Programación III(Re incompleto) ☹

**Semana1**

Un **objeto** es una instancia de una clase. Una **clase** es un modelo o plantilla que define comportamiento y propiedades de un tipo de objeto. Cuando se crea un objeto se está creando una instancia concreta de esa clase. Esto significa que se esta asignando memoria para almacenar los datos específicos de ese objeto y se utilizaran los métodos definidos en esa clase.

El estado de un objeto se refiere al conjunto de valores que tiene el objeto en un determinado instante de tiempo.

**Referencia a objeto**: La variable de tipo no primitiva almacenan la referencia a un objeto, no contienen al objeto en sí. Se puede pensar como punteros al objeto.

**Abstracción**: Significa abstraer las propiedades esenciales de un objeto que lo distinguen de los demás tipos de objetos.

**Encapsulamiento:** Permite ocultar al mundo exterior la representación interna del objeto.

**Herencia:** Permite a los objetos crearse a partir de otros objetos. La clase base tiene todas las características comunes. Las subclases contienen las características de la clase padre + las características de la subclase misma. Si la subclase hereda características de una clase base, se trata de una herencia simple, si hereda de dos o mas clases base, herencia múltiple.

**Polimorfismo:** “Cualidad de tener más de una forma”. Se refiere al hecho de que una misma operación pueda tener diferentes comportamientos en diferentes objetos. Diferentes objetos reaccionan al mismo mensaje(método) de diferente manera.

**Sobrecarga de métodos:** Dos o más métodos pueden tener el mismo nombre si tiene diferente numero o tipo de parámetros. Cuando se convoca a un método, el compilador compara el numero y tipo de parámetro para encontrar el método que mejor se ajusta entre las signaturas disponibles. Los constructores también pueden sobrecargarse.

**Semana 2**

**Relación de agregación:** (composición débil). La destrucción del compuesto no conlleva la destrucción de los componentes. Los componentes pueden ser compartidos por varios compuestos. Esta relación implica que la clase contenedora es responsable de crear, mantener y destruir las instancias de la clase contenida, y que estas instancias pueden existir independientemente de la clase contenedora.

**Relación de composición:** Forma fuente de asociación donde la vida de la clase contenida debe coincidir con la vida de la clase contenedora. Los componentes no pueden ser compartidos por varios objetos compuestos. La supresión del objeto conlleva a la supresión de los componentes.

**Relación de dependencia:** Se da cuando una clase depende de otra para realizar alguna operación, pero no tiene una relación de propiedad o perteneciente a ella. Puede ocurrir a través de parámetros de método, variables locales o incluso como resultado de un llamado a métodos de otra clase.

**Relación de asociación:** Es un tipo de relación donde la clase “principal” esta relacionada con una clase “secundaria” de alguna manera. Esta relación implica que las instancias de la clase “principal” puedan estar relacionadas con cero, una o más instancias de clases “secundarias”. Esta relación puede ser unidireccional o bidireccional y puede tener multiplicidad uno a uno, uno a muchos, muchos a uno, muchos a muchos.

**Patrón de diseño SINGLETON:** Se asegura de que una clase tenga una única instancia, proporcionando acceso global a ella. Contiene un método \_instance de carácter estático que se encargará de instanciar la clase y un constructor privado que evitará que se creen nuevos objetos mediante new(). El método \_instance se encarga de comprobar si el atributo es null. En caso que lo sea, se invocará al constructor. Caso contrario devolverá el objeto existente.

**Semana3**

**Concepto de herencia:** El termino herencia se refiere al hecho de que una clase hereda las características (atributo, variables) y el comportamiento (métodos) de otra clase. Los atributos de la clase padre no deben volver a escribirse a no ser que se quieran ocultar. Los métodos heredados no deben volver a escribirse a no ser que se quieran sobrescribir.

**Herencia simple:** Una clase hereda solo de una clase base.

**Herencia múltiple:** Una clase hereda atributos y métodos de mas de una clase base. Java no admite esta herencia debido a problemas de ambigüedad y complejidad.

