en sí y debe ser declarada como tal.
- Una clase se puede declarar abstract, aun cuando no tenga métodos abstract.
- Una clase abstract no se puede iniciar
- La subclase de una clase abstract se inicia al invalidar todos los métodos abstract de su superclase y proporciona una instrumentación para todos ellos.
- Si una subclase de una clase abstract no instrumenta todos los métodos abstract que hereda, dicha subclase es en sí abstract.

El siguiente ejemplo muestra una clase abstract *Forma* y dos subclases no abstract de ella:

```
public abstract class Forma
   {
    public abstract double area();
    public abstract double perimetro();
   }
```

```
class Circulo extends Forma
   {
    protected double x,y;
    protected double r;
    private static final double PI= 3.141592;

    public Circulo() {r=1.0;}
    public Circulo(double r) { this.r=r; }

    public double area() { return PI*r*r; }
    public double perimetro() { return 2*PI*r;}

    public double conseguirRadio() { return r;}
   }

class Rectangulo extends Forma
   {
    protected double w,h;

    public Rectangulo() { w=0.0; h=0.0 }
    public Rectangulo(double w, double h) {this.w=w; this.h=h;}

    public double area() { return w*h;}
    public double perimetro(return 2*(w+h);}
    public double ancho() { return w; }
    public double altura() { return h;}
   }
```

La utilidad del anterior ejemplo puede ser ilustrada con la siguiente parte de código fuente de la aplicación principal de este problema:

```
Forma[ ] formas= new Forma[3];

formas[0] = new Circulo(2.0);
formas[1] = new Rectángulo (1.0, 3.0);
formas[2] = new Rectángulo(4.0,2.0);

double total_area= 0;

for (int i=0; i<formas.length; i++)
        total_area= formas[i].area();
```

II.7. INTERFACES

Las interfaces son módulos compuestos por un conjunto de métodos abstractos y constantes. Una interfaz se parece mucho a una clase abstracta, excepto en que utiliza la palabra clave *interface* en vez de las palabras *"abstract"* y *"class"*.

Se dice entonces que una clase puede extender su superclase (extends) pero también puede implementar (implements) opcionalmente una interfaz.

II.7.1. Ejemplo de una interfaz:

```
public interface Dibujable
   {
    public void ponerColor(Color c);
    public void ponerPosicion(double x, double y);
    public void dibujar(DibujarVentana dw);
   }
```

II.7.2. Ejemplo de la instrumentación de una interfaz:

```
public class RectanguloDibujable extends Rectángulo implements
                                                        Dibujable
{
  //Nuevas variabes de instancia...
   private Color c;
   private double x,y;

  //Un constructor
   public RectanguloDibujable(double w, double h) { super(w,h); }

  //Instrumentaciones de los métodos de la interfaz heredando
  //todos los métodos públicos de Rectángulo

   public void ponerColor(Color c) { this.c = c; }
   public void ponerPosicion(double x, double y)
           { this.x=x; this.y=y;}
   public void dibujar (DibujarVentana dw)
           { dw.dibujarRect(x,y,w,h,c);}
}


public class CirculoDibujable extends Circulo implements
                                                      Dibujable
{
   .
   .
   .
}
```

II.7.3. Ejemplo del uso de interfaces:

```
Forma[] formas= new Formas[3];
Dibujable[] dibujables= new Dibujable[3];

CirculoDibujable dc= new CirculoDibujable(1.1);
CuadradoDibujable ds= new CuadradoDibujable(2.5);
RectanguloDibujable dr= new RectanguloDibujable(2.3,4.5);
```

```
formas[0]=dc;   dibujables[0]=dc;
formas[1]=ds;   dibujables[1]=ds;
formas[2]=dr;   dibujables[2]=dr;

double total_area=0;
DibujarVentana cuadro= new DibujarVentana(3.0,10.0,50.0, 155.0);

for (int i=0; i<formas.length; i++)
   {
    total_area+= formas[i].area();
    dibujables[i].ponerPosicion(i*10.0,i*10.0);
    dibujables[i].dibujar(cuadro);
   }

.
.
.
```

II.7.4. Modelado informal de la aplicación: