FORMATO DE GUÍA METODOLÓGICA	E-MAIL: lbegnini@itsjpapon.edu.ec Av. Marieta de Veintimilla Telf: 593 – 2 – 2356 368 Quito - Ecuador
	FORMATO: DI-ISTJ-FOR-048

1. IDENTIFICACIÓN DE GUÍA METODOLÓGICA

Nombre de la Asignatura:	**Componentes del Aprendizaje**	
PROGRAMACIÓN ORIENTADA A OBJETOS II		Componente docencia 36 Componente de prácticas de aprendizaje 30 Componente de aprendizaje autónomo 53

Resultado del Aprendizaje:

- ✓ Diseñar e implementar programas utilizando el paradigma de la programación orientada a objetos
- ✓ Entrega la representación gráfica de clases (atributos y métodos) y herencia a partir de problemas planteados
- ✓ Elabora un compendio de programas documentados que contengan la implementación de:
- ✓ Clases
- ✓ Hilos
- ✓ Clases genéricas
- ✓ Persistencia de objetos

COMPETENCIAS Y OBJETIVOS

OBJETIVOS:

- ✓ Capacidad para diseñar, desarrollar, seleccionar y evaluar aplicaciones y sistemas informáticos, asegurando su fiabilidad, seguridad y calidad, conforme a principios éticos y a la legislación y normativa vigente
- ✓ Conocimiento y aplicación de los procedimientos algorítmicos básicos de las tecnologías informáticas para diseñar soluciones a problemas, analizando la idoneidad y complejidad de los algoritmos propuestos
- ✓ Capacidad para analizar, diseñar, construir y mantener aplicaciones de forma robusta, segura y eficiente, eligiendo el paradigma y los lenguajes de programación más adecuados
- ✓ Conocimiento y aplicación de las herramientas necesarias para el almacenamiento, procesamiento y acceso a los Sistemas de información, incluidos los basados en web

COMPETENCIAS

- ✓ Diseñar e implementar programas para el manejo de concurrencia.
- ✓ Aplica los conceptos fundamentales de la Programación Orientada a Objetos (POO en la resolución de problemas

- ✓ Implementa soluciones bajo el paradigma orientado a objetos, utilizando un lenguaje de programación orientado a objetos.
- ✓ Aplica los conceptos de programación orientada a objetos referentes al manejo de archivos y bases de datos.

Docente de Implementación:	
Ing. Byron Giovanny Cholca MSc.	**Duración:** 50 Horas

Unidades	Competencia	Resultados de Aprendizaje	Actividades	Tiempo de Ejecución
Unidad 1: Clases	Diseñar e implementar programas utilizando el paradigma de la programación orientada a objetos.	Identificar los componentes básicos de una clase.	Práctica en Clase Diapositivas Videos	**5 horas**
Unidad 2: Threading	Maneja el concepto y utilidad de los Hilos en la POO.	Conocer el funcionamiento y operatividad de los Hilos.	Practica en 5 clase, realizan varios ejercicios para obtener resultados pantalla, operaciones matemáticas horas	**5 horas**

Unidad 3: Java Generics	Entrega en medio electrónico el ejecutable de la aplicación y código fuente.	Crea y manipula clases genéricas.	Ejercicios Clase, con ejemplos casos reales. Debate y Prácticas clase	**5 horas**
Unidad 4: Persistencia de Objetos	Maneja el concepto de objetos y persistencia	Participa activamente en un equipo de trabajo desarrollando aplicaciones que empleen conocimientos de programación orientada a objetos.	Ejercicios de programación en clase, practicas guiadas en la plataforma	**5 horas**

2. CONOCIMIENTOS PREVIOS Y RELACIONAD

Correquisitos

1.-Identificar los conceptos básicos de programación orientada a objetos.

2.-Realizar prácticas en clase que permitan plantear ejercicios de dominio matemático y estadístico.

3.-Realizar prácticas en clase que permitan conocer lista comandos, tipos de datos verdaderos y errados. (Comprobar en la computadora)

4. Conocer las estrategias para diferenciar las variables, constantes y operadores que se aplican en el sistema informático.

5. Diseñar y ejecutar presentaciones que plantean fórmulas de ingeniería para resolver con algoritmos y codificación en cualquier lenguaje de Programación Orientado a Objetos.

3. UNIDADES TEÓRICAS

- **Desarrollo de las Unidades de Aprendizaje (contenidos)**

A. Base Teórica

UNIDAD I: Clases

TEMA 1: Introducción.

Dentro de la programación orientada a objetos, las clases son un pilar fundamental. Estas nos van a permitir abstraer los datos y sus operaciones asociadas como si tuviéramos en frente una caja negra. La mayoría de los lenguajes de programación modernos incluyen la posibilidad de usar clases y las ventajas que ello nos aporta.

Una clase es la descripción de un conjunto de objetos similares; consta de métodos y de datos que resumen las características comunes de dicho conjunto. En un lenguaje de programación orientada a objetos se pueden definir muchos objetos de la misma clase de la misma forma que, en la vida real, haríamos galletas (objeto) con el mismo molde (clase) solo que, para entenderlo mejor, cada galleta tendría igual forma, pero es posible que tenga distinto sabor, textura, olor, color, etc.

Un objeto es un simple elemento, no importa lo complejo que pueda ser. Una clase, por el contrario, describe una familia de elementos similares. En la práctica, una clase es un esquema o plantilla que se usa para definir o crear objetos.

A partir de una clase, se puede definir un número indeterminado de objetos. Cada uno de estos tendrá un estado particular propio (una pluma estilográfica puede estar llena, otra puede estar medio llena y otra totalmente vacía) y otras características (como su color), aunque compartan algunas operaciones comunes (como escribir o llenar su depósito de tinta).

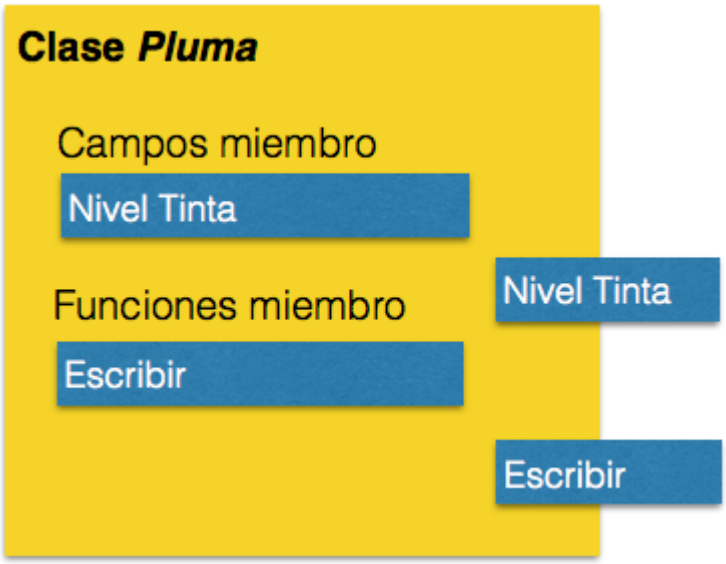


Diagrama de una Clase

TEMA 2: Métodos Setters y Getters, paso de parámetros.

