INDICE

1. RELACIONES ENTRE CLASES

Una aplicación es un conjunto de objetos que se relacionan. Por ello en el diagrama de clases se debe indicar la relación que hay entre las clases. En este sentido el diagrama de clases UML nos ofrece distintas posibilidades:

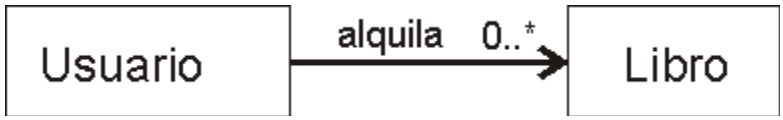
- Asociación
- Agregación
- Composición
- Herencia

ASOCIACIÓN

Marca 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 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:



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.

- **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.

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
             //el usuario solo puede alquilar un libro
…
```

Con una cardinalidad entre 0 e indefinido:

```
public class Usuario{
…
Libro libro[]; //representa la relación Usuario->Libro con cardinalidad 0..*
               //el usuario puede alquilar varios libros
…
```

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.

El diagrama UML de agregación es:

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.

COMPOSICIÓN

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.

El diagrama UML de composición es:

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.

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):

En la composición, 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.

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

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
   ……
```

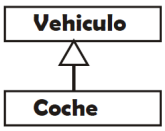
HERENCIA

La herencia define una relación entre clases en la cual una clase posee características (métodos y propiedades) que proceden de otra. Esto permite estructura de forma muy atractiva los programas y reutilizar código de forma más eficiente. Es decir genera relaciones entre clases del tipo es como… (también se las llama **es un** en inglés **is a**).

A la clase que posee las características a heredar se la llama superclase y la clase que las hereda se llama subclase. Una subclase puede incluso ser superclase en otra relación de herencia.

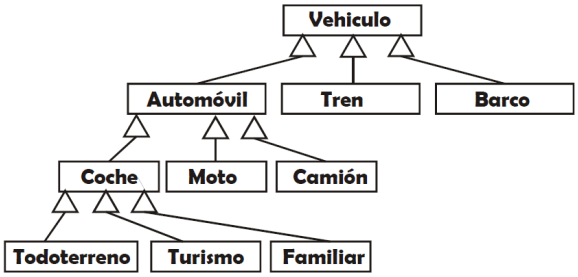
También se emplea habitualmente los términos madre para referirnos a una superclase e hija para una subclase (e incluso abuela y nieta).

El diagrama UML para la herencia es:



El diagrama representa que un Coche es un Vehiculo. Con lo cual los coches tendrán todas las características de los vehículos y además añadirán características particulares.

Puede haber varios niveles de herencia, es decir clases que heredan de otra clase que a su vez es heredera de otra.



2. HERENCIA

La herencia y el polimorfismo son dos de los principios básicos (paradigmas) de la orientación a objetos. Constituyen los mecanismos más potentes de la orientación a objetos.

La herencia se basa en la existencia de relaciones de **especialización / generalización** entre clases:

- ✓ En modelos de representación de la información (como Entidad / Relación) supone relaciones del tipo **"es un"**.
- ✓ Las clases se organizan de manera jerárquica de forma que las clases hijas heredan atributos y métodos de las clases padres. Los métodos (los atributos no) se pueden sobrescribir(override), en la clase derivada. No confundir con sobrecargar (overload)
- ✓ Las clases hijas pueden introducir atributos y métodos propios (como si la subclase "contuviera" un objeto de la superclase). En realidad lo "amplía" con nuevos atributos y métodos.

Tipos de herencia

- ✓ **Herencia simple:** una clase solo puede tener una clase padre
- ✓ **Herencia múltiple:** una clase puede tener varias clases padres. Java no permite herencia múltiple aunque esta puede simularse con la utilización de interfaces (se verá más adelante).

La implementación de la herencia en Java se realiza mediante la palabra reservada **extends** en la definición de la clase.

Sintaxis :

```
[modificador] class NombClase extends NombClasePadre
{
   …//Definición de la clase
}
```

Ejemplo:

```
public class Perro extends Mamifero
{
   …//Definición de la clase
}
```

- ✓ Una clase puede ser superclase y subclase al mismo tiempo.
- ✓ Una subclase hereda de su superclase todos los atributos y métodos, tanto de clase (**static**) como de objeto.
- ✓ Una subclase siempre puede añadir nuevos atributos y/o métodos.
- ✓ Una subclase puede sobrescribir métodos heredados siempre que no estén declarados como **final** en la superclase correspondiente.

