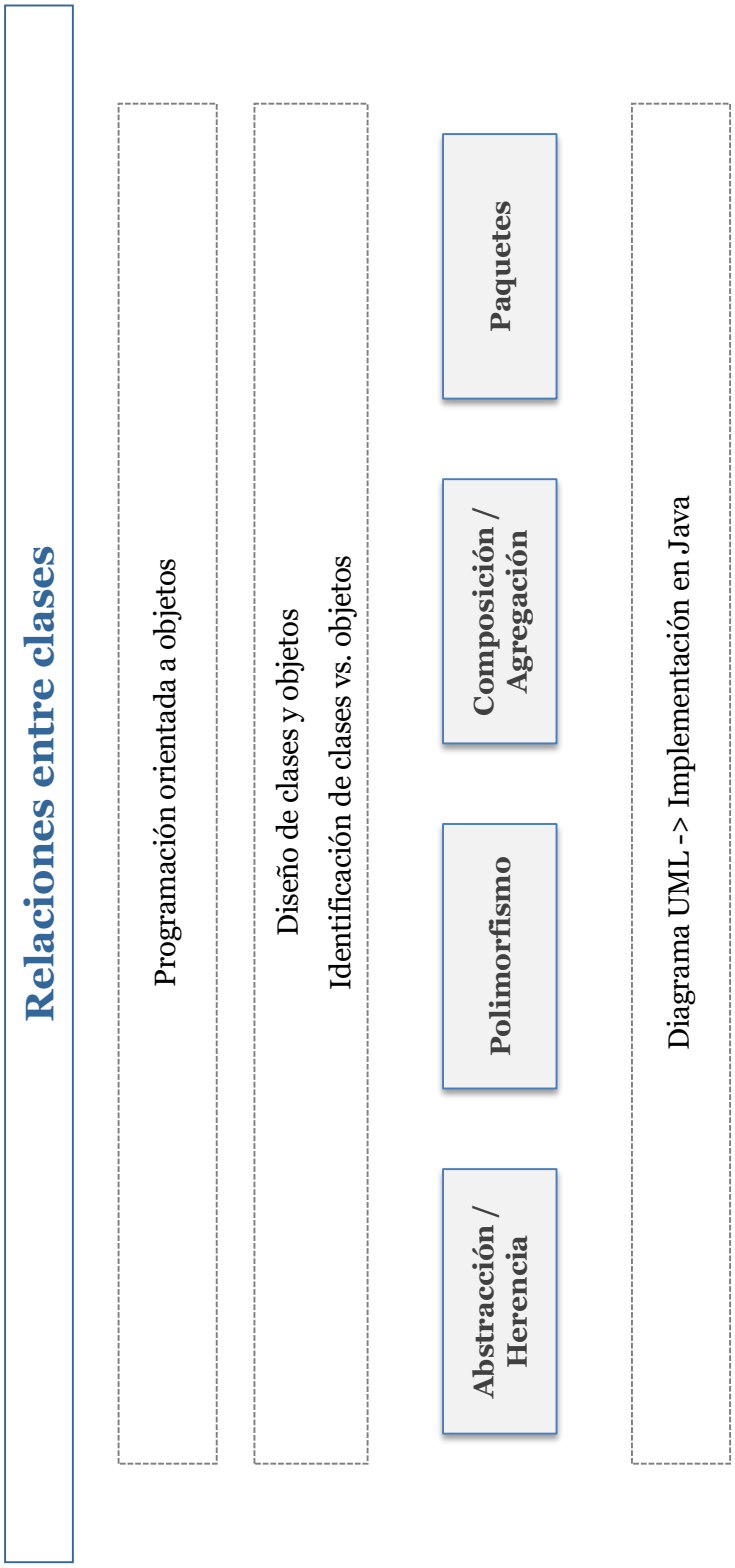
Relaciones entre clases

TEMA 2

Esquema



Ideas clave

2.1. ¿Cómo estudiar este tema?

Para estudiar este tema debes leer las **páginas 75-114 y 123-124** del siguiente libro, disponible en la Biblioteca Virtual de UNIR:

García, L. F. (2010). *Todo lo básico que debería saber sobre programación orientada a objetos en Java*. Barranquilla: Uninorte.

Para comprobar si has comprendido los conceptos realiza el test de autoevaluación del tema.