**Herencia multinivel:** Una clase hereda de una clase que a su vez hereda de otra clase, formando una cadena de jerarquía.

**Herencia de contrato o tipo:** la subclase adquiere el tipo de la superclase y se puede utilizar polimórficamente allí donde se pudiera utilizar la superclase.

**Herencia de implementación:** la subclase adquiere la implementación de la superclase en términos de sus campos y métodos accesibles. (clase tipo INTERFACE).

**Delegación:** Se caracteriza por “Reutilización selectiva”. En lugar de realizar la tarea por sí mismo, el objeto delegante (delegador) pasa la solicitud a un objeto delegado que tiene la capacidad de manejarla. La delegación permite la composición de objetos en lugar de la herencia, lo que significa que un objeto puede utilizar funcionalidades de otro objeto sin heredar directamente de él. Esto promueve la reutilización de código y la separación de preocupaciones, ya que cada objeto puede concentrarse en realizar tareas específicas. La delegación se puede implementar de varias formas, incluidas la pasada de referencias de objetos como parámetros de método, la implementación de interfaces comunes y la creación de objetos internos para manejar tareas específicas.

**Clase concreta:** Clase a la cual se puede instanciar directamente y que no contiene métodos abstractos. Proporciona una implementación completa de todos los métodos definidos en su base o interfaz.

**Clase abstracta:** Es una clase que no se puede instanciar directamente y se usa como plantilla para otras clases. Puede tener métodos abstractos, los cuales deben ser implementados en la clase que heredan. Las clases abstractas pueden tener métodos concretos también.

**Semana4**

**Principio de Sustitución de Liskov:** Debe ser posible utilizar cualquier objeto instancia de una subclase en lugar de cualquier objeto instancia de su superclase sin que la semántica del programa escrito en los términos de la superclase se vea afectado. “Barbara Liskov”.

**Polimorfismo:** Se refiere a la capacidad de objetos de diferentes clases de responder al mismo mensaje de manera distinta. Permite tratar diferentes objetos de una manera uniforme, independientemente de su clase concreta.

**Polimorfismo de sobrecarga:** Permite definir operadores cuyos comportamientos varían de acuerdo a los parámetros que se es aplican.

**Polimorfismo paramétrico:** Capacidad para definir varios métodos utilizando el mismo nombre, pero usando los parámetros diferentes (nombre y/o tipo).

**Polimorfismo de inclusión o subtipado:** Habilidad para redefinir un método en clases que se heredan de una clase base se llama especialización. Se puede llamar a un método de objeto sin tener que conocer su tipo intrínseco. Permite no tomar en cuenta detalles de las clases especializadas de una familia de objetos, enmascarándolos con una interfaz común.

**Concepto de enlace Tardío:** La herencia permite el tratamiento de un objeto como si fuese de su tipo o del tipo de su clase padre. Esta característica es crítica porque permite que varios tipos sean tratados como si fueran uno solo. En tiempo de compilación no se conoce cuál será el objeto usado al ejecutar cierto método, por lo tanto, se tiene que resolver en tiempo de ejecución.

**Ligadura:** Conexión de una llamada a un método.

**Ligadura temprana -> compilación**

**Ligadura tardía -> ejecución.**

**Concepto de contrato:** El contrato de un método establece bajo qué condiciones el método tendrá éxito y cuál será el resultado una vez que se termine su ejecución. Para esto se declara Precondiciones y Postcondiciones.

**Precondición:** Conjunto de suposiciones, expresadas como condiciones que deben ser verdaderas para que el método se ejecute con éxito. Estas pueden referirse a: El estado del objeto que va ejecutar el método (el valor de sus atributos), El estado de algún elemento del mundo con el cual el objeto tenga una asociación, Condiciones sobre los parámetros de entrada entregados al método.

**Postcondición:** La descripción del resultado obtenido después de ejecutar un método la llamamos postcondición. Esta se expresa en términos de un conjunto de condiciones que deben ser verdaderas después de que el método ha sido ejecutado, siempre y cuando no se haya lanzado una excepción. Estas condiciones hacen referencia a: Una descripción del valor de retorno, Una descripción del estado del objeto después de haber ejecutado el método.