Las subclases se pueden **sobrescribir** (override) los métodos que heredan. Para sobrescribir un método hay que respetar totalmente la declaración del método:

- ✓ El nombre ha de ser el mismo
- ✓ Los parámetros han de ser los mismo
- ✓ Los métodos sobrescritos (o redefinidos) **pueden ampliar los derechos de acceso** de la superclase (por ejemplo ser public, en vez de protected o package), **pero nunca restringirlos**.
- ✓ Los **atributos no** se pueden **sobrescribir** (redefinir).

Al ejecutar un método se busca su implementación de abajo hacia arriba en la jerarquía de clases.

Sobrescritura de métodos

- ✓ Los métodos de la superclase que han sido sobrescritos, todavía pueden ser accedidos por medio de la palabra **super** (se explica posteriormente).
- ✓ Métodos **private** o **static** →no pueden sobrescribirse.
- ✓ Métodos **final** →no pueden sobrescribirse.
- ✓ Métodos **abstract** →obligatoriamente se tienen que sobrescribir si pertenecen a una clase abstracta.

Sobrescritura y Sobrecarga

No se debe confundir la **sobrescritura** con la **sobrecarga** de métodos. Sobrescribir (override) un método es un concepto distinto que sobrecargar un método.

La sobrecarga (overload) de un método significa tener varias implementaciones del mismo método con parámetros distintos:

- ✓ El nombre ha de ser el mismo.
- ✓ Los tipo y/o número de parámetros tienen que ser distintos.
- ✓ El tipo de retorno puede ser distinto.
- ✓ El modificador de acceso puede ser distinto.

Sobrescribir un método es dar una nueva definición. En este caso el método debe tener exactamente los mismos argumentos en tipo y número que el método redefinido.

La Clase Object

En Java todas las clases heredan por defecto de la clase **Object** (definida en el paquete **java.lang**) siendo por tanto "la madre de todas las clases". Así todas las clases que construyamos dispondrán de los atributos y métodos de la clase Object.

Algunos métodos importantes son:

- ✓ **public boolean equals(Object o):** compara dos objetos y decide si "son iguales". Realmente devuelve true si las referencias de los objetos pasados al método es la misma. Es decir devuelve true sólo si ambas referencias apuntan al mismo objeto.
- ✓ **public String toString():** retorna un String que contiene una representación del objeto (entre otras cosas el nombre de la clase a la que pertenece el objeto).Conviene sobrescribirlo.

Las Referencias this y super

La palabra reservada **this** hace referencia a la clase actual. Un objeto puede hacer referencia a si mismo utilizando **this**.

La palabra reservada **super** hace referencia a la superclase respecto a la clase actual. La palabra reservada **super** permite acceder a métodos anulados por herencia.

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.

Existen dos ocasiones en las que su uso no es redundante sino imprescindible:

- ✓ Acceso a un constructor desde otro constructor.
- ✓ Acceso a un atributo desde un método donde hay definida una variable local con el mismo nombre que el atributo.

La palabra reservada **super** es una referencia al objeto actual pero apuntando al padre. Se utiliza para acceder a métodos definidos en la clase padre (incluyendo a los constructores). Cuando el atributo o método al que se hace referencia no ha sido sobrescrito en la subclase, el uso de super es redundante.

Constructores de clases heredadas

Los constructores no se heredan ni se sobrescriben.

El constructor de una clase que hereda de otra puede llamar al constructor de su superclase por medio de la palabra **super(),** seguida entre paréntesis de los argumentos apropiados para alguno de los constructores de la superclase.

De esta forma, un constructor sólo tiene que inicializar directamente las variables no heredadas.

La llamada al constructor de la superclase 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 superclase: super().

Esta llamada en cadena a los constructores de las superclases llega hasta el origen de la jerarquía de clases, esto es, al constructor de Object.

Como ya se ha dicho, si el programador no programa un constructor por defecto, el compilador crea uno, inicializando las variables de los tipos primitivos a sus valores por defecto, y los Strings y demás referencias a objetos a **null**. Y en primer lugar, incluirá una llamada al constructor de la superclase.

Invocación automática implícita

- ✓ Se invoca al constructor de la clase padre sin argumentos
- ✓ Si no está definido → error

Invocación explícita

- ✓ Invocación a super(...) en la primera línea del constructor

Invocación a otros constructores de la misma clase:

- ✓ this (...)

