# Programación orientada a objetos

Resumen de Temas

Unidad 6: Polimorfismo y reutilización

# 6.1 Concepto de Polimorfismo

- Polimorfismo quiere decir "un objeto y muchas formas". Esta propiedad permite que un objeto presente diferentes comportamientos en función del contexto en que se encuentre. Por ejemplo un método puede presentar diferentes implementaciones en función de los argumentos que recibe, recibir diferentes números de parámetros para realizar una misma operación, y realizar diferentes acciones dependiendo del nivel de abstracción en que sea llamado.
- Los objetos de distintas clases pertenecientes a una misma jerarquía o que implementan una misma interface pueden tratarse de una forma general e individualizada, al mismo tiempo.
- Una referencia a un objeto de una determinada clase es capaz de servir de referencia o de nombre a objetos de cualquiera de sus clases derivadas.
- El poder utilizar nombres de una super-clase o de una interface permite tratar de un modo unificado objetos distintos, aunque pertenecientes a distintas sub-clases o bien a clases que implementan dicha interface.

# 6.1.1 Vinculación temprana y tardía

- 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 (en tiempo de compilación) o tardía (en tiempo de ejecución).
- Con funciones normales o sobrecargadas se utiliza vinculación temprana (es posible y es lo más eficiente).
- Con funciones redefinidas en Java se utiliza siempre vinculación tardía, excepto si el método es final. El polimorfismo es la opción por defecto en Java.
- 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.

# 6.1.2 Ventajas y desventajas del polimorfismo

- 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.
- El polimorfismo 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.

# 6.1.3 Conversión de objetos$_1$

- 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.

# 6.1.3 Conversión de objetos$_2$

- 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 );**

# 6.1.3 Conversión de objetos$_3$

- 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
```

# Polimorfismo

- El poder de la POO se logra a través del polimorfismo
- Digamos que queremos trabajar con un lista de usuarios de correo del departamento de Sistemas
- Hemos definido una clase denominada CuentaUsuario con dos subclases
  - CuentaUsuarioAlumno
  - CuentaUsuarioDocente
- El polimorfismo nos permite trabajar con una lista de CuentaUsuario sin saber o importarnos de que tipo de cuenta se trata

# Polimorfismo

- En Java podemos asignar a una variable de un cierto tipo de clase una instancia de esa clase particular o una instancia de cualquier subclase de la clase

```
CuentaUsuario miCuenta = new CuentaUsuarioAlumno(...);
```

- Ahora la variable MiCuenta se ve como un objeto CuentaUsuario y puede ser tratada de esa manera aunque internamente se trate de un objeto CuentaUsuarioAlumno
- Podemos hacer una conversión de regreso a CuentaUsuarioAlumno si alguna vez necesitamos la funcionalidad especial de la clase CuentaUsuarioAlumno

```
CuentaUsuarioAlumno miCuentaEstudiante=
(CuentaUsuarioAlumno)miCuenta;
```

# Chequeo de Tipos Dinámico!

- Java tiene soporte para un chequeo de tipos dinámico el cual es el verdadero poder del polimorfismo
- Ejemplo:
  - CuentaUsuario tiene un método denominado obtenNombreCompleto() que regresa el nombre completo del usuario
  - CuentaUsuarioAlumno y CuentaUsuarioDocente superponen cada una el método obtenNombreCompleto() con su propia función especializada
  - Una variable CuentaUsuario instanciada con un objeto CuentaUsuarioAlumno llama al método obtenNombreCompleto()
  - En tiempo de ejecución, Java checa el tipo de este objeto y ve que es CuentaUsuarioAlumno y por lo tanto llama a la version CuentaUsuarioAlumno del método obtenNombreCompleto() en vez de la version CuentaUsuario

# 6.2 Clases Abstractas

- 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 Persona { ... }**

- 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 ser redefinido o 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***.

# Clases abstractas

- Se puede declarar una clase abstracta con la palabra reservada **abstract**

