Sistemas y Aplicaciones Informáticas

Tema 29. Programación Orientada a Objetos. Objetos. Clases. Herencia. Poliformismos.

1. ÁMBITO DE DOCENCIA. 3

2. PROGRAMACIÓN ORIENTADA A OBJETOS. 3

2.1. CONCEPTO. CARACTERÍSTICAS. 3
2.2. COMPARACIÓN CON PROGRAMACIÓN MODULAR. 3
2.3. CONCEPTO DE OBJETO. CARACTERÍSTICAS. 4
2.4. CONCEPTO DE CLASE, ATRIBUTO Y MÉTODO. RELACIÓN CON LOS OBJETOS. 4
2.5. COMUNICACIÓN ENTRE OBJETOS. 4
2.6. ENCAPSULAMIENTO Y OCULTACIÓN. VENTAJAS. 4
2.7. CONSTRUCTORES Y DESTRUCTORES. 5

3. HERENCIA. 5

3.1. CONCEPTO. JERARQUÍA DE CLASES. TIPOS. 5
3.2. CARACTERÍSTICAS Y TIPOS DE HERENCIA. 6
3.3. CLASES ABSTRACTAS. 6

4. POLIMORFISMOS. 6

4.1. CONCEPTO. CLASIFICACIÓN. 6
4.2. LIGADURA ESTÁTICA Y DINÁMICA. 7

5. LENGUAJES ORIENTADOS A OBJETOS. 7

5.1. PROGRAMAS ORIENTADOS A OBJETOS. CARACTERÍSTICAS Y MODELO DE EJECUCIÓN. 7
5.2. LENGUAJE HÍBRIDO: C++ 8
5.2.1. Características principales. 8
5.2.2. Clases, objetos, métodos y mensajes. 8
5.2.3. Herencia y polimorfismo. 9
5.3. LENGUAJE PURO: JAVA. 10
5.3.1. Características principales. 10
5.3.2. Clases, objetos, métodos y mensajes. 10
5.3.3. Herencia y polimorfismo. 10

1. Ámbito de docencia.

- Sistemas informáticos monousuario y multiusuario (ASI 1).
- Sistemas informáticos multiusuario y en red (DAI 1).
- Sistemas operativos en entornos monousuario y multiusuario (ESI 1).

2. Programación orientada a objetos.

2.1. Concepto. Características.

- La programación orientada a objetos es un paradigma de programación que consiste en modelar la realidad como un conjunto de objetos que interactúan entre sí para resolver un problema.
- Las características básicas comunes que presentan los lenguajes orientados a objetos son:
  - *Encapsulamiento*. Las propiedades y el comportamiento de los objetos están definidos en las clases, formadas por estructuras de datos y algoritmos denominadas atributos y métodos. Un objeto es una instancia de una determinada clase y tiene su propio conjunto de datos. Los métodos se aplican a un objeto concreto y son propias de la clase a la que pertenece.
  - *Ocultación.* Para aumentar la modularidad y como medida de protección, un objeto se considera como una caja negra en la que se puede introducir o de la que se puede extraer información sin necesidad de conocer su funcionamiento interno.
  - *Envío de mensajes*. Consiste en nombrar y activar con los parámetros adecuados uno de los métodos del objeto al que se envía el mensaje. Es la manera que tienen los objetos de relacionarse, y es equivalente a las llamadas a procedimientos de los lenguajes imperativos.
  - *Herencia*. Es un mecanismo mediante el cual pueden crearse clases a partir de otras, transmitiéndose todos los atributos y todos los métodos de clases padres a clases hijas, con la ventaja de que en estas últimas pueden añadirse más atributos y métodos.
  - *Polimorfismo*. Es la capacidad que tiene un objeto de una determinada clase de hacerse pasar por un objeto de una clase ancestro. Esta característica, junto a la posibilidad de las clases hijas de redefinir los métodos heredados, permiten la reutilización de código.