EJEMPLOS

Se ha definido la clase Persona con las siguientes características mínimas:

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

        Persona (String nom, int ed)
        {
            nombre = nom;
            edad = ed;
        }
        String descripcion ()
        {
            ...
        }
}
```

Se define la clase empleado que hereda de la clase Persona:

```
class Empleado extends Persona
{
        long sueldoBruto;
        Empleado (String nom, int ed, long sueldo)
        {
                nombre = nom;
                edad = ed;
                sueldoBruto = sueldo;
        }
        String descripcion ()
        {
                ...
        }
}
```

Error al crear un Empleado: se invoca automáticamente a super() (que sería Persona()) y resulta que este constructor no está definido (está definido otro con dos parámetros)

Un empleado tiene un **nombre** y una **edad** que no es necesario definirlos, ya están definidos en su superclase.

Tan solo es necesario definir el atributo propio de la clase Empleado **sueldoBruto**.

Para establecer el nombre y la edad del nuevo empleado es necesario llamar al constructor de la superclase con la palabra reservada **super(nom,ed)**

```
class Empleado extends Persona
{
        long sueldoBruto;
        Empleado (String nom, int ed, long sueldo)
        {
            super (nom, ed)
            sueldoBruto = sueldo;
        }
        String descripcion ()
        {
            ...
        }
}
```

Ahora funcionaría correctamente

A continuación se define la clase **Directivo** que hereda de la clase Empleado, por lo que de forma indirecta hereda de la clase **Persona**.

Cuando creamos un objeto **Directivo** debemos pasarle los atributos que tiene por ser una **Persona**, los que tiene por ser un **Empleado** y los que tiene por ser un **Directivo**.

```
class Directivo extends Empleado
{       int categoria;
        Directivo (String nom, int ed, long sueldo, int cat)
        {
            super (nom, ed, sueldo);
            categoria = cat;
        }
        Directivo (String nom, int ed, long sueldo)
        {
            this (nom, ed, sueldo, 0);          // asignará categoría 0
        }

        String descripcion ()
        {
        ...
        }
}       /* se producen llamadas encadenadas a 'super' que van subiendo por la
        jerarquía de clases desde Directivo hasta Persona */
```

Siempre en la primea línea

Modificadores de acceso de una Clase (Recordatorio)

public

- ✓ La clase es accesible desde cualquier lugar del programa.
- ✓ Si no aparece solo puede referenciarse desde el directorio o paquete donde esta declarada (se explica el concepto de paquete en el tema 8).

abstract

- ✓ Una clase abstracta es una clase de la que no se pueden crear objetos. Su utilidad es permitir que otras clases deriven de ella.
- ✓ Cuando la clase puede tener métodos declarados como abstract, en cuyo caso no se da definición del método.

final

- ✓ No pueden crearse subclases de esa clase
- ✓ Definiendo una clase como final conseguimos que ninguna otra clase pueda heredar de ella.

Modificadores de acceso de los atributos (recordatorio):

public

- ✓ Cuando se declara un atributo como público se está permitiendo que desde cualquier otra clase pueda referenciarse directamente.

private

- ✓ Si se declara un atributo como privado no se podrá acceder directamente a él desde otra clase distinta, sino que será necesario recurrir a algún método que proporcione o modifique su valor.

protected (Es el modificador por defecto si no se especifica)

- ✓ Los atributos de declarados así son accesibles directamente sólo desde la propia clase, de las subclases de está(herencia)y las clases que se encuentran dentro del mismo paquete.

final

- ✓ Un atributo definido como final una vez inicializado no puede cambiar su valor, es decir se comporta como una constante.

static

- ✓ Los atributos definidos como **static** son atributos cuyo valor es compartido por todos los objetos de la clase. A las variables de este tipo se les denomina atributos de clase.

Estos dos últimos modificadores se pueden combinar entre sí y con los tres primeros modificadores de acceso.

Modificadores de acceso de los métodos (recordatorio):

public

- ✓ Cuando se declara un método como publico se está permitiendo que desde cualquier otra clase pueda referenciarse directamente.

private

- ✓ Si se declara un método como privado no podrá ser invocado desde ninguna otra clase (se emplean para realizar labores auxiliares dentro de una clase que no es necesario ver desde fuera) .

protected (es el modificador por defecto si no se especifica)

- ✓ Los métodos así declarados son accesibles directamente sólo desde la propia clase, de las subclases de está (herencia) las clases que se encuentran dentro del mismo paquete.

static

- ✓ Los métodos definidos como static son métodos que se pueden ejecutar sin crear un objeto de la clase. Se invocan anteponiendo el nombre de la clase no el del objeto. No dependen de ningún objeto en particular. Se denominan métodos de clase.

final

- ✓ Un método declarado como final no puede ser sobrescrito por las subclases de la clase a la que pertenece.

abstract

- ✓ Define un método abstracto (se explica posteriormente)

EJEMPLOS

Modificador de acceso: private