Para los que no han oído hablar sobre éstos, los getters y los setters son construcciones habituales de los objetos que permiten acceder a valores o propiedades, sin revelar la forma de implementación de las clases. En pocas palabras, permiten encapsular los objetos y evitar mantenimiento de las aplicaciones cuando la forma de implementar esos objetos cambia.

Utilidad aparte, en la práctica son simplemente métodos que te permite acceder a datos de los objetos, para leerlos o asignar nuevos valores. El setter lo que hace es asignar un valor y el getter se encarga de recibir un valor.

Sin embargo, los getters y los setters de los que vamos a hablar en Javascript son un poco distintos de los tradicionales.

En el siguiente ejemplo vamos a profundizar un poco en los getters y los setters de Javascript, implementando un nuevo caso de uso que seguro que nos despejará posibles dudas. Además veremos cómo funciona set, para asignar un valor "computado".

En nuestro ejemplo tenemos un objeto intervalo que para su implementación define un valor máximo y un valor mínimo. Pero queremos disponer de un mecanismo que nos ofrezca los valores, enteros, comprendidos en ese intervalo. Para practicar con los get y set vamos a hacerlo mediante una propiedad computada.

Aunque primero vamos a ver cómo se resolvería de manera tradicional, con un método de toda la vida.

```
var intervalo = {
  valorMinimo: 3,
  valorMaximo: 4,
  valoresContenidos: function() {
    var contenidos = [];
    for(var i=this.valorMinimo; i<=this.valorMaximo; i++) {
      contenidos.push(i);
    }
    return contenidos;
  }
}
```

Para acceder a los valores contenidos tendríamos que ejecutar el método correspondiente:

```
var valores = intervalo.valoresContenidos();
```

Ahora vamos a ver cómo realizar esto mismo, pero con una propiedad "get", computada.

```
var intervalo = {
  valorMinimo: 3,
  valorMaximo: 7,
  get valoresContenidos() {
    var contenidos = [];
    for(var i=this.valorMinimo; i<=this.valorMaximo; i++) {
      contenidos.push(i);
    }
    return contenidos;
  },
}
```

Ahora llamaríamos al método de esta manera.

```
var valores = intervalo.valoresContenidos;
```

Nos queda por ver un ejemplo con el la construcción "set". En este caso lo que vamos a hacer es recibir un parámetro con aquello que se tenga que asignar y realizar los cambios en las propiedades de los objetos que sea pertinente.

```
var intervalo = {
  valorMinimo: 3,
  valorMaximo: 7,
  get valoresContenidos() {
    var contenidos = [];
    for(var i=this.valorMinimo; i<=this.valorMaximo; i++) {
      contenidos.push(i);
    }
    return contenidos;
  },
  set valoresContenidos(arrayValores) {
    arrayValores.sort();
    this.valorMinimo = arrayValores[0];
    this.valorMaximo = arrayValores[arrayValores.length - 1];
  }
}
```

El setter se define de manera similar al getter. En este caso como hemos dicho se usará una función que recibe un dato que haya que setear. La gracia está en el modo de uso de un setter, que no es un método tradicional en el que se enviarían parámetros, sino que se invoca mediante una asignación.

TEMA 3: Variables y constantes de clase. Uso static.

La definición formal de los elementos estáticos (o miembros de clase) nos dice que son aquellos que pertenecen a la clase, en lugar de pertenecer a un objeto en particular. Recuperando concetos básicos de orientación a objetos, sabemos que tenemos:

Clases: definiciones de elementos de un tipo homogéneo.

Objetos: concreción de un ejemplar de una clase.

En las clases defines que tal objeto tendrá tales atributos y tales métodos, sin embargo, para acceder a ellos o darles valores necesitas construir objetos de esa clase. Por ejemplo, una casa tendrá un número de puertas para entrar, en la clase tendrás definida

que una de las características de la casa es el número de puertas, pero solo concretarás ese número cuando construyas objetos de la clase casa. Un coche tiene un color, pero en la clase solo dices que existirá un color y hasta que no construyas coches no les asignarás un color en concreto. En la clase cuadrado definirás que el cálculo del área es el "lado elevado a dos", pero para calcular el área de un cuadrado necesitas tener un objeto de esa clase y pedirle que te devuelva su área.

Ese es el comportamiento normal de los miembros de clase. Sin embargo, los elementos estáticos o miembros de clase son un poco distintos. Son elementos que existen dentro de la propia clase y para acceder los cuales no necesitamos haber creado ningún objeto de esa clase. Ósea, en vez de acceder a través de un objeto, accedemos a través del nombre de la clase.

Para definir y usar miembros estáticos generalmente se usa el modificador "static" para definir los miembros de clase, al menos así se hace en Java o PHP, C#, pero podrian existir otros métodos en otros lenguajes.

```
class AutobusMetropolitano {
    public static precio = 1.20;
}
```

Luego podríamos acceder a esos elementos a través del nombre de la clase, como se puede ver a continuación:

```
if(1.5 >= AutobusMetropolitado.precio){
    //todo bien
}
```

Ese código sería en el caso que estés accediendo al atributo estático desde fuera de la clase, pero en muchos lenguajes puedes acceder a esos atributos (o métodos) desde la vista privada de tus clases (código de implantación de la clase) a través de la palabra "self".

```
class AutobusMetropolitano {
    static precio = 1.20;
    //...

    public function aceptarPago(dinero){
        if (dinero < self.precio){
            return false;
        }
        // ...
    }
}
```

En el caso de los métodos estáticos la forma de crearlos o usarlos no varía mucho, utilizando el mismo modificador "static" al declarar el método.

```
class Fecha{
    //...
public static function valida(ano, mes, dia){
        if(dia >31)
            return false;
        // ...
    }
    //...
}
```

Ahora para saber si un conjunto de año mes y día es válido podrías invocar al método estático a través del nombre de la clase.

```
if(Fecha.valida(2014, 2, 29)){
    // es valida
}else{
    // no es valida
}
```

Como en el caso de los atributos de clase, también podrías acceder a métodos de clase con la palabra "self" si estás dentro del código de tu clase.

