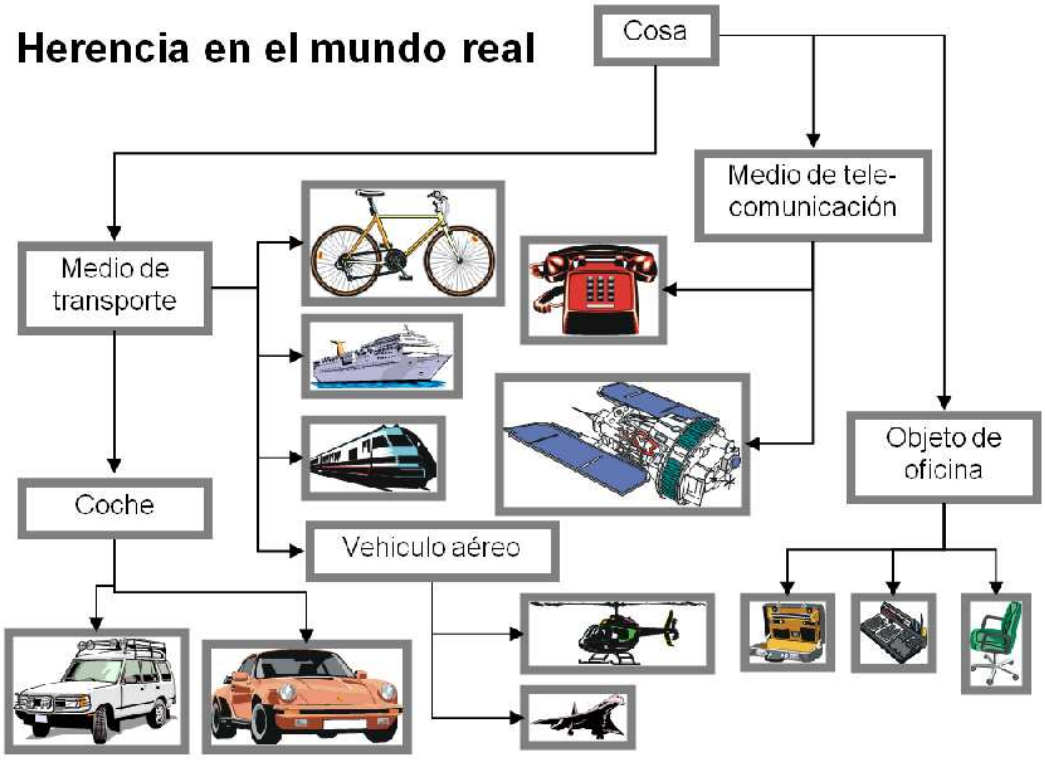
TEMA 7: PROGRAMACIÓN ORIENTADA A OBJETOS CON JAVA (II)

1. HERENCIA



1.1. Concepto de herencia

Se puede construir una clase a partir de otra mediante el mecanismo de la ***herencia***. Para indicar que una clase deriva de otra se utiliza la palabra ***extends***, como por ejemplo:

```
class CirculoGrafico extends Circulo {...}
```

Cuando una clase deriva de otra, hereda todas sus variables y métodos. Estas funciones y variables miembro pueden ser ***redefinidas*** (***overridden***) en la clase derivada, que puede también definir o añadir nuevas variables y métodos. En cierta forma es como si la ***sub-clase*** (la clase derivada) "contuviera" un objeto de la ***super-clase***; en realidad lo "amplía" con nuevas variables y métodos.

Java permite múltiples niveles de herencia, pero no permite que una clase derive de varias (no es posible la herencia múltiple). Se pueden crear tantas clases derivadas de una misma clase como se quiera.

Todas las clases de ***Java*** creadas por el programador tienen una ***super-clase***. Cuando no se indica explícitamente una ***super-clase*** con la palabra ***extends***, la clase deriva de ***java.lang.Object***, que es la clase raíz de toda la jerarquía de clases de ***Java***. Como consecuencia, todas las clases tienen algunos métodos que han heredado de ***Object***.

La ***composición*** (el que una clase contenga un objeto de otra clase como variable miembro) se diferencia de la ***herencia*** en que incorpora los datos del objeto miembro, pero no sus métodos o interface (si dicha variable miembro se hace ***private***).

```
Ejemplo: Clase Pixels.
        // fichero pixels.java
        class Pixels extends Punto {
              private byte color;
```

```
        public Pixel(double vx, double vy, byte vc) {
               super(vx, vy);    // debe ser la primera sentencia
               color=vc;
        }
        .......
}
```

1.2 La clase Object.

Como ya se ha dicho, la clase ***Object*** es la raíz de toda la jerarquía de clases de ***Java***. Todas las clases de ***Java*** derivan de ***Object***.