En este tema vamos a estudiar las principales relaciones entre clases. Entraremos en profundidad en todos los conceptos de diseño y modelado de diagramas de clases. A partir de estas relaciones podemos desarrollar soluciones a problemas complejos de programación.

- Identificar las relaciones entre clases en un problema.
- Identificar clases para diseñar soluciones con características tales como la reutilización y el encapsulamiento.

2.2. Abstracción y herencia

Abstracción

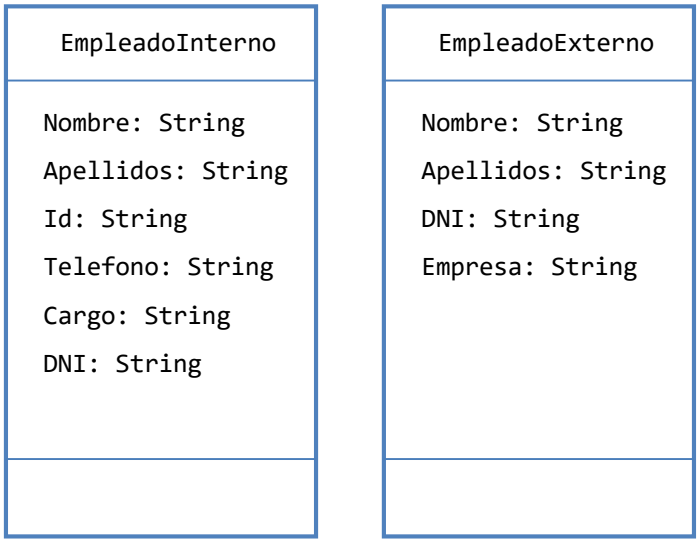
Según la RAE, es el proceso de **separar** por medio de una **operación intelectual** las **cualidades de un objeto** para considerarlas aisladamente o para considerar el mismo objeto en su pura esencia o noción.

En programación orientada a objetos hablamos de abstracción cuando en **el proceso de identificación de clases** seleccionamos aquellos **atributos y métodos comunes** para crear una clase que nos permita **generalizar los valores comunes**.

Ejemplo:

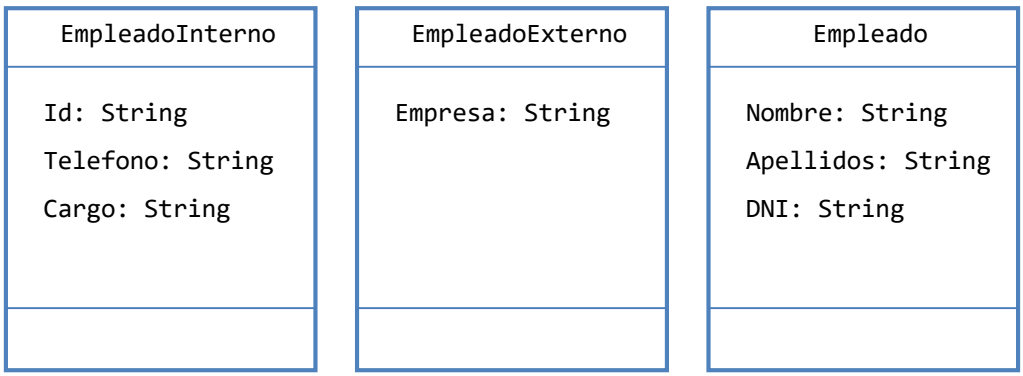
Se quiere realizar un programa informático que gestione los empleados de una empresa informática. Existen dos tipos de empleados los empleados: internos y los colaboradores. Los empleados internos tienen nombre, apellidos, DNI, teléfono, cargo e ID de empleado. De los empleados externos se almacena el nombre, apellidos, DNI y empresa a la que pertenece.

La identificación de clases nos quedaría así:



Como podemos ver, hay **atributos** que son **comunes**. Si realizáramos la identificación de los **métodos** veríamos que ocurre lo mismo. De estas dos clases podríamos extraer una en la que se agruparan aquellos atributos y métodos comunes.

Con lo que nos quedaría algo similar a una clase en la que se agrupan todos los elementos comunes y luego dos clases que tienen la parte especializada.