2.2. Comparación con programación modular.

- Semejanzas entre programación orientada a objetos y programación modular:
  - *Abstracción*. En un programa modular es suficiente saber que un módulo dado realiza una tarea específica, se puede utilizar sin tener que conocer cómo funciona su interior.
  - *Ocultación.* Un programa modular está formado por procedimientos y funciones que ocultan en su interior algoritmos y datos útiles exclusivamente para estos algoritmos.
- Diferencias entre programación orientada a objetos y programación modular:
  - *Separación de datos y código*. Las estructuras de datos utilizadas en programación modular son globales o se pasan como parámetros a los módulos. Esto hace que los cambios producidos en los tipos de dato deben reflejarse en todos los módulos implicados, y que raramente es posible anticipar el diseño de un sistema completo antes de implementarlo.
  - *Descomposición funcional*. La programación modular descompone el problema en módulos que cumplen una determinada función. Esto la convierte en más inestable porque depende de los cambios en los requisitos funcionales.

2.3. Concepto de objeto. Características.

- Un objeto es una representación de una entidad real o abstracta dentro de un programa, que puede ser un objeto físico, un elemento de interfaces gráficos de usuario, una estructura de datos, un tipo de dato definido por el usuario, etc.
- Se caracteriza por poseer:
  - *Unas propiedades*, que describen el estado del objeto dentro del programa.
  - *Un comportamiento*, que determina la respuesta del objeto frente a otros objetos.
  - *Una referencia*, que identifica a un único objeto dentro del programa y es independiente de su estado. Puede haber varias referencias a un mismo objeto, y dos objetos con diferentes referencias pueden tener el mismo estado.

2.4. Concepto de clase, atributo y método. Relación con los objetos.

- Una clase es una estructura que describe a todos los objetos que tienen las mismas propiedades y el mismo comportamiento mediante la definición de sus atributos y sus métodos.
- Los atributos son estructuras de datos simples o compuestas que representan las propiedades de los objetos pertenecientes a una determinada clase.
- Los métodos son un conjunto de funciones y procedimientos que definen el comportamiento de los objetos pertenecientes a una determinada clase. Están compuestos por:
  - *Una cabecera*, formada por un identificador y opcionalmente los parámetros de entrada, que pueden ser pasados por valor y por referencia. En programación orientada a objetos se utilizan los parámetros por referencia para pasar objetos a los métodos.
  - *Un cuerpo*, formado por una secuencia de instrucciones, que pueden ser asignaciones, estructuras alternativas, estructuras iterativas, llamadas a métodos y creaciones de objetos.
- Cada objeto es una instancia de una determinada clase, y los atributos de un conjunto de objetos de la misma clase pueden tener valores diferentes. Los métodos siempre se aplican sobre objetos concretos, pero son propios de la clase a la que pertenecen los objetos.

2.5. Comunicación entre objetos.

- Los objetos de un programa se comunican entre sí mediante mensajes. La modificación y la consulta del estado de un objeto por parte de otro se realiza de esta manera.
- Cuando un objeto emisor envía un mensaje a un objeto receptor, lo que ocurre realmente es que el primero realiza una petición al segundo para que éste ejecute uno de sus métodos utilizando los parámetros enviados por el emisor.
- Un mensaje consta de la identidad del objeto receptor, el método del receptor cuya ejecución se ha solicitado, y los parámetros que el receptor necesite para ejecutar el método requerido.
- El método a aplicar en respuesta a un mensaje se determina por la clase del receptor. Todos los objetos de una clase usan el mismo método como respuesta a mensajes similares. Los objetos de clases distintas pueden responder al mismo mensaje, aunque aplicando métodos distintos.

2.6. Encapsulamiento y ocultación. Ventajas.

- El encapsulamiento es una propiedad de la programación orientada a objetos por la cual los atributos y los métodos se encierran dentro de las clases. Los atributos y los métodos de una

clase pueden ser públicos, si son accesibles desde cualquier punto del programa, o privados, si sólo son accesibles desde la propia clase. Normalmente una clase ofrece un conjunto de métodos públicos denominado interfaz, a través de la cual se puede actuar sobre los atributos privados.

- La ocultación es otra propiedad de la programación orientada a objetos por la cual un objeto se considera como una caja negra en la que se puede introducir o de la que se puede extraer información a través de su interfaz sin necesidad de conocer su funcionamiento interno.
- El encapsulamiento y la ocultación aportan dos ventajas:
  - *Modularidad y protección*. Se puede escribir y mantener el código de una clase de manera independiente del código de las demás clases. También se garantiza que se hace buen uso de los objetos, manteniendo la coherencia de la información.
  - *Encubrimiento de detalles de implementación*. El objeto puede mantener información privada y proteger métodos cuyos detalles de implementación pueden cambiar.