La clase ***Object*** tiene métodos interesantes para cualquier objeto que son heredados por cualquier clase. Entre ellos se pueden citar los siguientes:

1. Métodos que pueden ser redefinidos por el programador:
   - ***clone()*** Crea un objeto a partir de otro objeto de la misma clase. El método original heredado de ***Object*** lanza una *CloneNotSupportedException*. Si se desea poder clonar una clase hay que implementar la interface ***Cloneable*** y redefinir el método ***clone()***. Este método debe hacer una copia miembro a miembro del objeto original. No debería llamar al operador ***new*** ni a los constructores.
   - ***equals()*** Indica si dos objetos son o no iguales. Devuelve ***true*** si son iguales, tanto si son referencias al mismo objeto como si son objetos distintos con iguales valores de las variables miembro.
   - ***toString()*** Devuelve un ***String*** que contiene una representación del objeto como cadena de caracteres, por ejemplo para imprimirlo o exportarlo.
   - ***finalize()*** Este método ya se ha visto al hablar de los ***finalizadores***.
2. Métodos que no pueden ser redefinidos (son métodos ***final***):
   - ***getClass()*** Devuelve un objeto de la clase ***Class***, al cual se le pueden aplicar métodos para determinar el nombre de la clase, su super-clase, las interfaces implementadas, etc. Se puede crear un objeto de la misma clase que otro sin saber de qué clase es.
   - ***notify()***, ***notifyAll()*** y ***wait()*** Son métodos relacionados con las ***threads.***

1.3. Redefinición de métodos heredados.

Una clase puede ***redefinir*** (volver a definir) cualquiera de los métodos heredados de su ***super-clase*** que no sean ***final***. El nuevo método sustituye al heredado para todos los efectos en la clase que lo ha redefinido.

Los métodos de la ***super-clase*** que han sido redefinidos pueden ser todavía accedidos por medio de la palabra ***super*** desde los métodos de la clase derivada, aunque con este sistema sólo se puede subir un nivel en la jerarquía de clases.

Los métodos redefinidos pueden ***ampliar los derechos de acceso*** de la ***super-clase*** (por ejemplo ser ***public***, en vez de ***protected*** o ***package***), pero nunca restringirlos.

Los ***métodos de clase*** o ***static*** no pueden ser redefinidos en las clases derivadas.

Ejemplo:

```
class Pixels extends Punto {
        .........
        double distancia(double cx, double cy) {
```

```
                    double dx=Math.abs(x-cx);
                    double dy=Math.abs(y-cy);
                    return dx+dy;           // distancia manhattan
                }
        }
```

1.4. Un ejemplo con herencia

En las siguientes líneas de código se muestra un ejemplo completo de herencia, en este caso, tenemos una clase **Persona** que guarda el nombre y la edad de una persona, y una clase que **Alumno** que hereda de esta clase persona. Son de destacar los siguientes aspectos de la clase Alumno:

- El constructor, llama al constructor de su superclase para acabar de inicializar toda la información que se tiene de un alumno (nombre, edad, curso y nivel académico).
- El método *toString*, se ha redefinido, ampliando en este caso la información que devuelve, pero aprovechando lo que ya se había hecho en la superclase.
- Se ha añadido un nuevo método setEdad.
- Cuando se cree un objeto de la clase Alumno, se podrán usar todos los métodos de su clase más aquellos métodos de su superclase que no sean privados.

```
class Persona {
   private String nombre;
   private int edad;
   public Persona() {}
   public Persona (String n, int e)
   {
       nombre = n;
       edad = e;
       }

   public String toString() { return nombre + edad; }
   public void setEdad(int e) { edad = e; }

}

class Alumno extends Persona {
   private int curso;
   private String nivelAcademico;
   public Alumno (String n, int e, int c, String nivel) {
          super(n, e);
          curso = c; nivel_academico = nivel;
   }

  public String toString() {
          return super.toString() + curso + nivelAcademico;

  public void setCurso(int c) { curso = c; }
}


public static void main(String[] args) {
          Alumno a = new Alumno("Pepe", 1, 2, "bueno");
          System.out.println(a.toString());
          a.setCurso(4);
          a.setEdad(15);
   }
```

1.5. Constructores en clases derivadas

Ya se comentó que un ***constructor*** de una clase puede llamar por medio de la palabra ***this*** a otro ***constructor*** previamente definido en la misma clase. En este contexto, la palabra ***this*** sólo puede aparecer en la primera sentencia de un ***constructor***.

De forma análoga el ***constructor*** de una clase derivada puede llamar al ***constructor*** de su ***super-clase*** por medio de la palabra ***super( )***, seguida entre paréntesis de los argumentos apropiados para uno de los constructores de la ***super-clase***. De esta forma, un ***constructor*** sólo tiene que inicializar directamente las variables no heredadas.

La llamada al ***constructor*** de la ***super-clase*** debe ser la ***primera sentencia del constructor***, excepto si se llama a otro constructor de la misma clase con ***this()***. Si el programador no la incluye, ***Java*** incluye automáticamente una llamada al ***constructor por defecto*** de la ***super-clase***, ***super()***. Esta llamada en cadena a los ***constructores de las super-clases*** llega hasta el origen de la jerarquía de clases, esto es al constructor de ***Object***.

Como ya se ha dicho, si el programador no prepara un ***constructor por defecto***, el compilador crea uno, inicializando las variables de los ***tipos primitivos*** a cero, los ***Strings*** a la cadena vacía y las ***referencias*** a objetos a ***null***. Antes, incluirá una llamada al constructor de la ***super-clase***.