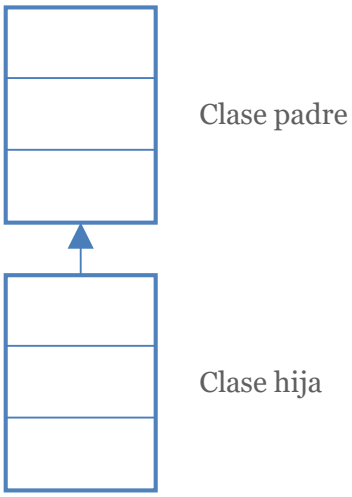


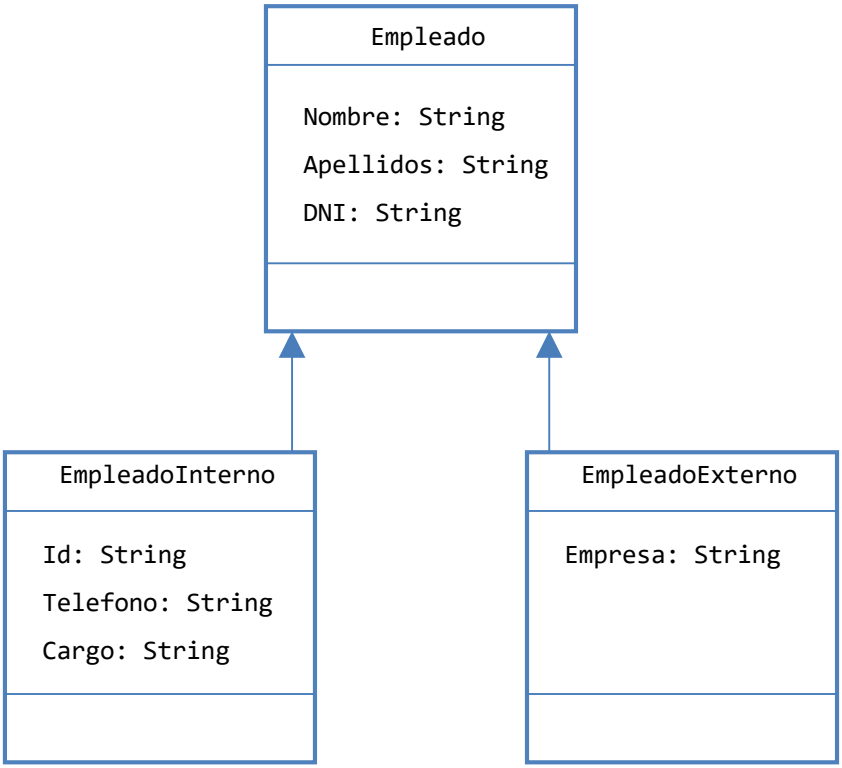
Herencia

Es el proceso por el que una subclase obtiene **todos los métodos y atributos de una clase superior**. Como se puede ver, la herencia no puede existir sin la abstracción.

Según el ejemplo anterior podemos decir que las clases `EmpleadoInterno` y `EmpleadoExterno` pueden heredar comportamiento de la clase `Empleado`.

En UML la herencia se identifica a través de una flecha que apunta a la clase padre.





Implementación en Java

En Java la herencia se implementa a través de la palabra reservada `extends`.

```
class padre {
}

class hijo extends padre{
}
```

Para acceder desde la clase hijo a la clase padre utilizamos la palabra reservada `super`. Si utlizamos `super()` o `super(argumentos)` lo que hacemos es invocar el método constructor de la clase padre.

La subclase:

- Hereda atributos y métodos.
- Siempre y cuando tengan los modificadores de accesibilidad adecuados (acceso paquete, `public` o `protected`).
- La subclase no tiene ningún miembro con el mismo nombre.
- Añade atributos y métodos.
- Modifica atributos y métodos.

Clase abstracta

Es aquella que **no tiene implementado alguno de los métodos** que la dotan de comportamiento. No es instanciable. Es decir, no se puede hacer `new` de la clase.

Cuando tenemos una clase abstracta pura, es decir, que no tiene implementados ninguno de sus métodos, hablamos de una `interface`. Para que una clase use una `interface` se utiliza la palabra reservada `implements`.

```
abstract class Figura{
   abstract int CalcularArea(){}

}
```

```
Interface  Instrumento{
   Void tocar(){}

}
Class viento implements instrumento{

}
```

2.3. Conceptos avanzados de herencia

Herencia múltiple

En POO la herencia múltiple se refiere al proceso por el que **una clase hereda de dos clases superiores**.