**Patrones de diseño:**

**Patrón Factory:**

**Semana 5**

**Semana 7: Excepciones**

Cuando un programa baja viola las restricciones semánticas del lenguaje, la maquina virtual java comunica este hecho mediante una excepción.

Un método se dice que es capaz de tratar una excepción si ha previsto el error y prevé también las operaciones para “Recuperar” el programa de ese error.

Los programas pueden lanzar excepciones explícitamente mediante la instrucción **throw**, que facilita la devolución de un “código de error” al método que invocó el método que causó el error.

Para esto es necesario declarar todas las posibles excepciones que es posible generar en el método, utilizando la clausula **throws** en la declaración de los métodos.

Para crear una Excepción es necesario crear un objeto de tipo **Exception** o alguna de sus clases hijas, y lanzarlo mediante la instrucción **throw.**

Una manejador de excepciones es una porción de código que se encarga de tratar las posibles excepciones que se puedan generar.

**Try:** para que un método sea capaz de tratar una excepción es declarar las instrucciones dentro de un bloque **try.**

Cualquier excepción que se produzca dentro de este bloque será analizado por el bloque ( o bloques) **catch**. En el momento que se produce la excepción, se abandona el bloque **try** y las instrucciones que sigan no serán ejecutadas.

**Catch:** AL producirse la excepción dentro de un bloque **try**, la ejecución del programa se pasa al primero bloque **catch**. Si la clase de la excepción se corresponde con la clase o alguna subclase declarada en el bloque **catch,**  se ejecuta el bloque de instrucciones catch.

Sin embargo, es mejor utilizar excepciones lo mas cercana al tipo de error previsto, ya que lo que se pretende es recuperar al programa de alguna condición de error.

**Finally:** Este bloque se utiliza para ejecutar instrucciones sea cual sea la excepción que se produzca. Se ejecutará sea cual sea el caso, incluso si no se produce ninguna excepción. Sirve para no tener que repetir código en el bloque try y en los bloques catch.

Try{

Bloque de instrucciones;

}

Catch( Excepcion e){

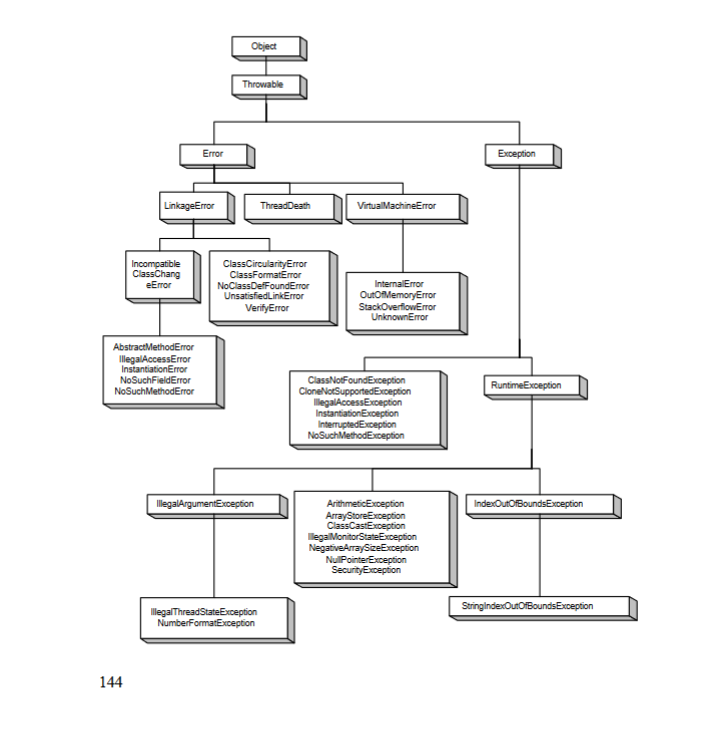
Recuperar error;

}

Finally{

Instrucción que se ejecuta sea cual sea el caso( lanza o no excepción);

}



**Semana 8: Colecciones**

Son receptáculos que permiten almacenar y organizar objetos de forma útil para un acceso eficiente. **Set, SortedSet, Map, SortedMap,List, ListIterator, HashSet, TreeSet, ArrayList, LinkedList, HashMap, TreeMap,Queue, PriorityQueue.**

La clase Object es la raíz de jerarquía de las clases, todas se extienden directa o indirectamente de esta clase.