TEMA 4: Manejo de archivos, Excepciones.

Manejo de archivos

El manejo de archivos (persistencia), es un tema fundamental en cualquier lenguaje de programación. Pues nos permite interactuar con los dispositivos de almacenamiento externo para poder mantener la información en el tiempo. Java no es una excepción. Cuando se desarrollan applets para utilizar en red, hay que tener en cuenta que la entrada/salida directa a fichero es una violación de seguridad de acceso. Muchos usuarios configurarán sus navegadores para permitir el acceso al sistema de ficheros, pero otros no. Por otro lado, si se está desarrollando una aplicación Java para uso interno, probablemente será necesario el acceso directo a ficheros.

Para realizar operaciones sobre los ficheros, necesitamos contar con la información referente sobre un fichero (archivo). La clase File proporciona muchas utilidades relacionadas con ficheros y con la obtención de información básica sobre esos ficheros. Para crear un objeto File nuevo, se puede utilizar cualquiera de los tres constructores siguientes:

Constructores de la clase File	ejemplo
public File(String pathname) ;	new File("/carpeta/archivo");
public File(String parent, String child) ;	new File("carpeta", "archivo");
public File(File parent, String child) ;	new File(new File("/carpeta"), "archive")
public File(URI uri) ;	new File(new URI(str);

El constructor utilizado depende a menudo de otros objetos File necesarios para el acceso. Por ejemplo, si sólo se utiliza un fichero en la aplicación, el primer constructor es el mejor. Si en cambio, se utilizan muchos ficheros desde un mismo directorio, el

segundo o tercer constructor serán más cómodos. Y si el directorio o el fichero es una variable, el segundo constructor será el más útil.

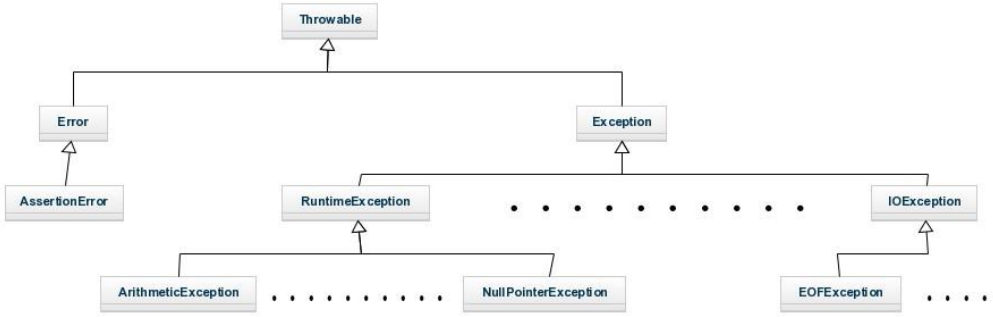
```
import java.io.*;
class InformacionFichero {
    public static void main( String args[] ) throws IOException {
        if( args.length > 0 ){
            for( int i=0; i < args.length; i++ ){
                File f = new File( args[i] );
                System.out.println( "Nombre: "+f.getName() );
                System.out.println( "Camino: "+f.getPath() );
                if( f.exists() ){
                    System.out.print( "Fichero existente " );
                    System.out.print( (f.canRead() ? "y se puede Leer":""));
                    System.out.print( (f.canWrite()?"se puese Escribir":""));
                    System.out.println( "." );
                    System.out.println( "La longitud del fichero son "
                    +f.length()+" bytes" );
                   }
                else{
                  System.out.println( "El fichero no existe." );
                }
                  }
            }
        else
            System.out.println( "Debe indicar un fichero." );
      }
}
```

Excepciones

En Java los errores en tiempo de ejecución (cuando se esta ejecutando el programa) se denominan excepciones, y esto ocurre cuando se produce un error en alguna de las instrucciones de nuestro programa, como por ejemplo cuando se hace una división entre cero, cuando un objeto es 'null' y no puede serlo, cuando no se abre correctamente un fichero, etc. Cuando se produce una excepción se muestra en la pantalla un mensaje de error y finaliza la ejecución del programa. En Java (al igual que en otros lenguajes de programación), existen muchos tipos de excepciones y enumerar cada uno de ellos seria casi una labor infinita. En lo referente a las excepciones hay que decir que se aprenden a base experiencia, de encontrarte con ellas y de saber solucionarlas.

Cuando en Java se produce una excepción se crear un objeto de una determina clase (dependiendo del tipo de error que se haya producido), que mantendrá la información sobre el error producido y nos proporcionará los métodos necesarios para obtener dicha información. Estas clases tienen como clase padre la clase Throwable, por tanto, se mantiene una jerarquía en las excepciones. A continuación, mostramos algunas de las

clases para que nos hagamos una idea de la jerarquía que siguen las excepciones, pero existen muchísimas más excepciones que las que mostramos:



A continuación, vamos a mostrar un ejemplo de cómo al hacer una división entre cero, se produce una excepción. Veamos la siguiente imagen en el que podemos ver un fragmento de código y el resultado de la ejecución del código:

```
1  package Main;
2
3  public class Main {
4
5      public static int numerador = 10;
6      public static Integer denominador = 0;
7      public static float division;
8
9      public static void main(String[] args) {
10
11     (1) System.out.println("ANTES DE HACER LA DIVISIÓN");
12
13     (2) division = numerador / denominador;
14
15     (3) System.out.println("DESPUES DE HACER LA DIVISIÓN");
16     }
17 }
```

```
Problems  Javadoc  Declaration  Console
<terminated> Main (5) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_20.j
ANTES DE HACER LA DIVISIÓN
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Main.Main.main(Main.java:13)
```

TEMA 5: Constructores, Modificadores de acceso.

Los modificadores de acceso permiten al diseñador de una clase determinar quien accede a los datos y métodos miembros de una clase. Los modificadores de acceso preceden a la declaración de un elemento miembro de la clase (ya sea atributo o método) y son cuatro:

- default o package-private (cuando no se escribe nada)
- public
- protected
- private

Si no especificamos ningún modificador de acceso se utiliza el nivel de acceso por defecto, que consiste en que el elemento puede ser accedido sólo desde las clases que pertenezcan al mismo paquete.