En Java la herencia múltiple solo es posible si se realiza a partir de interfaces. Es posible combinar varias interfaces en una clase para simular la herencia múltiple.

2.4. Polimorfismo

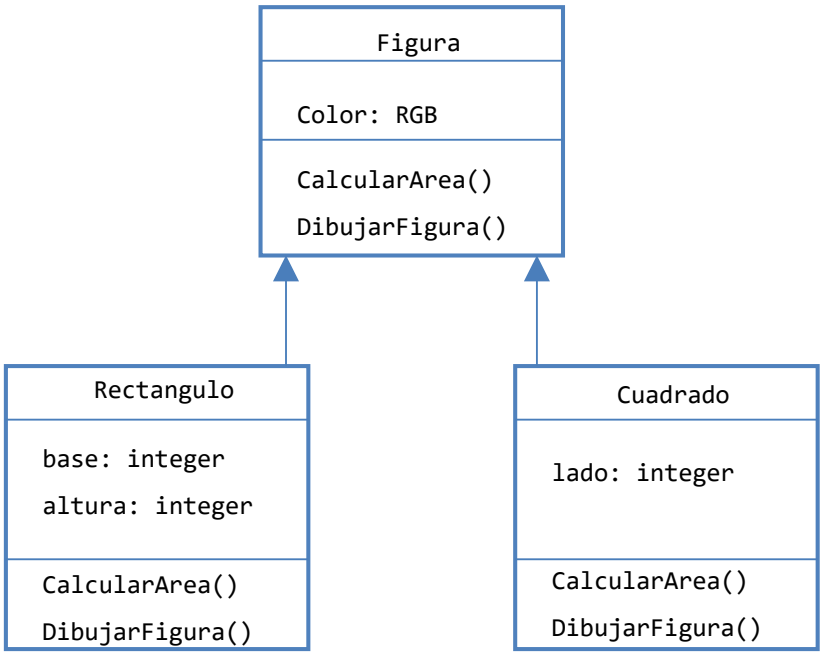
Polimorfismo

Según la RAE es la cualidad de lo que tiene o puede tener distintas formas.

El polimorfismo dentro de la programación orientada a objetos es la capacidad que tiene un objeto de **ejecutar un método que cambia de comportamiento** dependiendo de cómo se haya realizado la instanciación del mismo.

La base para poder realizar polimorfismo es la **sobreescritura de métodos**. Se realiza a través de la anotación `@override`.

Una clase hereda el comportamiento de la clase base, pero reescribe los métodos necesarios para dotarla de otro comportamiento.



En este caso, las figuras Rectangulo y Cuadrado heredan de la clase base `Figura` el atributo color. Los métodos `CalcularArea()` y `DibujarFigura()` los sobreescribiran, ya que son diferentes dependiendo de la figura.

Para saber si una clase es hija de otra utilizamos el operador `instanceof`.

Redefinición y sobrecarga

La redefinición es reescribir un método que se ha heredado con los mismo parámetro y Sobrecargar un método es implementar varios métodos con el mismo nombre pero diferentes parámetros. No se pueden redefinir métodos final ni métodos de clase (`static`).

La **ligadura** (conexión de un método con su llamada) se puede hacer en tiempo de ejecución.

- **Ligadura temprana** (métodos normales, sobrecargados, final).
  - En tiempo de compilación y enlace, el compilador necesita saber la referencia sobre la que se aplica el método.

- **Ligadura tardía** (polimorfismo).
  - La referencia al objeto que llama al método se hace en tiempo de ejecución.
  - No se sabe inicialmente el tipo de objeto que utiliza la llamada.

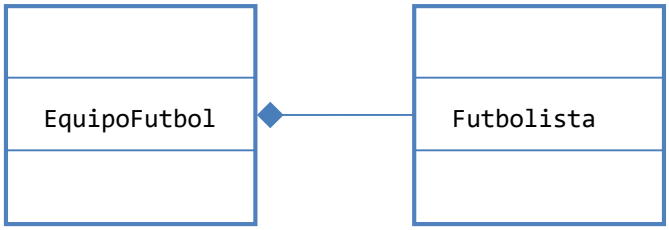
2.5. Composición y agregación

Agregación

Es un tipo de asociación que indica que una clase es parte de otra más grande, también se llama como **composición débil**. La **destrucción del objeto compuesto no supone la destrucción de los objetos que la componen**.