**Semana 9: Concurrencia**

La concurrencia se refiere a la ejecución simultanea de múltiples hilos de ejecución dentro de un programa. Esto es útil en situaciones donde hay operaciones que pueden ocurrir de manera independiente y no necesitan esperar a que otras finalicen para continuar su ejecución. Es importante para mejorar el rendimiento y la capacidad de respuesta, así como el aprovechamiento de los recursos disponibles.

Sin embargo, puede introducir desafíos, como la sincronización de acceso a recursos compartidos y la posibilidad de condiciones de carrera, para así evitar problemas como bloqueos y corrupción de datos.

Java proporciona un API para el uso de hilos: la clase Thread. Cuando arranca un programa existe un hilo principal: Main.

Para crear un hilo se crea un objeto de tipo Thread. -> **Thread hilo = new Thread();**

Para ejecutarlo se invoca al método start(). hilo.start(); ( solo se puede invocar el método una vez para cada hilo o se lanza excepción **IllegalThreadStateException**).

Una vez que se hace el llamado, este método llama a run() que es propio de la clase Thread, haciendo que el hilo este en estado RUNNEABLE.

Una vez iniciado el hilo, este puede invocar los métodos **sleep(), wait(), Notify(), notifyAll().**

Wait : este método deja esperando un hilo hasta que ocurra alguna condición.

Notify: avisa a un hilo que algo ocurrió para satisfacer la condición

NofityAll: avisa a todos lo hilos que algo ocurrió con la condición.

La condición debe siempre estar en un ciclo de repetición.

Cuando el método run() finaliza, el hilo termina y pasa a estado **DEAD**.

La finalización de un hilo suele ocurrir cuando termina de ejecutar el método run(), pero hay otras razones por las que puede terminar, como haberse generado alguna excepción, se llama al método **stop(),** o si se llama al método **detroy()**;

Otra forma de implementar THREAD es con la interface Runneable. Para comenzar la ejecución del método run() de cualquier objeto que implemente la interface Runneable, se crea un nuevo objeto de la clase Thread pasándole como parámetro al constructor el objeto que se desea ejecutar. Cuando se llame al método start(), el método que se ejecutara será el método run() del objeto pasado en el constructor.

Cuando varios hilos comparten un mismo objeto y modifican su estado, puede ocurrir inconsistencia. Al estar en un ambiente concurrente, si no se establece una sincronización ( synchronized) en el acceso al objeto compartido, el resultado será imprevisto. Se le llama **Sección Critica** a los segmentos de código dentro de un programa que acceden a zonas de memoria comunes desde distintos hilos.

**Monitores:** son objetos destinados a ser usador sin peligro por mas de un hilo de ejecución. Sus métodos son ejecutados con exclusión mutua, es decir, un hilo como máximo puede estar ejecutando cualquiera de sus métodos.Un monitor tiene cuatro componentes, inicialización, métodos privados, métodos del monitor, cola de entrada.

Exiten los métodos synchroniced y las sentencias.

**Public synchroniced void nombre\_func(){  
}**

**Public void nombre\_func2(){**

**Synchroniced(bloque){**

**}**

**}**

**Semaforo:** Es un objeto que permite o impide el paso de otros objetos. Este contiene un numero de permisos que son los que los objetos intentan obtener para la realizar alguna operación. Tiene un contador interno en el que lleva la cuenta de la cantidad de permisos que fueron otorgados.

Los métodos de un semáforo son : **acquire(), y reléase()**, el primero intenta obtener un permiso, si hay disponible termina al instante, de lo contrario, es bloqueado hasta que haya un permiso. El segundo método, libera un permiso y lo devuleve al semáforo.