El nivel de acceso **public** permite a acceder al elemento desde cualquier clase, independientemente de que esta pertenezca o no al paquete en que se encuentra el elemento.

private, por otro lado, es el modificador más restrictivo y especifica que los elementos que lo utilizan sólo pueden ser accedidos desde la clase en la que se encuentran. Es importante destacar que private convierte los elementos en privados para otras clases, no para otras instancias de la clase. Es decir, un objeto de una determinada clase puede acceder a los miembros privados de otro objeto de la misma clase, por lo que algo como lo siguiente sería perfectamente válido:

```
1 class MiObjeto {
2   private short valor = 0;
3
4   MiObjeto(MiObjeto otro) {
5     valor = otro.valor;
6   }
7 }
```

Los atributos y métodos protected se comportan exactamente igual que los atributos y métodos privados cuando son llamados desde los métodos de cualquier otra clase, excepto para métodos de las clases del mismo paquete o métodos de sus subclases con

independencia del paquete al que pertenezcan, para las que se comporta como un miembro público.

La siguiente tabla muestra cómo es el acceso en los distintos casos:

	El atributo o método declarado como:			
Puede ser accedido desde:	public	protected	default	private
Su misma clase	SI	SI	SI	SI
Cualquier clase que esté en el mismo paquete	SI	SI	SI	NO
Cualquier subclase que esté en otro paquete	SI	SI	NO	NO
Cualquier otra clase que esté en otro paquete	SI	NO	NO	NO

Los modificadores de acceso permiten al diseñador de la clase delimitar la frontera entre lo que es accesible para los usuarios de la clase, lo que es estrictamente privado y 'no importa' a nadie más que al diseñador de la clase e incluso lo que podría llegar a importar a otros diseñadores de clases que quisieran alterar, completar o especializar el comportamiento de la clase.

Con el uso de estos modificadores se consigue uno de los principios básicos de la Programación Orientada a Objetos, que es la encapsulación: las clases tienen un comportamiento bien definido para quienes las usan conformado por los elementos que tienen un acceso público, y una implementación oculta formada por los elementos privados, de la que no tienen que preocuparse los usuarios de la clase.

Los otros dos modificadores, **protected** y el acceso **por defecto** complementan a los otros dos. El primero es muy importante cuando se utilizan relaciones de herencia entre las clases y el segundo establece relaciones de 'confianza' entre clases afines dentro del mismo paquete. Así, la pertenencia de las clases a un mismo paquete es algo más que una clasificación de clases por cuestiones de orden.

Cuando se diseñan clases, es importante pararse a pensar en términos de quien debe tener acceso a que. Qué cosas son parte de la implantación y

deberían ocultarse (y en que grado) y que cosas forman parte de la interface y deberían ser públicas.

TEMA 6: Sobrecarga de constructores

Algunas veces hay una necesidad de inicializar un objeto de diferentes maneras. Esto se puede hacer usando la sobrecarga de constructor. Hacerlo le permite construir objetos de varias maneras. Tomemos un ejemplo para comprender la necesidad de sobrecargar constructores. Considere el siguiente programa:

```
//Demostración de Sobrecarga de constructores
class MiClase{
    int x;
    MiClase(){
        System.out.println("Dentro de MiClase().");
        x=0;
    }
    MiClase(int i){
        System.out.println("Dentro de MiClase(int).");
        x=i;
    }
    MiClase(double d){
        System.out.println("Dentro de MiClase(double).");
        x=(int)d;
    }
    MiClase(int i, int j){
        System.out.println("Dentro de MiClase(int, int).");
        x=i*j;
    }
}
class DemoSobrecargaConstructor{
    public static void main(String[] args) {
        MiClase t1=new MiClase();
        MiClase t2=new MiClase(28);
        MiClase t3=new MiClase(15.23);
        MiClase t4=new MiClase(2,4);
        System.out.println("t1.x: "+ t1.x);
        System.out.println("t2.x: "+ t2.x);
        System.out.println("t3.x: "+ t3.x);
        System.out.println("t4.x: "+ t4.x);
    }
}
```

Salida:

```
Dentro de MiClase().
Dentro de MiClase(int).
Dentro de MiClase(double).
Dentro de MiClase(int, int).
t1.x: 0
t2.x: 28
t3.x: 15
t4.x: 8
```

MiClase() está sobrecargado de cuatro maneras, cada una construyendo un objeto de manera diferente. El constructor apropiado se llama en función de los parámetros especificados cuando se ejecuta new. Al sobrecargar el constructor de una clase, le da flexibilidad al usuario de su clase en la forma en que se construyen los objetos.

UNIDAD 2: Threading

TEMA 1: Conceptos

En ingles se escribe Thread y la traducción más clara es hilo. Si tenemos un dispositivo móvil al que le damos click en el icono de una aplicación para que se abra, entonces el dispositivo seguramente ya tiene varias aplicaciones abiertas, así que podríamos decir que cada aplicación es un hilo. Queda claro que los hilos son independientes, es decir las aplicaciones, lo que haga una no debería de afectar a la otra, ahora entonces cuando estamos en una aplicación vemos que tenemos cosas como descarga de imágenes, accesos a las bases de datos, etc. Estas tareas normalmente tienen un problema: no sabemos cuánto tiempo van a tardar, de hecho, corremos siempre el riesgo de que la tarea no se complete. Imagina entonces que en una aplicación queremos descargar una imagen y contamos con una conexión a internet que no es la mejor, además es una imagen muy pesada, el usuario al no ver que la imagen se muestra en su teléfono puede pensar que la aplicación no funciona o que simplemente se quedó atorado, pero en realidad solo tomará más tiempo de lo debido. Entonces tenemos un problema ya que cada aplicación es un solo hilo y necesitamos el hilo para trabajar con la imagen y ya

que esta tarea está pendiente no podremos avanzar. Los hilos nos ayudan a resolver estos problemas.

Crear un hilo es muy sencillo, este sería el código para hacerlo con Java:

```
mythread = new Thread() {
    public void run() {
        try {
            System.out.println("...");

            Thread.sleep(1000);

            System.out.println("...");
        } catch(InterruptedException e) {
            System.out.println(e);
        }
    }
};

mythread.start();
```

En este código lo que hacemos es ejecutar un hilo y lo mandamos a dormir por 1 segundo, ya que son milisegundos y 1000 es igual a 1 segundo. Por lo tanto, como vemos no es mayor complejidad definirlos, no cambia mucho el código entre python, java y otros lenguajes, lo importante es conocer para que nos pueden servir y en resumen es para dividir y vender los problemas a los que enfrentamos.

TEMA 2: Programación concurrente

Para hablar de concurrencia, primero vamos a hablar de monotarea y multitarea.

Normalmente una aplicación tiene un hilo que permite ejecutar una monotarea y, si solo se trabaja con este hilo, los demás procesos de la aplicación quedan bloqueados hasta que este hilo termine de ejecutarse. Multitarea, por su parte, consiste en manejar varios hilos; es decir, mientras un hilo está ejecutando un proceso, otro hilo puede ejecutar al mismo tiempo otro proceso de manera independiente. Una vez teniendo claros estos dos términos, vamos a hablar de la concurrencia que se asocia a la multitarea.