La agregación la utilizamos cuando queremos mostrar una dependencia entre objetos relacionados.

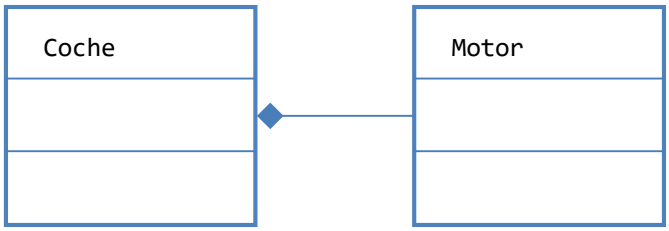
En UML se representa a través de un rombo en el contenedor.



En este ejemplo, la destrucción del objeto que represente `EquipoFutbol` no implicaría la destrucción de los `Futbolistas`. De esta manera los objetos `Futbolista` seguirían viviendo a pesar de que el contenedor haya desaparecido.

Composición

Este tipo asociación indica que una clase contenedora contiene **elementos más pequeños**. Los componentes forman parte del **objeto compuesto**. La **destrucción** del objeto contenedor supone la **eliminación de los objetos que la componen**. También se llama **agregación fuerte**.



En este caso lo que estamos indicando es que el objeto `motor` no puede vivir si desaparece el objeto `coche`. Ambos acabarían su ciclo de vida a la vez.

Para implementar la relación de composición en Java, se declaran los miembros de la clase compuesta como instancias de clase componedora.

Por ejemplo, una clase `MiLinea`, quedaría definida por dos objetos de la clase `MiPunto`, que representasen el punto de inicio y de fin de la línea:

```
class MiLinea {

    MiPunto punto1;
    MiPunto punto2;
    Libro() {
        punto1=new MiPunto();
        punto2=new MiPunto();
    }
}
```

2.6. This y super

La palabra reservada `this` nos permite acceder a los miembros de la propia clase. Mediante `this.NombreVariable` estamos indicando que la variable `NombreVariable` es la de la propia clase.

```
public class Potencia {
    double base;
    int exponente;
    double valor;

     public Pontencia(double base, int exponente) {

              this.base = base;
              this.exponente = exponente;

              if (exponente == 0) return;

          for (;exponent>0;exponent--)
            valor*=base;
    }
```

En el constructor de la clase `Potencia` los argumentos tienen el mismo nombre que las variables de la clase. En la sentencia `this.base = base`, `this.base` se refiere a la variable base de la clase `Potencia` y *base* se refiere al argumento del constructor.

La palabra reservada `super` nos permite hacer referencia al método o variable de la clase padre.


```
public class ClasePadre {
    int i;
      public ClasePadre() {
          i = 10;
          }

      public setValor( int valor ) {
          i = valor;
          }
      }

class ClaseHija extends ClasePadre {
    int j;

      public ClaseHija() {
          j = 12;
          }
      public setValor( int valor ) {
          j = valor/2;
               super.setValor( valor );

          }
      }
```

El constructor `ClaseHija` invoca por omisión al constructor sin argumentos de `ClasePadre`.

El método `setValor` de `ClaseHija` invoca al método `setValor` de `ClasePadre` utilizando la referencia `super`.

En el siguiente código, el constructor de la clase hija establecerá el valor de i a 10 y de j a 15, después cambiará el valor de j a 10 y el de j a 20:

```
ClaseHija ch;
 ch = new ClaseHija();
 ch.setValor( 20 );
```

Si realizamos la siguiente modificación en nuestras clases de ejemplo:

```
public class ClasePadre {
        int i;
          public ClasePadre() {
              i = 10;
              }

               public ClasePadre(int valor) {
              i = varlor;
              }

          public setValor( int valor ) {
              i = valor;
              }
          }


    class ClaseHija extends ClasePadre {
        int j;

          public ClaseHija() {
              j = 12;
              }

               public ClaseHija(int valor) {
              super(valor); //llamar al constructor de la superclase
                   j = valor;
              }

          public setValor( int valor ) {
              j = valor/2;
                   super.setValor( valor );

              }
          }
```

En el siguiente código, el constructor de la clase hija establecerá el valor de i y de j a 30, después cambiará el valor de j a 10 y el de j a 20:

```
ClaseHija ch;
 ch = new ClaseHija(30);
 ch.setValor( 20 );
```

2.7. Complejidad de un algoritmo

Anteriormente vimos el concepto de **eficiencia** y **la complejidad de un algoritmo**. En esta apartado se analizará la complejidad de un algoritmo. Para ello, debemos considerar que cada algoritmo consume tiempos diferentes y que existirá una constante $K > 0$ que estará relacionada entre ambos.