En el proceso de finalización o de liberación de recursos (diferentes de la memoria reservada con ***new***, de la que se encarga el ***garbage collector***), es importante llamar a los ***finalizadores*** de las distintas clases, normalmente en orden inverso al de llamada de los constructores. Esto hace que el ***finalizador de la sub-clase*** deba realizar todas sus tareas primero y luego llamar al finalizador de la super-clase en la forma ***super.finalize()***. Los métodos ***finalize()*** deben ser al menos ***protected***, ya que el método ***finalize()*** de ***Object*** lo es, y no está permitido reducir los permisos de acceso en la herencia.

2. CLASES Y MÉTODOS ABSTRACTOS

Una ***clase abstracta*** (***abstract***) es una clase de la que no se pueden crear objetos. Su utilidad es permitir que otras clases deriven de ella, proporcionándoles un marco o modelo que deben seguir y algunos métodos de utilidad general. Las clases abstractas se declaran anteponiéndoles la palabra ***abstract***, como por ejemplo,

```
public abstract class Geometria { ... }
```

Una clase ***abstract*** puede tener métodos declarados como ***abstract***, en cuyo caso no se da definición del método. Si una clase tiene algún método ***abstract*** es obligatorio que la clase sea ***abstract***. En cualquier ***sub-clase*** este método deberá bien ser redefinido, bien volver a declararse como ***abstract*** (el método y la ***sub-clase***).

Una clase ***abstract*** puede tener métodos que no son ***abstract***. Aunque no se puedan crear objetos de esta clase, sus ***sub-clases*** heredarán el método completamente a punto para ser utilizado.

Como los métodos ***static*** no pueden ser redefinidos, un método ***abstract*** no puede ser ***static***.

Ejemplo:

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

class Cuadrado extends Figura {
      prívate double lado;
      public cuadrado(double l) {lado=l;}
```

```
        public double perimetro() {return lado*4;}
        public double area() {return lado*lado;}
    }

    class Circulo extends Figura {
        private double radio;
        public Circulo(double r) {radio=r;}
        public double perimetro() {return 2*Math.PI*radio;}
        public double area() {return Math.PI*radio*radio;}
    }

    class Ejherencia {
        static public void main(String arg []) {
            Figura [] f=new Figura[2];
            F[0]=new Cuadrado(12);
            F[1]=new Circulo(7);
            For(int i=0;i<2;i++) {
                System.out.println("Figura "+i);
                System.out.println("\t"+f[i].perimetro());
                System.out.println("\t"+f[i].area());
            }
        }
    }
```

Ejemplo:

```
abstract class Alumno extends Persona {
   protected int curso;
   private String nivelAcademico;
   public Alumno (String n, int e, int c, String nivel) {
        super(n, e);
        curso = c; nivelAcademico = nivel;
   }
   public String toString() {
        return super.toString() + curso + nivelAcademico;
   }
   abstract double pagoMensual();
   abstract String getAsignaturas();
}


class Libre extends Alumno {
   private String []listaDeAsignaturas;
   private static float precioPorHora=10;
   private int noHorasDiarias;

   public Libre(String n, int e, int c, String nivel, int horas)
     {super(n,e,c,nivel); noHorasDiarias = horas; pedirAsignaturas(); }


   private void pedirAsignaturas() {}// se inicializa listaDeAsignaturas

   public double pagoMensual() {
        return precioPorHora*noHorasDiarias*30; }

   public String getAsignaturas() {
         String asignaturas="";
         for (int i=0; i<listaDeAsignaturas.length; i++)
               asignaturas += listaDeAsignaturas[i] + ' ';
         return asignaturas;
   }
}

class Presencial extends Alumno {
    private double matriculaCurso;
    private double plusPorConvocatoria;
    private int noConvocatoria;
```

```
    public Presencial(String n, int e, int c, String nivel,
                double mc, double pc, int nc) {
            super(n,e,c,nivel);
            matriculaCurso=mc;
            plusPorConvocatoria=pc;
            noConvocatoria=nc;
     }

     public double pagoMensual()
     { return (matriculaCurso+plusPorConvocatoria*noConvocatoria)/12; }

     public String getAsignaturas(){
            return "todas las del curso " + curso;
     }
}
```

3. CLASES Y MÉTODOS FINALES

Recuérdese que las ***variables*** declaradas como ***final*** no pueden cambiar su valor una vez que han sido inicializadas. En este apartado se van a presentar otros dos usos de la palabra ***final***.

Una ***clase*** declarada ***final*** no puede tener clases derivadas. Esto se puede hacer por motivos de ***seguridad*** y también por motivos de ***eficiencia***, porque cuando el compilador sabe que los métodos no van a ser redefinidos puede hacer optimizaciones adicionales.

Análogamente, un ***método*** declarado como ***final*** no puede ser redefinido por una clase que derive de su propia clase.

4. INTERFACES

4.1. Concepto de interface.

Una ***interface*** es un conjunto de ***declaraciones de métodos*** (sin ***definición***). También puede definir ***constantes***, que son implícitamente ***public***, ***static*** y ***final***, y deben siempre inicializarse en la declaración. Estos métodos definen un *tipo de conducta*. Todas las clases que implementan una determinada ***interface*** están obligadas a proporcionar una definición de los métodos de la ***interface***, y en ese sentido adquieren una ***conducta*** o ***modo de funcionamiento***.

Una ***clase*** puede ***implementar*** una o varias ***interfaces***. Para indicar que una clase implementa una o más interfaces se ponen los nombres de las ***interfaces***, separados por comas, detrás de la palabra ***implements***, que a su vez va siempre a la derecha del nombre de la clase o del nombre de la super-clase en el caso de herencia. Por ejemplo,