Esto se da cuando dos o más tareas se desarrollan en el mismo intervalo de tiempo.

Es importante aclarar que, aunque se desarrollen en el mismo intervalo de tiempo, no necesariamente se ejecutan en el mismo instante. A este otro concepto se le denomina paralelismo.

En el párrafo anterior, estábamos hablando de que cada hilo ejecuta una tarea. Vamos a profundizar un poco en este aspecto.

Un hilo es un objeto con capacidad de ejecutar una tarea o, dicho de otra manera, ejecutar concurrentemente el método run () de java.

Los hilos pueden tener diferentes estados entre los cuales están nuevo, ejecutable, bloqueado, en espera y terminado (este último se puede lograr con el método stop (), que destruye el hilo de manera brusca, lo que no es una buena práctica porque puede causar inconsistencias en el sistema y es un método depreciado).

Los hilos son herramientas muy poderosas para manejar aplicaciones multitarea, lo que ayuda a mejorar o mantener el rendimiento de un sistema que gestiona peticiones concurrentes.

Al mismo tiempo, al trabajar con hilos debe tenerse mucho cuidado porque se pueden generar starvation, deadlock o inconsistencias en las operaciones del sistema; por esta razón, vamos a entender un poco más la manera en la que los hilos funcionan.

En Java, los hilos manejan su propia memoria, y la memoria principal de java es externa a esta. El ciclo de vida para que un hilo tome una variable y cambie su valor en la memoria principal consiste en hacer las siguientes operaciones:

- read() para leer el dato de memoria principal. load() para cargar este dato a la memoria del hilo use() para usar el dato.
- assign() para asignar el nuevo valor.
- store() para llevar este dato a la memoria principal.
- write(), para modificar el valor en la memoria principal.

Los hilos corren de manera concurrente; pero no tienen un orden específico, a menos de que esto se requiera y se sincronicen los hilos.

Esto es importante saberlo porque debemos ser conscientes de que, como los hilos corren de manera independiente, dos hilos al tiempo pueden modificar una variable Y esto, a su vez, puede causar que, si con un hilo asignamos el valor X a una variable Y, vamos a hacer una operación teniendo presente X como el valor de dicha variable.
Sin embargo, si posterior a ello otro hilo le asignó el valor Z, podemos ejecutar operaciones inconsistentes con una variable diferente a la que pensamos que íbamos a trabajar. Es por ello que debemos sincronizar hilos.

```
@Override
    public void run() {
        StringBuilder text = new StringBuilder();

        for(int count=0; count <= 30; count++) {
            text.setLength(0);
            text.append("Valor del hilo ");
            this.value = this.value+100;
            text.append(this.name).append(" es
").append(this.value);
            System.out.print(text.toString());
        }
        text.setLength(0);
        text.append("valor final del hilo
").append(name).append(" es ").append(this.value);
        System.out.print(text.toString());
    }
```

En este método, vemos que el método run() de un hilo va a sumar 31 veces un valor de 100 en una variable value; posterior a ello, va a mostrar el valor final de la variable value para ese hilo.

TEMA 3: Hilos e Interrupción

Método interrupt(): si algún hilo está en estado de suspensión o espera, entonces usando el método interrupt(), podemos interrumpir la ejecución de ese hilo mostrando InterruptedException. Un hilo que está en estado de espera o de espera se puede interrumpir con la ayuda del método interrupt() de la clase Thread.

Ejemplo: suponga que hay dos hilos y si uno de los hilos está bloqueado en una invocación de los métodos wait(), wait (long) o wait (long, int) de la clase Object, o de join() , join (long), join (long, int), sleep (long) o sleep (long, int), métodos de esta clase, luego se borrará su estado de interrupción y recibirá una InterruptedException, que le da la oportunidad a otro hilo para ejecutar el método run() correspondiente de otro hilo, lo que da como resultado un alto rendimiento y reduce el tiempo de espera de los hilos. Diferentes escenarios donde podemos interrumpir un hilo.

1. Interrumpir un hilo que no deja de funcionar: En el programa, manejamos la InterruptedException usando el bloque try y catch, por lo que cada vez que un hilo interrumpe el hilo que se está ejecutando, saldrá del estado de suspensión, pero no dejará de funcionar.

```
// Java Program to illustrate the
// concept of interrupt() method
// while a thread does not stops working
class MyClass extends Thread {
    public void run()
    {
        try {
            for (int i = 0; i < 5; i++) {
                System.out.println("Child Thread
executing");

                // Here current threads goes to
sleeping state
                // Another thread gets the chance to
execute
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e) {
            System.out.println("InterruptedException
occur");
        }
    }
}

class Test {
    public static void main(String[] args)
              throws InterruptedException
    {
        MyClass thread = new MyClass();
        thread.start();

        // main thread calls interrupt() method on
        // child thread
        thread.interrupt();

        System.out.println("Main thread execution
completes");
    }
}
```

2. Interrumpir un hilo que deja de funcionar: en el programa, después de interrumpir el hilo que se está ejecutando actualmente, estamos lanzando una nueva excepción en el bloque catch para que deje de funcionar.

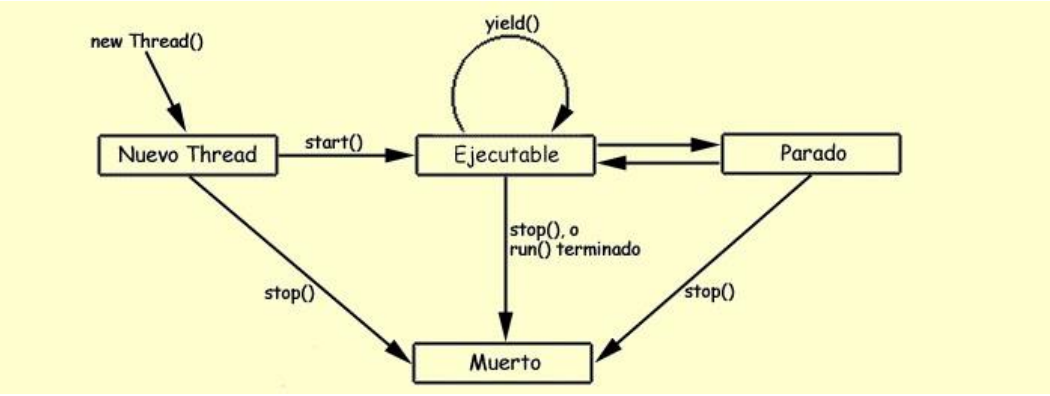
```
// Java Program to illustrate the
// concept of interrupt() method
// while a thread stops working
class Geeks extends Thread {
    public void run()
    {
        try {
            Thread.sleep(2000);
            System.out.println("Geeksforgeeks");
        }
        catch (InterruptedException e) {
            throw new RuntimeException("Thread " +
                                       "interrupted");
        }
    }
    public static void main(String args[])
    {
        Geeks t1 = new Geeks();
        t1.start();
        try {
            t1.interrupt();
        }
        catch (Exception e) {
            System.out.println("Exception handled");
        }
    }
}
```