Es decir que se pude definir que un tiempo más eficiente (T_e) es menor a otro tiempo de ejecución menos eficiente (T_m) y se puede expresar de la siguiente forma:

$$T_e < T_m$$

Por ejemplo, consideremos dos tiempos de ejecución que tienen los siguientes tiempos en función de una entra $n=100.000$.

$$T1\ (n) = 10^6 n^2 \text{ y } T2\ (n) = 5n^3$$

Comparando estos dos tiempos de ejecución, el tiempo $5n^3$ es mejor que $10^6 n^2$. Sin embargo, para tamaños de entrada mayores a 200.000, el tiempo $10^6 n^2$ será mejor que el de $5n^3$. En la siguiente tabla se puede ver la relación de entrada y salida para estos dos tiempos.

Entrada *n*	**$10^6 n^2$**	**$5n^3$**	**Eficiencia de tiempo**
100.000	$10x10^{15}$	$5x10^{15}$	$5n^3$
200.000	$40x10^{15}$	$40x10^{15}$	Igual
300.000	$60x10^{15}$	$135x10^{15}$	$10^6 n^2$

Relación de entrada y salida entre dos tiempos de ejecución.

El tiempo de ejecución de un algoritmo dependerá de varios factores como el número de instrucciones que realice a través del código, es decir que a veces tendrá una carga máxima de instrucciones y a veces una carga mínima. Dicho en otros términos, si el algoritmo pasa por un ciclo `while()` que tiene realizar un millón de operaciones (caso peor), no será lo mismo que si la condición del `while()` no se cumpla y no tenga que hacer todas las operaciones del ciclo.

Por lo tanto, se pueden definir tres casos diferentes como son: **caso mejor, caso medio y caso peor**. En donde el caso mejor será el caso con menos instrucciones y el caso medio será la media entre el máximo y mínimo de instrucciones.

Para poder analizar el tiempo de un algoritmo se tomará en cuenta que el valor de entrada $n = 1$ y todas aquellas operaciones básicas (OB) de cada línea de código valdrán 1.

Por ejemplo, consideremos el siguiente ejemplo:

No. de Línea	Código	Tipo de operación	OB
Línea 1	`i= 1;`	Una asignación	1
Línea 2	`while(numeros[i]<numeros && i<n){`	Dos comparaciones, una AND y un acceso a un vector	4
Línea 3	`i++;`	Un incremento	1
Línea 4	`if(i==numeros[i]){`	Una condición y un acceso a un vector	2
Línea 5	`return i;}`	Un retorno	1
Línea 6	`else {`		
Línea 7	`return numeros;}}`	Un retorno	1

Como se puede observar, las líneas 1, 5 y 7 tienen solo una operación y por lo tanto OB=1. Mientras que las demás realizan más operaciones.

Calculemos ahora el caso mejor, pero y medio:

a) Para el caso mejor, se indicó anteriormente que será el caso en donde se realicen menos operaciones. Es decir que empezara por la línea 1 y seguirá por la línea 2. En el mejor de los casos (en términos de menos operaciones) la línea dos, se realizará solo la mitad de la condición y seguirá por la línea 4 y terminará por las líneas 5 o 7 según sea el caso. Por lo tanto, el $T_{mejor} = 1+2+2+1 = 6$.

b) Para el caso peor, el algoritmo empezara por la línea 1 y llegara al bucle while que se repetirá *n-1* veces hasta que se cumpla la segunda condición. Dentro del bucle while hay un incremento con 1 OB y el bucle vale 4 OB es decir que se repetirá *(n-1)(4+1),* después que la condición se efectúa del while valdrá otros 4 OB. Después la línea 4 usa 2 OB y finalmente la función termina con la línea 7 que vale 1 OB. Por consiguiente, el tiempo en el peor caso es:

$$T_{peor} = 1+((n-1)(4+1)+4) + 2 + 1 = 5n +3.$$

c) Para el caso medio, será la mitad de veces que se repita el bucle while: $(n-1)/2=m$. Por lo tanto, el valor medio es:

$$T_{medio} = 1+(m(4+1)+4) + 2 + 1 = (5n +11)/2.$$

Para este ejemplo el grado de complejidad de este algoritmo es de orden 1. Si el algoritmo tuviese dos bucles entonces los casos serían de orden 2 de complejidad.