2.7. Constructores y destructores.

- Un constructor es un método especial que sirve para crear un nuevo objeto y asignar valores iniciales a sus atributos. Se caracteriza porque no devuelve valores y puede admitir parámetros. Si no se define ningún constructor para un determinado objeto, los compiladores normalmente generan un constructor por defecto, que sitúa ceros en cada byte de las variables de un objeto.
- Los constructores siempre se llaman de forma implícita al declarar un objeto. En objetos globales, el constructor se llama implícitamente cuando se arranca el programa. También se llama al constructor cuando se crea un objeto explícitamente reservando un espacio en memoria.
- Un destructor realiza la operación opuesta de un constructor, limpiando el almacenamiento asignado a los objetos cuando se crean. Se caracteriza porque no devuelve valores y tampoco admite parámetros. Si no se define ningún destructor para un determinado objeto, los compiladores normalmente generan un destructor por defecto, que no hace nada.
- Los compiladores llaman automáticamente al destructor correspondiente cuando el objeto sale fuera de su ámbito en el caso de los objetos locales, o cuando finaliza el programa en el caso de los objetos globales. También se llama al destructor cuando se libera la memoria de un objeto creado de manera explícita.

3. Herencia.

3.1. Concepto. Jerarquía de clases. Tipos.

- La mente humana clasifica los conceptos de acuerdo a dos dimensiones: pertenencia (relación has a) y variedad (relación is a). La pertenencia se implementa mediante las estructuras de datos. La variedad se consigue gracias a la herencia, mediante la cual se crean clases a partir de clases ya existentes por medio de la propagación de sus atributos y sus métodos. La clase existente se denomina clase base o superclase, y la nueva clase se conoce como clase derivada o subclase.
- Una clase derivada puede ser clase base en un nuevo proceso de herencia, creando de esta manera una jerarquía de clases, en la que las clases derivadas están formadas por los atributos y los métodos heredados de su clase base, más los atributos y los métodos propios. En esta jerarquía puede existir una clase raíz de la cual heredan las demás directa o indirectamente.

- La jerarquía de clases se produce por generalización (se detectan clases con un comportamiento común) o por especialización (se detecta que una clase es un caso especial de otra).

3.2. Características y tipos de herencia.

- Características de la herencia:
  - *Transitividad*. Si B hereda de A, y C hereda de B, entonces B es un descendiente directo de A y C es un descendiente indirecto de A.
  - *Reutilización de código*. Si B hereda de A entonces B incorpora las propiedades (atributos) y el comportamiento (métodos) de la clase A, pero puede incluir adaptaciones:
    - B puede añadir nuevos atributos y nuevos métodos.
    - B puede redefinir métodos, extendiendo o mejorando la implementación original.
    - B puede renombrar atributos o métodos.
    - B puede implementar un método abstracto en A.
- Tipos de herencia:
  - *Herencia simple*. Consiste en que una clase derivada hereda de una única clase base.
  - *Herencia múltiple*. Consiste en que una clase derivada hereda de varias clases base.
  - *Herencia de tipos*. Da lugar a jerarquías basadas en aspectos comunes, considerando un objeto como del tipo de su clase o de cualquier clase base suya.
  - *Herencia de implementación*. Agrupa clases no relacionadas pero que tienen código similar. Permite que una clase herede una parte o toda su implementación de otra clase.

3.3. Clases abstractas.

- Una clase abstracta es aquella clase base que no tiene ninguna instancia y que es utilizada para crear clases derivadas. Su función es la de agrupar métodos comunes de las clases que se derivan de ella, y simplificar el código y mejorar su comprensión.
- Las clases abstractas contienen métodos sin código que deben ser implementados en sus clases derivadas. Son abstractas también las clases que hereden de las abstractas y no implementen todos los métodos declarados en la clase base. No es posible crear instancias de una clase abstracta, pero si declarar entidades de estas clases.
- Las clases abstractas especifican una funcionalidad común a un conjunto de clases derivadas. Por ejemplo, la clase humano puede servir para definir las clases derivadas hombre y mujer, mientras que las instancias se harán de estas dos clases.