```
class Persona
{
    private String nombre;
    private int edad;
    String descripcion ()
    {   return "Nombre: " + nombre + "\nEdad: " + edad;
    }
}
class Empleado extends Persona
{   long sueldoBruto;
    String descripcion ()
    {
        return "Nombre: " + nombre + "\nEdad: " + edad + "\nSueldo: " +
                sueldoBruto;
    }
}
```

Error: nombre y edad son privados

Modificador de acceso: private (solución)

```
class Empleado extends Persona
{
    Long sueldoBruto;
    String descripcion ()
    {
        return "Nombre: " + getNombre() + "\nEdad: " + getEdad() +
                "\nSueldo: " + sueldoBruto;
    }
}
```

Métodos que tendrán que estar definidos en la superclase Persona (con modificador protected como poco)

Modificador de acceso: protected

```
class Persona
{
      protected String nombre;
      protected int edad;
      protected String descripcion () {
            return "Nombre: " + nombre + "\nEdad: " + edad;
      }
}
class Empleado extends Persona    Error : sólo puede ser protected o public
 {
      long sueldoBruto;
      private String descripcion () {
            return super.descripcion () + "\nSueldo: " + sueldoBruto;
      }
}
```

Los modificadores de acceso no pueden ser más restrictivos en la clase hija que en la clase padre.

3. CLASES ABSTRACTAS

A menudo existen clases de las que no tiene sentido crear objetos. Estas clases pueden usarse para agrupar otras clases (que heredan de ella) para contener código reutilizable, etc.

En ellas se definen un conjunto de métodos y atributos que permitan modelar un cierto concepto, que será refinado mediante la herencia.

Las clases abstractas son posibles gracias al mecanismo de la herencia. 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**

```
public abstract class NombreClase
{
    //Atributos
    //Métodos
}
```

Una clase **abstract** puede tener métodos declarados como **abstract** (métodos abstractos) o no tenerlos. Pero si la clase tiene un método **abstract** la clase debe ser declarada como **abstract**.

Métodos abstractos

- ✓ Sólo cuentan con la declaración y no poseen cuerpo de definición
- ✓ La implementación es específica de cada subclase
- ✓ Si una clase tiene algún método **abstract** es obligatorio que la clase sea **abstract**
- ✓ En cualquier subclase este método deberá bien ser redefinido, bien volver a declararse como **abstract** (el método y la subclase).
- ✓ Como los métodos **static** y los final no pueden ser redefinidos, un método **abstract** no puede ser **static** ni **final**.
- ✓ Una clase **abstract** puede tener métodos que no son **abstract**.
- ✓ Si una subclase no implementa un método abstracto heredado, debe ser abstracta también.
- ✓ No se puede hacer un new (es decir crear objetos de las clases) de una clase abstracta.
- ✓ Pero sí se pueden definir referencias cuyo tipo sea una clase abstracta.

Ejemplo:

Si Figura es una clase abstracta:

```
Figura v[]=new Figura[10];// funciona, es una referencia a un array
Figura v=new Figura(); // no funciona, es una creación de objeto
```

Ejemplos de definición de un método abstracto