De esta manera se puede saber los grados de complejidad que un algoritmo tiene y analizar hasta qué punto el algoritmo será más o menos eficiente como se vio en la comparación de un algoritmo de grado cubico y cuadrado.

Lo + recomendado

No dejes de leer…

Herencia

García, L. F. (2010). *Todo lo básico que debería saber sobre programación orientada a objetos en Java*. Barranquilla: Uninorte.

Las páginas 59-74 de este libro recogen varios ejemplos de implementación de la herencia en Java.

El libro está disponible en la Biblioteca Virtual de UNIR.

Paquetes en Java

Consulta la página de Oracle en la que podrás encontrar información detallada sobre los paquetes en Java.

Accede al artículo desde el aula virtual o a través de la siguiente dirección web:
http://docs.oracle.com/javase/tutorial/java/package/

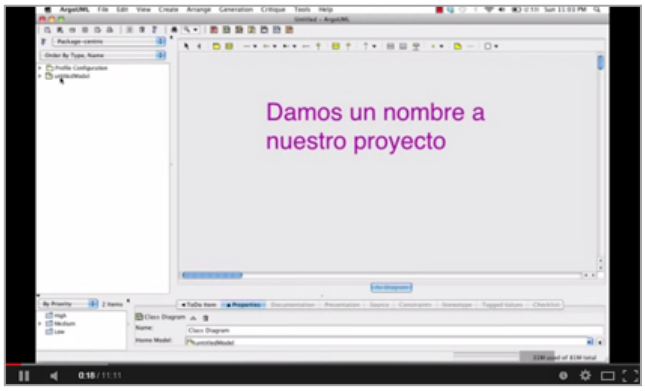
Herencia en Java

Consulta la página de Oracle en la que podrás encontrar información detallada sobre herencia en Java.

Accede al artículo desde el aula virtual o a través de la siguiente dirección web:
http://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html

No dejes de ver...

Proyecto en ArgoUML

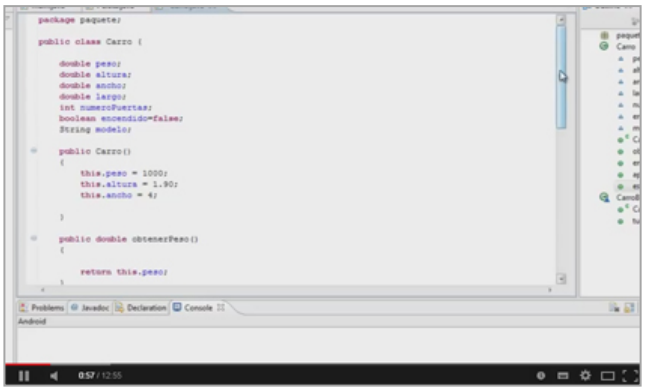


Vídeo que explica cómo crear un proyecto en ArgoUML.

Accede al vídeo desde el aula virtual o a través de la siguiente dirección web:
https://www.youtube.com/watch?v=BGM1QbPb9dQ

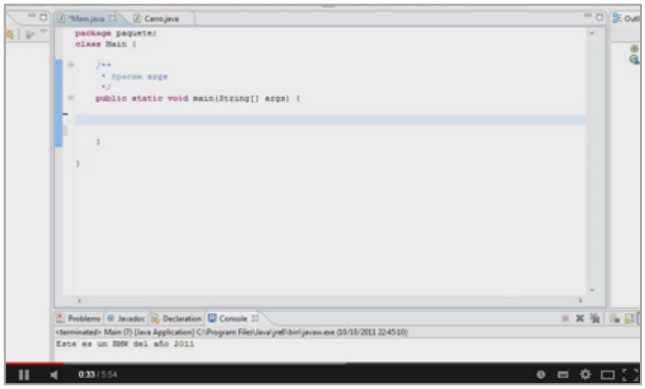
Herencia en Java

Vídeo que explica mediante un ejemplo la herencia en Java.



Accede al vídeo desde el aula virtual o a través de la siguiente dirección web:
https://www.youtube.com/watch?v=CYNLhg42O9c

Polimorfismo en Java



Vídeo que explica mediante un ejemplo el polimorfismo en Java.