```
public abstract class FormaGeometrica {
    final double PI = 3.14;
    public String toString() {
        return "Esta es una forma ";
    }
    public abstract double obtenArea();
    public abstract double obtenPerimetro();
}
```

- Las clases abstractas proporcionan un prototipo pero no una implementación de algunos de sus métodos porque el contexto de la implementación en solo importante en las subclases

# Clases abstractas

- Su propósito es ser clases base
- No pueden ser instanciadas
- Se pueden declarar variables de un tipo de clase abstracto pero solo se le pueden asignar instanciaciones de una subclase de la clase abstracta

# Ejercicio 6

- Cree clases para extender la clase abstracta FormaGeometrica para representar a un circulo y a un cuadrado.

# 6.3 Definición de una Interfaz

- Una interface es un conjunto de declaraciones de métodos (sin implementación, es decir, sin definir el código de dichos métodos). La declaración consta del tipo del valor de retorno y del nombre del método, seguido por el tipo de los argumentos entre paréntesis.
- Cuando una clase implementa una determinada interface, se compromete a dar una definición a todos los métodos de la interface.
- En cierta forma una interface se parece a una clase abstract cuyos métodos son todos abstract.
- La ventaja de las interfaces es que no están sometidas a las más rígidas normas de las clases; por ejemplo, una clase no puede heredar de dos clases abstract, pero sí puede implementar varias interfaces.
- Una de las ventajas de las interfaces de Java es el establecer pautas o modos de funcionamiento similares para clases que pueden estar o no relacionadas mediante herencia.
- En efecto, todas las clases que implementan una determinada interface soportan los métodos declarados en la interface y en este sentido se comportan de modo similar. Las interfaces pueden también relacionarse mediante mecanismos de herencia, con más flexibilidad que las clases.

# 6.3.1 Definición de una Interfaz

- Una interface se define de un modo muy similar a las clases, ejemplo:

```
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.

# Interfaces

- Toman la abstracción un paso más alla
- Solo definen un conjunto de métodos que representan la interfaz de la clase pero no proporcionan una implementación
- También se pueden definir constantes

```
public interface FormaGeometricaInt {
    final double PI = 3.14;
    double obtenArea();
    double obtenPerimetro();
    public String toString();

}
```

- Todos los métodos de una interfaz son automaticamente public y abstract, por tanto no se requiere explicitamente especificarlo

# 6.4 Implementación de la definición de una interfaz.

- Una interface es un conjunto de declaraciones de funciones. Si una clase implementa (implements) una interface, debe definir todas las funciones especificadas por la interface.
- Una clase puede implementar más de una interface, representando una forma alternativa de la herencia múltiple.
- A su vez, una interface puede derivar de otra o incluso de varias interfaces, en cuyo caso incorpora todos los métodos de las interfaces de las que deriva.
- Las interfaces pueden definir también variables finales (constantes).
- 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
        {  ...
         }
```

# Interfaces

- Java proporciona soporte para herencia pseudo-múltiple a través del uso de interfaces
  - Las clases pueden extender solo una clase
  - Sin embargo, las clases pueden **implementar** cualquier número de interfaces