4. Polimorfismos.

4.1. Concepto. Clasificación.

- El polimorfismo es la capacidad que tiene una jerarquía de clases relacionadas por la herencia para llamar a métodos distintos con el mismo nombre, según la clase a la que pertenece el objeto.
- Hay varias formas de polimorfismo:
  - *Polimorfismo ad hoc*. Hace referencia a métodos que funcionan con varios tipos distintos que no tienen que tener una estructura común, y que pueden producir resultados distintos para cada tipo. Se pueden identificar dos tipos:

        - **Polimorfismo de coerción**. Operación semántica que convierte implícitamente un argumento en el tipo esperado en una ocasión que produciría un error.
        - **Polimorfismo de sobrecarga**. Técnica sintáctica según la cual se usa el mismo nombre para denotar funciones distintas en las que se usa el contexto para distinguir entre ellas.
    - *Polimorfismo universal*. Hace referencia a métodos que funcionan con todos los tipos que posean una estructura común dada. Se pueden identificar dos tipos:
        - **Polimorfismo parametrizado**. Hace referencia a funciones que tienen un parámetro de tipo implícito o explícito para cada tipo de aplicación de esa función.
        - **Polimorfismo de inclusión**. Consigue la funcionalidad de los métodos a través de la herencia y los subtipos.

4.2. Ligadura estática y dinámica.

- **Ligadura estática**. Cuando la decisión de qué método aplicar ante un mismo mensaje, se realiza durante la compilación, en función de la clase a la que pertenezca el objeto. Es más eficaz, no se pierde tiempo en la ejecución, pero es menos flexible.
- **Ligadura dinámica**. Cuando la decisión de qué método aplicar ante un mismo mensaje, se realiza en tiempo de ejecución, en función de la clase a la que pertenezca el objeto. Es muy flexible, permite definir y manejar jerarquías de clases de modo muy simple, pero es menos eficaz ya que durante la ejecución el programa debe decidirse que método debe aplicarse.

5. Lenguajes orientados a objetos.

5.1. Programas orientados a objetos. Características y modelo de ejecución.

- Un programa orientado a objetos está formado por una colección de clases y un conjunto de objetos que se comunican entre sí, con las siguientes características:
    - La comunicación entre objetos se realiza por medio de mensajes, que indica la acción a realizar por el objeto. El conjunto de mensajes a los cuales responde un objeto se denomina protocolo del objeto. Los métodos de un objeto implementan su protocolo.
    - Es necesaria una función principal que cree objetos y comience la ejecución llamando a sus métodos. Las distintas clases y la función principal se guardan en ficheros independientes, que son ensamblados durante la compilación dando lugar a una unidad de código ejecutable.
    - Para obtener un código ejecutable se deben ensamblar las clases para formar sistemas (cerrado). Un sistema viene dado por un conjunto de clases, la clase raíz y el procedimiento de creación de la clase raíz.
- El modelo de ejecución de un programa orientado a objetos es el siguiente:
    - La ejecución empieza con la creación de un objeto raíz y la aplicación de un mensaje sobre él. Se debe proporcionar el nombre de la clase que conduzca la aplicación. Al ejecutar un programa, el sistema localizará esta clase y ejecutará el método main que contenga.
    - El flujo de ejecución siempre se encuentra aplicando un método sobre un objeto, creando nuevos objetos o ejecutando una instrucción de asignación. Los mensajes se envían desde unos objetos y se reciben en otros.
    - Los objetos se crean cuando se necesitan, y se borran cuando ya no son necesarios, recuperando la memoria ocupada por ellos.

5.2. Lenguaje híbrido: C++.

5.2.1. Características principales.

- Los lenguajes híbridos son lenguajes de programación orientados a objetos que además permiten realizar programación modular, permitiendo mayor flexibilidad y eficiencia del código.
- C++ es una evolución de C que incluye la programación orientada a objetos. Fue propuesto por Bjarne Stroupstrup en 1983 y está basado en Simula utilizando sintaxis de C. Es un lenguaje compilado de propósito general que incorpora una biblioteca estándar de clases.