```
public class CirculoGrafico  extends Circulo
                             implements Dibujable, Cloneable {
        ...
}
```

En algunos aspectos los nombres de las ***interfaces*** pueden utilizarse en lugar de las ***clases***. Por ejemplo, las ***interfaces*** sirven para definir ***referencias*** a cualquier objeto de las clases que implementan esa ***interface***. Con ese nombre o referencia, sin embargo, sólo se pueden utilizar los métodos de la interface. Este es un aspecto importante del ***polimorfismo***.

Una ***interface*** puede derivar de otra o incluso de varias ***interfaces***, en cuyo caso incorpora las declaraciones de todos los métodos de las ***interfaces*** de las que deriva (a diferencia de las clases, las interfaces de Java si tienen herencia múltiple).

¿Qué diferencia hay entre una ***interface*** y una ***clase abstract***? Ambas tienen en común que pueden contener varias declaraciones de métodos (la clase ***abstract*** puede además definirlos). A pesar de esta semejanza, que hace que en algunas ocasiones se pueda sustituir una por otra, existen también algunas ***diferencias importantes***:

1. Una clase no puede heredar de dos clases ***abstract***, pero sí puede heredar de una clase ***abstract*** e implementar una ***interface***, o bien implementar dos o más ***interfaces***.
2. Una clase no puede heredar métodos -definidos- de una ***interface***, aunque sí ***constantes***.
3. Las ***interfaces*** permiten mucha más flexibilidad para conseguir que dos clases tengan el mismo comportamiento, independientemente de su situación en la jerarquía de clases de ***Java***.
4. Las ***interfaces*** permiten "publicar" el comportamiento de una clase desvelando un mínimo de información.
5. Las ***interfaces*** tienen una ***jerarquía*** propia, independiente y más flexible que la de las clases, ya que tienen permitida la ***herencia múltiple***.
6. De cara al ***polimorfismo,*** las ***referencias*** de un tipo ***interface*** se pueden utilizar de modo similar a las clases ***abstract***.

4.2. Definición de interfaces

Una ***interface*** se define de un modo muy similar a las clases. A modo de ejemplo se reproduce aquí la definición de la interface ***Dibujable***:

```
// fichero Dibujable.java
import java.awt.Graphics;
public interface Dibujable {
        public void setPosicion(double x, double y);
        public void dibujar(Graphics dw);
}
```

Cada ***interface public*** debe ser definida en un fichero ****.java*** con el mismo nombre de la ***interface***. Los ***nombres de las interfaces*** suelen comenzar también con ***mayúscula***.

Las ***interfaces*** no admiten más que los modificadores de acceso ***public*** y ***package***. Si la ***interface*** no es ***public*** no será accesible desde fuera del ***package*** (tendrá la accesibilidad por defecto, que es ***package***). Los métodos declarados en una ***interface*** son siempre ***public*** y ***abstract***, de modo implícito.

4.3. Herencia en interfaces.

Entre las ***interfaces*** existe una ***jerarquía*** (independiente de la de las clases) que permite ***herencia simple y múltiple***. Cuando una ***interface*** deriva de otra, incluye todas sus constantes y declaraciones de métodos.

Una ***interface*** puede derivar de varias ***interfaces***. Para la herencia de ***interfaces*** se utiliza asimismo la palabra ***extends***, seguida por el nombre de las ***interfaces*** de las que deriva, separadas por comas.

Una ***interface*** puede ocultar una constante definida en una ***super-interface*** definiendo otra constante con el mismo nombre. De la misma forma puede ocultar, re-declarándolo de nuevo, la declaración de un método heredado de una ***super-interface***.

Las ***interfaces*** no deberían ser modificadas más que en caso de extrema necesidad. Si se modifican, por ejemplo añadiendo alguna nueva declaración de un método, las clases que hayan implementado dicha ***interface*** dejarán de funcionar, a menos que implementen el nuevo método.

4.4. Utilización de interfaces.

Las ***constantes*** definidas en una ***interface*** se pueden utilizar en cualquier clase (aunque no implemente la ***interface***) precediéndolas del nombre de la ***interface***, como por ejemplo (suponiendo que PI hubiera sido definida en ***Dibujable***):