```
public abstract void dibujar();
public abstract int perimetro();
```

Reglas de Herencia

Se heredan todos los atributos y métodos, aunque sólo son accesibles los declarados **public** o **protected**, en caso de no tener calificador de acceso (protected por defecto) es posible el acceso si la subclase se declara en el mismo directorio o paquete.

Los atributos y métodos declarados **private** se heredan aunque sin posibilidad de acceso dentro de la implementación de la clase.

- ✓ Las clases con el modificador **final** no pueden tener subclases.
- ✓ Una subclase puede sobrescribir un método de la superclase a efectos de especializar ésta.
- ✓ Método sobrescrito →sustituye al método heredado, mismo nombre, parámetros y tipo devuelto
- ✓ Método sobrecargado →mismo nombre y distinto tipo y/o número de argumentos o tipo de retorno
- ✓ Métodos **private** o **static** →no pueden sobrescribirse
- ✓ Métodos **final** →tampoco pueden sobrescribirse
- ✓ Métodos **abstract** →obligatoriamente sobrescritos pueden ser extendidos (sobrescritos en clases hijas).

Formas de Llamar a los Métodos:

En la misma clase o subclases (acceso protected)

- ✓ idMétodo(parámetros);
- ✓ Ejemplo :mover(x,y);
- ✓ Dentro de la misma clase (métodos sobrescritos de superclases)
  - super.idMétodo(parámetros); // llama al método de la clase padre

Fuera de la clase (acceso public)

- ✓ idobjeto.idMétodo(parámetros);
- ✓ Ejemplo: rectangulo1.mover(x,y)

Fuera de la clase (métodos static)

- ✓ idClase.idMétodo(parámetros);
- ✓ Ejemplo: Math.random();

4. POLIMOSRFISMO

Es otro de los paradigmas de la programación orientada a objetos. **Polimorfismo** hace referencia a poder utilizar un mismo elemento de distintas formas según la situación.

Java proporciona diversas formas de manejar el polimorfismo.

- ✓ La sobrecarga de métodos es un tipo de polimorfismo.
- ✓ Como consecuencia de la herencia y para aprovechar al máximo sus posibilidades, Java ofrece otro tipo de polimorfismo relacionado con la utilización de objetos de una subclase mediante una referencia declarada como superclase.

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.

En función del tipo de información que contienen en Java se pueden distinguir dos tipos de variables:

- ✓ **Variables de tipos primitivos.** Las variables de tipo primitivo contienen el dato directamente.
- ✓ **Variables referencia.** Referencias a arrays u objetos de una determinada clase. contienen una referencia (puntero) a la zona de memoria donde está el objeto.

A diferencia de C/C++, Java no permite acceder al valor de la dirección (pues en el lenguaje se han eliminado los punteros).

Al declarar una referencia todavía no se encuentra "**apuntando**" a ningún objeto por eso se la asigna el valor **null**. Es el operador **new** al llamar al constructor el que reserva memoria para el objeto.

Cuando se sobrecarga un método se está empleando el concepto de polimorfismo. La ejecución del método dependerá del numero y del tipo de los parámetros que reciba.

Es posible referenciar a un objeto de una subclase mediante una referencia declarada como una de sus superclases.

Aclaración de términos (tipos a nivel del compilador):

- ✓ **Tipo de una referencia:** tipo (clase) de la variable con que se declara la referencia. Una referencia de un tipo determinado (clase) puede referenciar a objetos de tipos (subclases) que hereden del mismo.
- ✓ **Tipo de un objeto:** es el tipo (clase) del que se crea el objeto al hacer el new y llamar al constructor

Que la referencia de un objeto sea de otro tipo (clase) no significa que los métodos que se ejecuten sean otros.

- ✓ En tiempo de ejecución se vincula el código del método que se va a ejecutar al que corresponde con el tipo (clase) del objeto(que además solo se puede conocer en tiempo de ejecución) y no con el de su referencia (la vinculación en tiempo de ejecución se denomina ligadura dinámica).
- ✓ Los métodos estáticos (static) tienen ligadura estática.

Existe una limitación en polimorfismo visto anteriormente: ***el tipo de referencia limita los métodos y atributos que se pueden utilizar y acceder.***

Si se desea utilizar todos los métodos y acceder a todas los atributos 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.

El **cast** consiste en convertir una variable de un tipo origen en un tipo de destino, con el fin de poder utilizarla en puntos del programa que necesitan una variable declarada como perteneciente al tipo de destino.

El casting sólo puede realizarse entre tipos primitivos o en una misma jerarquía de herencia.

- ✓ Para la conversión entre objetos de distintas clases, Java exige que dichas clases estén relacionadas por herencia (una deberá ser subclase de la otra).
- ✓ Se realiza una conversión implícita o automática de una subclase a una superclase siempre que se necesite, ya que el objeto de la subclase siempre tiene toda la información necesaria para ser utilizado en lugar de un objeto de la superclase. No importa que la superclase no sea capaz de contener toda la información de la subclase.
- ✓ La conversión en sentido contrario (utilizar un objeto de una superclase donde se espera encontrar uno de la subclase)
- ✓ En Java existe una palabra reservada denominada **instanceof** que informa si un objeto (no la referencia al objeto) es de una clase o hereda de ella. debe hacerse de modo explícito y puede producir errores por falta de información o de métodos (responsabilidad del programador).

Java proporciona diversas formas de manejar el polimorfismo.

- ✓ Sobrecarga de métodos.
- ✓ Utilización de objetos de una subclase mediante una referencia declarada como superclase.

Ligadura dinámica. En tiempo de ejecución se vincula el código del método que se va a ejecutar al que corresponde con el tipo del objeto (que además solo se puede conocer en tiempo de ejecución) y no con el de su referencia.