5.2.2. Clases, objetos, métodos y mensajes.

- Para C++ una clase es un tipo especial de estructura (*struct*) formada por atributos y métodos denominados miembros. El acceso a cada uno de los miembros de una clase puede ser:
  - *Público (public).* Un miembro público puede utilizarse en cualquier punto del programa.
  - *Privado (private).* Un miembro privado sólo puede utilizarse dentro de la misma clase.
  - *Protegido (protected).* Un miembro protegido puede utilizarse dentro de la misma clase y en todas sus clases derivadas.
- El control de acceso *protected* y *private* es a veces más restrictivo de lo deseado. En ocasiones, una clase puede permitir que otra tenga acceso a sus datos privados. Esto se puede especificar si una clase declara a otra como amiga (*friend*), permitiéndole el acceso a su estructura interna.
- Los objetos son creados implícitamente mediante constructores definidos con el mismo nombre de la clase, los cuales realizan la inicialización de los atributos después de que los objetos sean creados. También pueden crearse explícitamente utilizando *new*. Los objetos pueden ser referenciados mediante variables del tipo objeto, o punteros al tipo objeto.
- Los objetos son destruidos implícitamente mediante destructores definidos con el mismo nombre de la clase precedido del símbolo '~', o explícitamente utilizando *delete*.

```
class Complejo {
     public:
      float real;
      float imag;
      Complejo(real=0, imag=0);
      ~Complejo();
}

Complejo *c1, *c2, *c3;
c1 = new Complejo();
c2 = new Complejo(3.141592);
c3 = new Complejo(3.141592, 2.4);
```

- Para llamar un método, se utiliza el identificador del objeto y '->' o '.' en función de que el identificador del objeto sea o no un puntero, respectivamente.

```
Cuenta Cta; Cuenta *pCta;
Cta.reintegro(1000);
pCta->reintegro(1000);
```

- Los parámetros de los métodos pueden pasarse por valor y por referencia. Si se utilizan los punteros, el programador tiene distinguir entre *p y &p para los parámetros por valor y por referencia respectivamente.

5.2.3. Herencia y polimorfismo.

- En C++ existe la herencia de tipos y la herencia de implementación, y están permitida la herencia simple y múltiple. No existe clase raíz en la jerarquía de clases.
- El proceso de herencia puede efectuarse de dos formas distintas:
  - *Herencia pública (class B: public A {...}).* La clase B hereda los miembros *public* y *protected* de la clase A como miembros *public* y *protected*, respectivamente.
  - *Herencia privada (class B: private A {...}).* La clase B hereda los miembros *public* y *protected* de la clase A como miembros *private*.
- Los constructores, los destructores y las funciones amigas no pueden ser heredados. Tampoco se heredan las funciones y las variables estáticas, ni el operador de asignación sobrecargado.
- El constructor de la clase derivada debe llamar primero al constructor de la clase base. Al definir el constructor de la clase derivada se debe especificar un inicializador base, que es la forma de llamar a los constructores de las clases base y poder así inicializar las variables miembro heredadas. El inicializador base puede ser omitido en el caso de que la clase base tenga un constructor por defecto. En el caso de que el constructor de la clase base exista, al declarar un objeto de la clase derivada se ejecuta primero el constructor de la clase base.

```
Punto::Punto(float a, float b) {
//constructor de la clase base
     x=a; y=b;
}

class Circulo: public Punto {
     Circulo::Circulo(float r, float a, float b):Punto(a, b) {
     //llamada al constructor de la clase base
          radio=r;
     }
}
```

- Las funciones virtuales son métodos incluidos en varios niveles de una jerarquía de clases con el mismo nombre pero con distinta definición, que pueden actuar sobre objetos distintos sin tener que especificar el tipo exacto de los objetos.
- Se produce redefinición de funciones si la clase base contiene una función definida como virtual, y la clase que se deriva de ella contiene una función del mismo nombre que devuelve el mismo tipo de dato. Si ambas funciones devuelven tipos de dato distintos, se consideran funciones diferentes y no se invoca al mecanismo virtual.
- Para referirse explícitamente a la función virtual de la clase base se utiliza el operador de calificación (*claseBase :: f()*). Una clase con funciones virtuales se puede convertir en abstracta al declarar como pura una de sus funciones virtuales con la expresión *virtual void f ( ) = 0*.
- Por defecto, se utiliza la ligadura estática para determinar qué método se aplica a un determinado objeto. Se puede utilizar la ligadura dinámica si el método se define como virtual. Esto es imposible utilizando nombres de objetos, ya que siempre se aplica la función miembro de la clase correspondiente al nombre del objeto, y esto se decide en tiempo de compilación. Pero utilizando punteros a objetos, si se utilizan funciones virtuales es la clase del objeto al que apunta el puntero lo que determina la función que se llama.