```
area = 2.0*Dibujable.PI*r;
```

Sin embargo, en las clases que ***implementan*** la ***interface*** las constantes se pueden utilizar directamente, como si fueran constantes de la clase. A veces se crean interfaces para agrupar constantes simbólicas relacionadas (en este sentido pueden en parte suplir las variables ***enum*** de C/C++).

De cara al ***polimorfismo***, el nombre de una ***interface*** se puede utilizar como un ***nuevo tipo de referencia***. En este sentido, el nombre de una interface puede ser utilizado en lugar del nombre de cualquier clase que la implemente, aunque su uso estará restringido a los métodos de la ***interface***. Un objeto de ese tipo puede también ser utilizado como valor de retorno o como argumento de un método.

Ejemplo:
```
interface Agrandable {
        void zoom(int i);
}
class Circulo extends Figura implements Agrandable {
        ........
        public void zoom(int i) {
                radio*=i;  }
        .....
}
class Punto implements Agrandable {
        .........
        public void zoom(int i) {
                x*=i;
                y*=i;  }
        ..........
}
class EjInterfaz {
        static public void main(String arg []) {
                Agrandable [] f=new Agrandable [2];
                f[0]=new Circulo(12);
                f[1]=new Punto(7, 4);
                for(int i=0;i<2;i++)
                        f[i].zoom(3);
        }
}
```

5. PERMISOS DE ACCESO EN JAVA.

Una de las características de la *Programación Orientada a Objetos* es la ***encapsulación***, que consiste básicamente en ocultar la información que no es pertinente o necesaria para realizar una determinada tarea. Los permisos de acceso de ***Java*** son una de las herramientas para conseguir esta finalidad.

5.1. Accesibilidad de los packages.

El primer tipo de accesibilidad hace referencia a la conexión física de los ordenadores y a los permisos de acceso entre ellos y en sus directorios y ficheros. En este sentido, un ***package*** es accesible si sus directorios y ficheros son accesibles (si están en un ordenador accesible y se tiene permiso de lectura). Además de la propia conexión física, serán accesibles aquellos ***packages*** que se encuentren en la variable ***CLASSPATH*** del sistema.

5.2. Accesibilidad de clases o interfaces.

En principio, cualquier ***clase*** o i***nterface*** de un ***package*** es accesible para todas las demás clases del ***package***, tanto si es ***public*** como si no lo es. Una clase ***public*** es accesible para cualquier otra clase siempre que su ***package*** sea accesible. Recuérdese que las ***clases*** e ***interfaces*** sólo pueden ser ***public*** o ***package*** (la opción por defecto cuando no se pone ningún modificador).

5.3. Accesibilidad de las variables y métodos miembros de una clase.

Desde dentro de la propia clase:

1. Todos los miembros de una clase son directamente accesibles (sin cualificar con ningún nombre o cualificando con la referencia ***this***) desde dentro de la propia clase. Los métodos no necesitan que las variables miembro sean pasadas como argumento.
2. Los miembros ***private*** de una clase sólo son accesibles para la propia clase.
3. Si el ***constructor*** de una clase es ***private***, sólo un método ***static*** de la propia clase puede crear objetos.

Desde una ***sub-clase***:

1. Las ***sub-clases*** heredan los miembros ***private*** de su ***super-clase***, pero sólo pueden acceder a ellos a través de métodos ***public***, ***protected*** o ***package*** de la super-clase.

Desde otras clases del ***package***:

1. Desde una clase de un ***package*** se tiene acceso a todos los miembros que no sean ***private*** de las demás clases del ***package***.

Desde otras clases fuera del ***package***:

1. Los métodos y variables son accesibles si la clase es ***public*** y el miembro es ***public***.
2. También son accesibles si la clase que accede es una ***sub-clase*** y el miembro es ***protected***.

La siguiente tabla muestra un resumen de los permisos de acceso de **Java**.

Visibilidad	public	protected	private	default
Desde la propia clase	SI	SI	SI	SI
Desde otra clase en el propio package	SI	SI	NO	SI
Desde otra clase fuera del package	SI	NO	NO	NO
Desde una sub-clase en el propio package	SI	SI	NO	SI
Desde una sub-clase fuera del propio package	SI	SI	NO	NO

En el anexo I del tema se puede consultar más información sobre el control de acceso.

6. POLIMORFISMO

El ***polimorfismo*** tiene que ver con la relación que se establece entre la llamada a un método y el código que efectivamente se asocia con dicha llamada. A esta relación se llama ***vinculación*** (*binding*). La ***vinculación*** puede ser ***temprana o estática*** (en tiempo de compilación) o ***tardía o dinámica*** (en tiempo de ejecución).

La *vinculación temprana* se realiza mediante funciones normales o sobrecargadas. En el *polimorfismo por sobrecarga* los nombres de las funciones miembro de una clase pueden repetirse si varía el número o el tipo de los argumentos.

Ejemplo: Polimorfismo por vinculación temprana (por sobrecarga de funciones).

```
// fichero punto.java
import java.lang.Math.*;
class Punto {
      double x, y;
      Punto(double x, double y) {this.x=x; this.y=y;}
      void valorInicial(double vx, double vy) {
            x=vx;
            y=vy;
      }
      double distancia(Punto q) {
            double dx=x-q.x;
            double dy=y-q.y;
            return Math.sqrt(dx*dx+dy*dy);
      }
      double distancia(double cx, double cy) {
            double dx=x-cx;
            double dy=y-cy;
            return Math.sqrt(dx*dx+dy*dy);
      }
}
```

Con funciones redefinidas en ***Java*** se utiliza siempre ***vinculación tardía***, excepto si el método es ***final***. La ***vinculación tardía*** hace posible que, con un método declarado en una clase base (o en una interface) y redefinido en las clases derivadas (o en clases que implementan esa interface), sea ***el tipo de objeto*** y ***no el tipo de la referencia*** lo que determine qué definición del método se va a utilizar. El tipo del objeto al que apunta una referencia sólo puede conocerse en tiempo de ejecución, y por eso el ***polimorfismo*** necesita evaluación tardía.

El ***polimorfismo*** permite a los programadores separar las cosas que cambian de las que no cambian, y de esta manera hacer más fácil la ampliación, el mantenimiento y la reutilización de los programas.

El ***polimorfismo*** puede hacerse con referencias de ***super-clases abstract***, ***super-clases normales*** e ***interfaces***. Por su mayor flexibilidad y por su independencia de la jerarquía de clases estándar, las ***interfaces*** permiten ampliar muchísimo las posibilidades del polimorfismo.

Ejemplo: Polimorfismo por vinculación tardía (vinculación dinámica).