3. Interrumpir un hilo que funciona normalmente: En el programa, no se produjo ninguna excepción durante la ejecución del hilo. Aquí, la interrupción solo establece el indicador de interrupción en verdadero, que puede ser utilizado por el programador de Java más adelante.

```
// Java Program to illustrate the concept of
// interrupt() method
class Geeks extends Thread {
    public void run()
    {
        for (int i = 0; i < 5; i++)
            System.out.println(i);
    }
    public static void main(String args[])
    {
        Geeks t1 = new Geeks();
        t1.start();
        t1.interrupt();
    }
}
```

TEMA 4: Estados de los Hilos

Durante el ciclo de vida de un hilo, éste se puede encontrar en diferentes estados. La figura siguiente muestra estos estados y los métodos que provocan el paso de un estado a otro. Este diagrama no es una máquina de estados finita, pero es lo que más se aproxima al funcionamiento real de un hilo.



Nuevo Thread

La siguiente sentencia crea un nuevo hilo de ejecución pero no lo arranca, lo deja en el estado deNuevo Thread:

```
Thread MiThread = new MiClaseThread();
Thread MiThread = new Thread( new UnaClaseThread,"hiloA" );
```

Cuando un hilo está en este estado, es simplemente un objeto Thread vacío. El sistema no ha destinado ningún recurso para él. Desde este estado solamente puede arrancarse llamando al método start(), o detenerse definitivamente, llamando al método stop(); la llamada a cualquier otro método carece de sentido y lo único que provocará será la generación de una excepción de tipoIllegalThreadStateException.

Ejecutable

Ahora obsérvense las dos líneas de código que se presentan a continuación:

```
Thread MiThread = new MiClaseThread();
MiThread.start();
```

La llamada al método start() creará los recursos del sistema necesarios para que el hilo puede ejecutarse, lo incorpora a la lista de procesos disponibles para ejecución del sistema y llama al método run() del hilo de ejecución. En este momento se encuentra en el estado Ejecutable del diagrama. Y este estado es Ejecutable y no En Ejecución, porque cuando el hilo está aquí no está corriendo. Muchos ordenadores tienen solamente un procesador lo que hace imposible que todos los hilos estén corriendo al mismo tiempo. Java implementa un tipo de scheduling o lista de procesos, que permite que el procesador sea compartido entre todos los procesos o hilos que se encuentran en la lista. Sin embargo, para el propósito que aquí se persigue, y en la mayoría de los casos, se puede considerar que este estado es realmente un estado En Ejecución, porque la impresión que produce ante el usuario es que todos los procesos se ejecutan al mismo tiempo.

Cuando el hilo se encuentra en este estado, todas las instrucciones de código que se encuentren dentro del bloque declarado para el método run(), se ejecutarán secuencialmente.

Parado

El hilo de ejecución entra en estado Parado cuando alguien llama al método suspend(), cuando se llama al método sleep(), cuando el hilo está bloqueado en un proceso de entrada/salida o cuando el hilo utiliza su método wait() para esperar a que se cumpla una determinada condición. Cuando ocurra cualquiera de las cuatro cosas anteriores, el hilo estará Parado.

Por ejemplo, en el trozo de código siguiente:

```
Thread MiThread = new MiClaseThread()
MiThread.start();
try {
    MiThread.sleep( 10000 );
} catch( InterruptedException e ) {
    ;
    }
```

la línea de código que llama al método sleep():

```
MiThread.sleep( 10000 );
```

hace que el hilo se duerma durante 10 segundos. Durante ese tiempo, incluso aunque el procesador estuviese totalmente libre, MiThread no correría. Después de esos 10 segundos. MiThread volvería a estar en estado Ejecutable y ahora sí que el procesador podría hacerle caso cuando se encuentre disponible.

Para cada una de los cuatro modos de entrada en estado Parado, hay una forma específica de volver a estado Ejecutable. Cada forma de recuperar ese estado es exclusiva; por ejemplo, si el hilo ha sido puesto a dormir, una vez transcurridos los milisegundos que se especifiquen, él solo se despierta y vuelve a estar en estado Ejecutable. Llamar al método resume() mientras esté el hilo durmiendo no serviría para nada.

Los métodos de recuperación del estado Ejecutable, en función de la forma de llegar al estadoParado del hilo, son los siguientes:

1. Si un hilo está dormido, pasado el lapso de tiempo
2. Si un hilo de ejecución está suspendido, después de una llamada a su método resume()
3. Si un hilo está bloqueado en una entrada/salida, una vez que el comando de entrada/salida concluya su ejecución
4. Si un hilo está esperando por una condición, cada vez que la variable que controla esa condición varíe debe llamarse al método notify() o notifyAll()

Muerto

Un hilo de ejecución se puede morir de dos formas: por causas naturales o porque lo maten (constop()). Un hilo muere normalmente cuando concluye de forma habitual su método run(). Por ejemplo, en el siguiente trozo de código, el bucle while es un bucle finito -realiza la iteración 20 veces y termina:

```
public void run() {
    int i=0;
    while( i < 20 ) {
        i++;
        System.out.println( "i = "+i );
        }
    }
```

Un hilo que contenga a este método run(), morirá naturalmente después de que se complete el bucle y run() concluya.

También se puede matar en cualquier momento un hilo, invocando a su método stop(). En el trozo de código siguiente:

```
Thread MiThread = new MiClaseThread();
MiThread.start();
try {
    MiThread.sleep( 10000 );
} catch( InterruptedException e ) {
    ;
    }
MiThread.stop();
```

se crea y arranca el hilo MiThread, se duerme durante 10 segundos y en el momento de despertarse, la llamada a su método stop(), lo mata.

El método stop() envía un objeto ThreadDeath al hilo de ejecución que quiere detener. Así, cuando un hilo es parado de este modo, muere asíncronamente. El hilo morirá en el momento en que reciba ese objeto ThreadDeath.

Los applets utilizarán el método stop() para matar a todos sus hilos cuando el navegador con soporte Java en el que se están ejecutando le indica al applet que se detengan, por ejemplo, cuando se minimiza la ventana del navegador o cuando se cambia de página.