5.3. Lenguaje puro: Java.

5.3.1. Características principales.

- Los lenguajes puros son lenguajes de programación orientados a objetos que no permiten realizar programación modular, con una menor flexibilidad y eficiencia del código.
- Java es un lenguaje basado en la sintaxis de C++ que sólo admite la creación de programas a partir de clases. Fue propuesto por Sun en 1995 y es un lenguaje compilado e interpretado de propósito general que soporta la programación concurrente y elimina la aritmética de punteros.

5.3.2. Clases, objetos, métodos y mensajes.

- En Java no existen las estructuras (struct) ni los punteros. Las clases están formadas por atributos y métodos, cuyo acceso puede ser puede ser *public*, *protected* y *private*, como en C++.
- Además de clases existen paquetes (*packages*) que agrupan una gran cantidad de clases. Si se quiere utilizar una clase situada dentro de un paquete se puede importar el paquete (*import*).
- No existen funciones amigas, y los objetos son creados implícitamente mediante constructores definidos con el mismo nombre de la clase, los cuales realizan la inicialización de los atributos antes de que los objetos sean creados. También pueden crearse explícitamente utilizando *new*. Los objetos siempre se referencian mediante variables del tipo objeto.
- No existen destructores en Java, se realiza una recolección automática de memoria. Existe un método *finalize* para casos especiales en los que se asigna memoria por un procedimiento distinto al normal (*new*). Este método se invoca justo antes de la recolección de memoria.
- Para llamar un método, se utiliza el identificador del objeto y '.' y los parámetros de los métodos siempre se pasan por valor excepto los objetos, que se pasan por referencia automáticamente.

5.3.3. Herencia y polimorfismo.

- En Java existe la herencia de tipos y la herencia de implementación, y está permitida solamente la herencia simple. Existe una clase raíz denominada Object.
- La herencia no cambia el nivel de protección de los miembros de la clase base. Como en C++, los constructores no pueden ser heredados. Se puede incluir una llamada explícita con la palabra *super* al constructor de la clase padre como primer enunciado del constructor de la subclase. En otro caso, se llamará implícitamente al constructor por omisión.

```
public class Punto {
    public Punto(double a, double b) {
        setPunto(a, b);
    }
}

public class Circulo extends Punto {
    public Circulo(double r, double a, double b) {
        super(a, b); //llamada al constructor de la clase base
        setRadio(r);
    }
}
```

- Una interfaz es el nombre que se le da a un conjunto de métodos que debe definir una clase que lo implemente, con independencia de su relación jerárquica, y no incluye ningún detalle de implementación. Se dice que una clase implementa una interfaz si define todos los métodos

abstractos de la interfaz. Los interfaces son las clases más abstractas que existen, puesto que consisten sólo en métodos públicos y abstractos, y atributos públicos y finales (constantes).

- Los métodos abstractos son métodos que no tienen implementación y que obligan a ser implementados en las subclases. Si una clase posee un método abstracto es una clase abstracta, de la cual no se pueden instanciar objetos. Si una clase tiene una superclase abstracta y no implementa algunos de sus métodos abstractos, entonces los hereda y se convierte en abstracta. Para poder crear objetos de esa clase, es necesario implementar todos los métodos abstractos heredados. Una clase se hace abstracta declarándola con la palabra clave *abstract*.
- En Java el método que tiene el mismo nombre que un método de la superclase anulará siempre automáticamente el método de la superclase, no hay que indicarlo explícitamente. Para referirse explícitamente a la función de la superclase se utiliza la palabra *super* (*super.toString()*).
- Por defecto, se utiliza la ligadura dinámica para determinar qué método se aplica a un determinado objeto. Si se utiliza una referencia de superclase para referirnos a un objeto de subclase, el programa escogerá el método de la subclase correcta en tiempo de ejecución mediante la búsqueda ascendente entre clases, comenzando en la clase del receptor, para encontrar un método que responda a un mensaje. Si se alcanza la última superclase sin encontrar ningún método, se emite un mensaje de error.