Accede al vídeo desde el aula virtual o a través de la siguiente dirección web:
https://www.youtube.com/watch?v=BSw1MLEc4PQ

+ Información

A fondo

Ejemplo de herencia en Java

Ejemplo en el que se explica la herencia y el uso de `super`.

Accede al ejemplo desde el aula virtual o a través de la siguiente dirección web: http://www.aprenderaprogramar.com/index.php?option=com_attachments&task=download&id=594

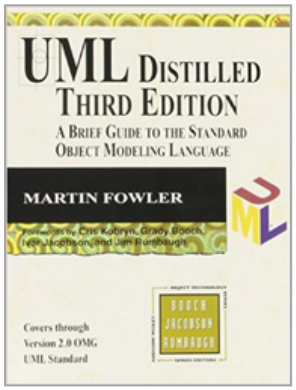
Ejemplo de polimorfismo en Java

Ejemplo en el que se explica el polimorfismo en Java.

Accede al ejemplo desde el aula virtual o a través de la siguiente dirección web: http://www.aprenderaprogramar.com/index.php?option=com_attachments&task=download&id=599

UML Distilled

Fowler, M. (2004). *UML Distilled: A Brief Guide to de Standard Object Modeling Language*. Massachusetts: Addison Wesley.



Libro que nos explica los fundamentos de UML de manera sencilla y útil. El libro describe los principales diagramas y cómo se puede hacer uso de ellos para desarrollar aplicaciones.

Accede a una parte del libro desde el aula virtual o a través de la siguiente dirección web: http://books.google.es/books?id=nHZslSr1gJAC&printsec=frontcover

Programación orientada a objetos usando Java

Flórez, H. A. (2012). *Programación orientada a objetos usando Java*. Colombia: Ecoe Ediciones.

Manual en el que se hace una introducción a la programación orientada objetos con Java.

El libro está disponible en la Biblioteca Virtual de UNIR.

Test

1. En Java la herencia se implementa:

A. Indicando en la clase padre los hijos que derivan de ella.

B. Indicando en el paquete las clases padre.

C. Indicando en la clase hija el padre de la que deriva.

D. Indicando en la definición del proyecto la jerarquía.

2. En Java la herencia múltiple:

A. No existe pero se puede simular.

B. No existe y no se puede implementar.

C. Existe y se implementa igual que la herencia simple.

D. Existe y se implementa a través de una sentencia especial.

3. La agregación es:

A. La propiedad por la que una clase hereda comportamiento de otra clase llamada clase padre.

B. Es la relación que tienen dos clases que se crean de manera consecutiva.

C. Es la relación que tienen dos clases en las que una clase forma parte de la otra.

D. Es la relación que tienen dos clases que se intercambian mensajes.

4. Para acceder a una clase desde su clase hija utilizamos:

A. El operador `super`.

B. El operador `this`.

C. No se puede.

D. Instanciando la clase padre.

5. Para saber si un objeto se ha creado a partir de otra se utiliza:

A. El operador `super`.

B. El operador `this`.

C. Se comparan con el ==

D. Se utiliza el operador `instanceof`

6. Las clases abstractas en Java:

A. Se utilizan para implementar comportamientos especiales.

B. Se utilizan para crear objetos especiales que pueden ser serializados.

C. No pueden ser instanciadas.

D. Se pueden instanciar con un operador especial

7. En UML la herencia se representa:

A. Con un triángulo apuntando a la clase hija.

B. Con un triángulo apuntando a la clase padre.

C. Con un rombo en la clase padre.

D. Con un rombo en la clase hija.

8. En UML la composición se representa:

A. Con un triángulo apuntando a la clase contenedora.

B. Con un triángulo apuntando a la clase contenida.

C. Con un rombo apuntando a la clase contendora.

D. Con un rombo apuntando a la clase contenida.

9. Los paquetes en Java se utilizan:

A. Para agrupar clases por comportamiento o función.

B. Para definir el comportamiento de las clases.

C. Para instanciar los objetos de manera más controlada.

D. Para aislar los programas del resto de ejecuciones.

10. Una clase que hereda de otra:

A. Ve todos los métodos y atributos independientemente de su visibilidad.

B. Ve todos los métodos y atributo menos los privados y `protected`.

C. Ve todos los métodos y atributos menos los privados.

D. Solo ve los métodos y atributos públicos.