```
public class Circulo implements Forma {
    …
}
```

- Una clase que implementa una interfaz debe proporcionar una implementación para cada método definido en la interfaz
- Si una clase implementa mas de una interfaz con métodos idénticos, una implementación del método es suficiente siempre y cuando el tipo de retorno sea igual en ambas interfaces

# 6.5 Diferencia entre clase abstracta e interface

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.

# 6.5.1 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 asi mismo 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 hayanimplementado dicha interface dejarán de funcionar, a menos que implementen el nuevo método.

# 6.5.2 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.

# Ejercicio 7

- Cree clases para implementar la interfaz FormaGeometricaInt para representar a un circulo y a un cuadrado.

# 6.6 Definición y creación de paquetes (librería)

- Un package es una agrupación de clases.
- El usuario puede crear sus propios packages.
- Para que una clase pase a formar parte de un package llamado pkgName, hay que introducir en ella la sentencia:

  **package pkgName;**

  que debe ser la primera sentencia del fichero sin contar comentarios y líneas en blanco.
- Los nombres de los packages se suelen escribir con minúsculas, para distinguirlos de las clases, que empiezan por mayúscula. El nombre de un package puede constar de varios nombres unidos por puntos (los propios packages de Java siguen esta norma, como por ejemplo **java.awt.event**).
- Todas las clases que forman parte de un package deben estar en el mismo directorio. Los nombres compuestos de los packages están relacionados con la jerarquía de directorios en que se guardan las clases.
- Es recomendable que los nombres de las clases de Java sean únicos en Internet. Es el nombre del package lo que permite obtener esta característica. Una forma de conseguirlo es incluir el nombre del dominio (quitando quizás el país), como por ejemplo en el package siguiente:

  **mapaches.itz.edu.ordenar**

# 6.6 Definición y creación de paquetes (continuación)

- Las clases de un package se almacenan en un directorio con el mismo nombre largo (path) que el package. Por ejemplo, la clase:

**mapaches.itz.edu.ordenar.QuickSort.class**

debería estar en el directorio:

**CLASSPATH\mapaches\itz\edu\ordenar\QuickSort.class**

donde CLASSPATH es una variable de entorno del PC que establece la posición absoluta de los directorios en los que hay clases de Java (clases del sistema o de usuario), en este caso la posición del directorio es en los discos locales del ordenador.

# 6.6.1 Utilización de paquetes

Los packages se utilizan con las finalidades siguientes:

1. Para agrupar clases relacionadas.

2. Para evitar conflictos de nombres (se recuerda que el dominio de nombres de Java es la Internet). En caso de conflicto de nombres entre clases importadas, el compilador obliga a cualificar en el código los nombres de dichas clases con el nombre del package.

3. Para ayudar en el control de la accesibilidad de clases y miembros.

# 6.7 Funcionamiento de los paquetes

- Con la sentencia **import packname**; se puede evitar tener que utilizar nombres muy largos, al mismo tiempo que se evitan los conflictos entre nombres. Si a pesar de todo hay conflicto entre nombres de clases, Java da un error y obliga a utilizar los nombres de las clases cualificados con el nombre del package.

- El importar un package no hace que se carguen todas las clases del package: sólo se cargarán las clases public que se vayan a utilizar. Al importar un package no se importan los sub-packages. Éstos deben ser importados explícitamente, pues en realidad son packages distintos.

  Por ejemplo: Al importar java.awt no se importa java.awt.event.

- Es posible guardar en jerarquías de directorios diferentes los ficheros *.class y *.java, con objeto por ejemplo de no mostrar la situación del código fuente. Los packages hacen referencia a los ficheros compilados *.class.

# 6.7.1 Referencia a clases dentro de un paquete

- En un programa de Java, una clase puede ser referida con su nombre completo (el nombre del package más el de la clase, separados por un punto).
- También se pueden referir con el nombre completo las variables y los métodos de las clases. Esto se puede hacer siempre de modo opcional, pero es incómodo y hace más difícil el reutilizar el código y portarlo a otras máquinas.
- La sentencia import permite abreviar los nombres de las clases, variables y métodos, evitando el tener que escribir continuamente el nombre del package importado. Se importan por defecto el package java.lang y el package actual o por defecto (las clases del directorio actual).
- Existen dos formas de utilizar import: para una clase y para todo un package:

```
import mapaches.itz.edu.ordenar.QuickSort.class;
import mapaches.itz.edu.ordenar.*;
```

que deberían estar en el directorio:

```
classpath\mapaches\itz\edu\ordenar
```

# 6.7.2 Accesibilidad de los paquetes

- 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 archivos 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.

| Visibilidad | public | protected | private | default |
|---|---|---|---|---|
| Desde la propia clase | Sí | Sí | Sí | Sí |
| Desde otra clase en el propio package | Sí | Sí | No | Sí |
| Desde otra clase fuera del package | Sí | No | No | No |
| Desde una sub-clase en el propio package | Sí | Sí | No | Sí |
| Desde una sub-clase fuera del propio package | Sí | Sí | No | No |