```
class PruebaPixel2
{
      public static void main(String arg [])
      {
            Punto x=new Pixel(4, 3, (byte)2);
            Punto p=new Punto(4, 3);
            double d1, d2;
            d1=x.distancia(1, 1);       // método de Pixel
            d2=p.distancia(1, 1);       // método de Punto
            System.out.println("Distancia pixel: " + d1);
            System.out.println("Distancia punto: ") + d1);
      }
}
```

Si se desea acceder a algún método oculto de la clase base se ha de utilizar *super*.

Ejemplo: Acceso al método oculto de la clase base.

```
class Punto {
      .......
      void trasladar(double cx, double cy) {
            x+=cx;
            y+=cy;
      }
}
```

```
class Pixel extends Punto {
      ........
      void trasladar(double cx, double cy) {
            super.trasladar(cx, cy);
            color=(byte)0;
      }
}
```

6.1. Conversión de objetos

El ***polimorfismo*** visto previamente está basado en utilizar referencias de un tipo más "amplio" que los objetos a los que apuntan. Las ventajas del polimorfismo son evidentes, pero hay una importante limitación: el tipo de la referencia (clase abstracta, clase base o interface) limita los métodos que se pueden utilizar y las variables miembro a las que se pueden acceder. Por ejemplo, un objeto puede tener una referencia cuyo tipo sea una ***interface***, aunque sólo en el caso en que su ***clase*** o ***una de sus super-clases*** implemente dicha ***interface***. Un objeto cuya referencia es un tipo ***interface*** sólo puede utilizar los métodos definidos en dicha ***interface***. Dicho de otro modo, ese objeto no puede utilizar las variables y los métodos propios de su clase. De esta forma las ***referencias de tipo interface*** definen, limitan y unifican la forma de utilizarse de objetos pertenecientes a clases muy distintas (que implementan dicha interface).

Si se desea utilizar todos los métodos y acceder a todas las variables que la clase de un objeto permite, hay que utilizar un ***cast explícito***, que convierta su referencia más general en la del tipo específico del objeto. De aquí una parte importante del interés del ***cast*** entre objetos (más bien entre referencias, habría que decir).

Para la conversión entre objetos de distintas clases, ***Java*** exige que dichas clases estén relacionadas por ***herencia*** (una deberá ser ***sub-clase*** de la otra). Se realiza una ***conversión implícita*** o ***automática*** de una ***sub-clase*** a una ***super-clase*** siempre que se necesite, ya que el objeto de la ***sub-clase*** siempre tiene toda la información necesaria para ser utilizado en lugar de un objeto de la ***super-clase***. No importa que la ***super-clase*** no sea capaz de contener toda la información de la ***subclase***.

La conversión en sentido contrario -utilizar un objeto de una ***super-clase*** donde se espera encontrar uno de la ***sub-clase***- debe hacerse de modo ***explícito*** y puede producir errores por falta de información o de métodos. Si falta información, se obtiene una ***ClassCastException***.

No se puede acceder a las variables exclusivas de la sub-clase a través de una referencia de la super-clase. Sólo se pueden utilizar los métodos definidos en la super-clase, aunque la definición utilizada para dichos métodos sea la de la sub-clase.

Por ejemplo, supóngase que se crea un objeto de una ***sub-clase B*** y se referencia con un nombre de una ***super-clase A***,

```
A a = new B();
```

en este caso el objeto creado dispone de más información de la que la referencia ***a*** le permite acceder (podría ser, por ejemplo, una nueva variable miembro ***j*** declarada en ***B***). Para acceder a esta información adicional hay que hacer un ***cast*** explícito en la forma ***(B)a***. Para imprimir esa variable ***j*** habría que escribir (los paréntesis son necesarios):

```
System.out.println( ((B)a).j );
```

Un ***cast*** de un objeto a la ***super-clase*** puede permitir utilizar variables -no métodos- de la ***super-clase***, aunque estén redefinidos en la ***sub-clase***. Considérese el siguiente ejemplo: La clase ***C*** deriva de ***B*** y ***B*** deriva de ***A***. Las tres definen una variable ***x***. En este caso, si desde el código de la sub-clase ***C*** se utiliza:

```
x            // se accede a la x de C
this.x       // se accede a la x de C
super.x      // se accede a la x de B. Sólo se puede subir un nivel
((B)this).x  // se accede a la x de B
((A)this).x  // se accede a la x de A
```

7.- RELACIONES ENTRE CLASES

Hasta lo visto en los temas anteriores, se puede entender que el diseño de una aplicación es prácticamente el diseño de una clase. Sin embargo en realidad una aplicación es un conjunto de objetos que se relacionan. Por ello en el diagrama de clases se deben indicar la relación que hay entre las clases. En este sentido el diagrama de clases UML nos ofrece distintas posibilidades.

7.1. Asociaciones

Las asociaciones son relaciones entre clases. Es decir, marcan una comunicación o colaboración entre clases. Dos clases tienen una asociación si:

- Un objeto de una clase envía un mensaje a un objeto de la otra clase. Enviar un mensaje, como ya se comentó, es utilizar alguno de sus métodos o propiedades para que el objeto realice una determinada labor.
- Un objeto de una clase, crea un objeto de otra clase.
- Una clase tiene propiedades cuyos valores son objetos o colecciones de objetos de otra clase
- Un objeto de una clase recibe como parámetros de un método objetos de otra clase.

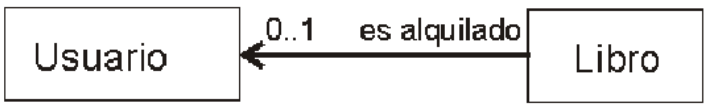
En UML las asociaciones se representan con una línea entre las dos clases relacionadas, encima de la cual se indica el nombre de la asociación y una flecha para indicar el sentido de la asociación. Ejemplo:

Como se observa en el ejemplo la dirección de la flecha es la que indica que es el usuario el que alquila los libros. Los números indican que cada usuario puede alquilar de cero a más (el asterisco significa muchos) libros. Esos números se denominan cardinalidad, e indican con cuántos objetos de la clase se puede relacionar cada objeto de la clase que está en la base de la flecha.

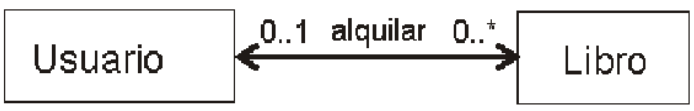
Puede ser:

- 0..1. Significa que se relaciona con uno o ningún objeto de la otra clase.
- 0..*. Se relaciona con cero, uno o más objetos
- 1..*. Se relaciona al menos con uno, pero se puede relacionar con más
- un número concreto. Se puede indicar un número concreto (como 3 por ejemplo) para indicar que se relaciona exactamente con ese número de objetos, ni menos, ni más.

La dirección de la flecha determina de dónde a dónde nos referimos. Así esa misma relación al revés:



Por eso se suele reflejar así de forma completa:



En muchos casos en las asociaciones no se indica dirección de flecha, se sobreentenderá que la asociación va en las dos direcciones.

Las asociaciones como es lógico implican decisiones en las clases. La clase usuario tendrá una estructura que permita saber qué libros ha alquilado. Y el libro tendrá al menos una propiedad para saber qué usuario le ha alquilado. Además de métodos para relacionar los usuarios y los libros.

Normalmente la solución a la hora de implementar es que las clases incorporen una propiedad que permita relacionar cada objeto con la otra clase. Por ejemplo si la clase usuario Alquila cero o un libro:

```
public class Usuario{
  …
  Libro libro; //representa la relación Usuario->Libro con
      //cardinalidad 0..1 o  1
  …
```

Con una cardinalidad fija pero mayor de uno (por ejemplo un usuario siempre alquila 3 libros):

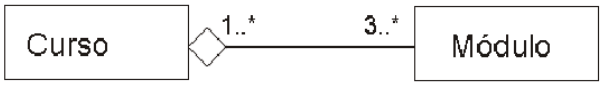
```
public class Usuario{
  …
  Libro libro[]; //representa la relación Usuario->Libro con
      //cardinalidad 0..1 o  1
  …
```

Si la cardinalidad es de tamaño indefinido (como 1..* por ejemplo) entonces la propiedad será una colección de libros (en temas posteriores se habla sobre colecciones de datos).

7.2. Agregación y composición

Son asociaciones pero que indican más información que una asociación normal. Definen asociaciones del tipo es parte de o se compone de.

La **agregación** indica que un elemento es parte de otro. Indica una relación en definitiva de composición. Así la clase Curso tendría una relación de composición con la clase Módulo.



Cada curso se compone de tres o más módulos. Cada módulo se relaciona con uno o más cursos.

En Java al final se resuelven como las asociaciones normales, pero el diagrama representa esta connotación importante.

La **composición** indica una agregación fuerte, de hecho significa que una clase consta de objetos de otra clase para funcionar. La diferencia es que cada objeto que compone el objeto grande no puede ser parte de otro objeto, es decir pertenece de forma única a uno.

La existencia del objeto al otro lado del diamante está supeditada al objeto principal y esa es la diferencia con la agregación.

Ejemplo:

En este caso se refleja que un edificio consta de pisos. De hecho con ello lo que se indica es que un piso sólo puede estar en un edificio. La existencia del piso está ligada a la del edificio. Como se ve la asociación es más fuerte.

La implementación en Java de clases con relaciones de agregación y composición es similar. Pero hay un matiz importante. Puesto que en la composición, los objetos que se usan para componer el objeto mayor tienen una existencia ligada al mismo, se deben crear dentro del objeto grande. Por ejemplo (composición):

```
public class Edificio {
  private Piso piso[];
  public Edificio(…..){
    piso=new Piso[x]; //composición
    ……
```

En la composición (como se observa en el ejemplo), la existencia del piso está ligada al edificio por eso los pisos del edificio se deben de crear dentro de la clase Edificio y así cuando un objeto Edificio desaparezca, desaparecerán los pisos del mismo.

Eso no debe ocurrir si la relación es de agregación. Por eso en el caso de los cursos y los módulos, como los módulos no tienen esa dependencia de existencia según el diagrama, seguirán existiendo cuando el módulo desaparezca, por eso se deben declarar fuera de la clase cursos. Es decir, no habrá new para crear módulos en el constructor. Sería algo parecido a esto:

```
public class Cursos {
  private Módulo módulos[];
  public Edificio(….., Módulo m[]){
    módulos=m; //agregación
    ……
```

ANEXO I: CONTROL DE ACCESO

El control de acceso se aplica siempre a nivel de clase, no a nivel de objeto. Es decir, los métodos de instancia de un objeto de una clase determinada tiene acceso directo a los miembros privados de cualquier oto objeto de la misma clase.

Java implementa cuatros tipos de ***especificadores de acceso***: **private, protected, public y package**. Por lo tanto, cuando se crea una nueva clase en Java, se puede especificar el nivel de acceso que se quiere para las variables de instancia y los métodos definidos en la clase. La siguiente tabla muestra el nivel de acceso que está permitido a cada uno de los especificadores:

Nivel de acceso	Clase	Subclase	Paquete	Todos
`private`	X			
`protected`	X	X*	X	
`public`	X	X	X	X
`package`	X		X	

Si se profundiza en el significado de la tabla, se puede observar que la columna ***clase*** indica que todos los métodos de una clase tienen acceso a todos los otros miembros de la misma clase, independientemente del nivel de acceso especificado.

La columna ***subclase*** se aplica a todas las clases heredadas de la clase, independientemente del paquete en que residan. Los miembros de una subclase tienen acceso a todos los miembros de la superclase que se hayan designado como *public*. El asterisco (*) en la intersección *subclase-protected* quiere decir que si una clase es a la vez subclase y está en el mismo paquete que la clase con un miembro *protected*, entonces la clase tiene acceso a ese miembro protegido.

En general, si la subclase no se encuentra en el mismo paquete que la superclase, no tiene acceso a los miembros protegidos de la superclase. Los miembros de una subclase no tienen acceso a los miembros de la superclase catalogados como *private* o *package*, excepto a los miembros de una subclase del mismo paquete, que tienen acceso a los miembros de la superclase designados como *package*.

La columna **paquete** indica que las clases del paquete tienen acceso a los miembros de una clase, independientemente de su árbol de herencia. La tabla indica que todos los miembros *protected*, *public* y *package* de una clase pueden ser accedidos por otra clase que se encuentre en el mismo paquete.

La columna ***todos*** indica que los privilegios de acceso para métodos que no están en la misma clase, ni en una subclase, ni en el mismo paquete, se encuentran restringidas a los miembros públicos de la clase.

Si se observa la misma tabla desde el punto de vista de las filas, se pueden describir los cualificadores de los métodos:

private

Las variables y métodos de instancia privados sólo pueden ser accedidos desde dentro de la clase. No son accesibles desde las subclases de esa clase. Hay que resaltar, una vez más, que un método de instancia de un objeto de una clase puede acceder a todos los miembros privados de ese objeto, o miembros privados de cualquier otro objeto de la misma clase. Es decir, que en Java el control de acceso existe a nivel de clase, pero no a nivel de objeto de la calse.

public

Cualquier clase desde cualquier lugar puede acceder a las variables y métodos de instancia públicos.

protected

Sólo las subclases de la clase y nadie más puede acceder a las variables y métodos de instancia protegidos. Los métodos protegidos puede ser vistos por las clases derivadas y también por los paquetes. Todas las clases de un paquete puede ver los métodos protegidos de ese paquete.

package

Por defecto, si no se especifica el control de acceso, las variables y métodos de instancia se declaran *package*, lo que significa que son accesibles por todos los objetos dentro del mismo paquete, pero no por los externos al paquete. Aparentemente, parece lo mismo que *protected*; la diferencia estriba en la designación de *protected* es heredada por las subclases de un paquete diferente, mientras que la designación *package* no es heredada por subclases de paquetes diferentes.

Debido a la complejidad y posible confusión respecto a los niveles de protección que proporciona Java para permitir el control preciso de la visibilidad de variables y métodos, se puede generar otra tabla en base a cuatro categorías de visibilidad entre los elementos de la clase:

	private	**Sin modificador**	**protected**	**public**
Misma clase	SI	SI	SI	SI
Misma subclase de paquete	NO	SI	SI	SI
Misma no-subclase de paquete	NO	SI	SI	SI
Subclase de diferente paquete	NO	NO	SI	SI
No-subclase de diferente paquete	NO	NO	NO	SI

Y una guía de uso indicaría tener en cuenta lo siguiente:

- ✓ Usar *private* para métodos y variables que solamente se utilicen dentro de la clase y que deberían estar ocultas para el resto.
- ✓ Usar *public* para métodos, constantes y otras variables importantes que deseen ser visibles para todo el mundo.
- ✓ Usar *protected* si se quiere que las clases del mismo paquete puedan tener acceso a estas variables y métodos.
- ✓ Usar la sentencia *package* para poder agrupar las clases en paquetes.
- ✓ No usar nada, dejar la visibilidad por defecto (*package*) para métodos y variables que deban estas ocultas fuera del paquete, pero que deban estar disponibles al acceso desde dentro del mismo paquete. Utilizar *protected* en su lugar si se quiere que esos componentes sean visibles fuera del paquete.