TEMA 5: Sincronización de Hilos I

Los métodos sincronizados se utilizan para conseguir un acceso controlado a los objetos. El lenguaje de programación Java proporciona diversos modos principales de sincronización. La primera son los métodos sincronizados. En la implementación de una clase, pueden especificarse algunos (o todos) los métodos como sincronizados (synchronized).

Tipos o métodos de Sincronizacion:

1. Método synchronized
2. Método join
3. Monitores

En el método se debe colocar la palabra reservada synchronized.

Sintaxis de un método sincronizado:

```
public synchronized void nombre_de_metodo() {

        " Instrucciones "

    }
```

Cuando una hebra realiza una llamada a un método sincronizado, antes de que se comience a ejecutar el código del método, la hebra debe conseguir bloquear el cerrojo asociado con el objeto this que se está utilizando.

Ejemplo:

En este ejemplo se ejecuta como debe ser el contenido del método run, pero la aplicación detecta un método synchronized y lo ejecuta hasta culminar dicho método, para continuar con el método run.

```
public class numeros extends Thread {
public int a;
private String identidad;
public void nombre(String identidad){
this.identidad = identidad;
}
public void run() {
for ( a=1;a<11;a++){
System.out.println(a + " ");
try {
sleep(1000);
} catch(InterruptedException ex) {
}
}
}
public synchronized void impares() {
for ( a=1;a<10;a++){
System.out.print(identidad + " " +a );
a++;}
}
public static void main(String []args){
Thread num1 = new numeros();
numeros num2 = new numeros();
num2.nombre(" Impar --> ");
num1.start();
System.out.println("\n");
try{
num2.impares();
System.out.println("\n");
num2.sleep(5000);
}catch(Exception ex){
}
}
}
```

Resultado

```
1
 Impar -->  1 Impar -->  3 Impar -->  5 Impar -->  7 Impar -->  9

2
3
4
5
6
7
8
9
10
BUILD SUCCESSFUL (total time: 11 seconds)
```

TEMA 6: Sincronización de Hilos II

La Sincronización se usa para crear thread-safe, para asegurar que las variables críticas no perderán modificaciones ni darán mal la información.Todos los métodos desde los que se pueden modificar esas variables críticas deberían de ser Synchronized.

Synchronized usa bloqueos para sincronizar. Se bloquean partes de código usando locks de OBJETO o de CLASE. Un lock de Objeto es un valor único que tiene cada Objeto y un lock de Clase es un valor único que tiene cada Clase. Los bloqueos sólo funcionan cuando varios hilos usan el mismo lock. Por lo que si dos hilos quieren entrar en un método Synchronized que usa un lock de Objeto y están usando locks de Objetos distintos, no se bloquearán. Ejemplo de uso: si vamos a sacar dinero de la misma cuenta del banco, debería bloquear (1º uno y luego el otro). Si yo voy a sacar de la mia y tú de la tuya, no debería bloquear.

```java
// si usamos synchronized ("x") bloqueará siempre!

// ya que "x" está en el String Pool del Heap y tiene un único lock.

// Sería diferente si hicieramos new String("x").

// En ese caso crearía un objeto cada vez, y cada uno tendría su lock

class A implements Runnable {
    @Override
    public void run() {
        synchronized ("x") {
            try {
                Thread.sleep(1000);
                System.out.println(Thread.currentThread().getName());
            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }
    }
}

public class Uno {
    public static void main(String[] args) {
        A a1 = new A();
        A a2 = new A();
        Thread t1 = new Thread(a1, "t1");
        Thread t2 = new Thread(a2, "t2");
        t1.start();
        t2.start();
    }
}
```

Al OBJETO o CLASE que se USA para SINCRONIZAR un método o un bloque de código se le suele LLAMAR también MONITOR.

- Si no ponemos un método crítico a Synchronized podrán acceder varios hilos a la vez, preguntar, dormir, actuar sobre algo que ya no es lo que era.
- Sólo los métodos (no static o static) y los bloques de código (partes/trozos de un método no static o static) pueden ser Synchronized. Ni clases, ni variables podrán ser Synchronized.
- Cuando un THREAD obtiene un LOCK de un MONITOR (OBJETO O CLASE) todos los métodos (ya sean static, no static, de bloque o de método) que usen ese monitor quedarán BLOQUEADOS para el resto de threads y hasta que ese hilo no libere el lock del monitor NINGÚN otro HILO que PREGUNTE por ESE MISMO LOCK PODRÁ ENTRAR en ninguno de ellos.

- Un Thread puede adquirir más de un lock de distintos monitores.

- Si un Thread tiene el lock de un monitor (objeto o clase), puede entrar en más de uno de los métodos Synchronized de ese monitor.
- Si un Thread intenta adquirir un lock de un monitor que está bloqueado, pasará al -Pool de Threads Waiting que esperan a que se libere- de ese monitor y una vez se libere el lock cualquiera de los hilos del -Pool de Threads Waiting que siguen esperando- podrá coger el lock y con ello bloquear el método/bloque a los demás, que tendrán que esperar su oportunidad.
- Métodos de Object para la sincronización:

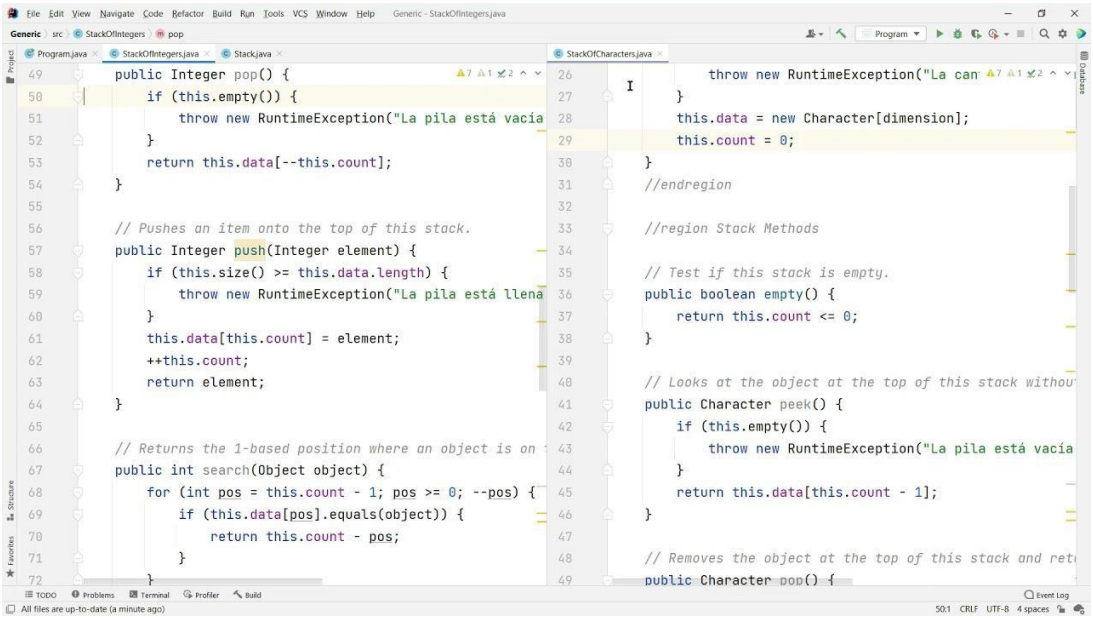
```
1
2    * Estos tres métodos deben ejecutarse siempre dentro de un método/bl
3      sincronizado y los llamará el monitor (objeto o clase usado para s
4
5   void notify()
6   // LIBERA UN Thread cualquiera del
7   // "Pool de Threads Waiting"
8   // y lo manda al RUNNABLE-POOL.
9
10  void notifyAll()
11  // LIBERA TODOS los Threads del
12  // "Pool de Threads Waiting"
13  // y los manda al RUNNABLE-POOL.
14
15  void wait()
16  void wait(long timeout)
17  void wait(long timeout, int nanos)
18  // Lo llama el monitor (Objeto o clase usado para sincronizar)
19  // Wait añade el thread que estaba RUNNING,
20  // el que ha llamado a wait(),
21  // al "Pool de Threads Waiting" de ese monitor.
22  // Ahí esperaran a notify o a notifyAll para volver
23  // al RUNNABLE-POOL.
24  // Exige try catch para controlar la Exception.InterruptedException!
```

UNIDAD 3: Java Generics

TEMA 1: Programación genérica

La programación genérica tiene como objetivo facilitar el desarrollo de algoritmos (o métodos en la POO) de manera tal que el tipo de datos específico manipulado por el algoritmo sea especificado al momento de ejecutar el algoritmo y no al momento de desarrollar el algoritmo. Por ejemplo, se puede desarrollar un algoritmo que opera con números (en términos genéricos), y sólo al momento de ejecutar el algoritmo se especifica el tipo concreto de número como Integer, Float o Double.

Antes que la programación genérica sea incorporada como una característica del lenguaje Java, este estilo programación de algoritmos podría realizarse empleando parámetros de tipos abstractos o empleado sobrecarga de métodos, el problema con estos enfoques, sin embargo, radica principalmente en que, en tiempo de compilación, no era posible detectar el uso inapropiado de tipos que se manifestaban luego en excepciones en tiempo de ejecución y en la duplicación innecesaria de código violando el principio DRY. La especificación genérica de tipo se emplea en la práctica totalidad de las Interfaces y clases de la librería de colecciones de Java.



En Java los mecanismos de programación genérica existen solo en tiempo de compilación y la información sobre parámetros y valores de tipos es eliminada y reemplazada por tipos concretos de acuerdo a las siguientes reglas:

1. Un tipo no acotado se reemplaza con Object
2. Un tipo acotado se reemplaza con el tipo en límite superior (el tipo más genérico)
3. Los tipos genéricos en los valores de retorno se reemplazan por conversiones explicitas (cast)

TEMA 2: Creación de clases genéricas propias

Las clases genéricas encapsulan operaciones que no son específicas de un tipo de datos determinado. El uso más común de las clases genéricas es con colecciones como listas vinculadas, tablas hash, pilas, colas y árboles, entre otros. Las operaciones como la adición y eliminación de elementos de la colección se realizan básicamente de la misma manera independientemente del tipo de datos que se almacenan.

Para la mayoría de los escenarios que necesitan clases de colección, el enfoque recomendado es usar las que se proporcionan en la biblioteca de clases de Java. Normalmente, crea clases genéricas empezando con una clase concreta existente, y

cambiando tipos en parámetros de tipo de uno en uno hasta que alcanza el equilibrio óptimo de generalización y facilidad de uso. Al crear sus propias clases genéricas, entre las consideraciones importantes se incluyen las siguientes:

- Los tipos que se van a generalizar en parámetros de tipo.

Como norma, cuantos más tipos pueda parametrizar, más flexible y reutilizable será su código. En cambio, demasiada generalización puede crear código que sea difícil de leer o entender para otros desarrolladores.

- Las restricciones, si existen, que se van a aplicar a los parámetros de tipo.

Una buena norma es aplicar el máximo número de restricciones posible que todavía le permitan tratar los tipos que debe controlar. Por ejemplo, si sabe que su clase genérica está diseñada para usarse solo con tipos de referencia, aplique la restricción de clase. Esto evitará el uso no previsto de su clase con tipos de valor, y le permitirá usar el operador as en T, y comprobar si hay valores NULL.

- Si separar el comportamiento genérico en clases base y subclases.

Como las clases genéricas pueden servir como clases base, las mismas consideraciones de diseño se aplican aquí con clases no genéricas. Vea las reglas sobre cómo heredar de clases base genéricas posteriormente en este tema.

- Si implementar una o más interfaces genéricas.

Por ejemplo, si está diseñando una clase que se usará para crear elementos en una colección basada en genéricos, puede que tenga que implementar una interfaz como IComparable<T> donde T es el tipo de su clase.

Las reglas para los parámetros de tipo y las restricciones tienen varias implicaciones para el comportamiento de clase genérico, especialmente respecto a la herencia y a la accesibilidad de miembros. Antes de continuar, debe entender algunos términos. Para una clase genérica Node<T>, el código de cliente puede hacer referencia a la clase especificando un argumento de tipo, para crear un tipo construido cerrado (Node<int>). De manera alternativa, puede dejar el parámetro de tipo sin especificar, por ejemplo,

cuando especifica una clase base genérica, para crear un tipo construido abierto (Node<T>). Las clases genéricas pueden heredar de determinadas clases base construidas abiertas o construidas cerradas:

```
class BaseNode { }
class BaseNodeGeneric<T> { }

// concrete type
class NodeConcrete<T> : BaseNode { }

//closed constructed type
class NodeClosed<T> : BaseNodeGeneric<int> { }

//open constructed type
class NodeOpen<T> : BaseNodeGeneric<T> { }
```

Las clases no genéricas, en otras palabras, concretas, pueden heredar de clases base construidas cerradas, pero no desde clases construidas abiertas ni desde parámetros de tipo porque no hay ninguna manera en tiempo de ejecución para que el código de cliente proporcione el argumento de tipo necesario para crear instancias de la clase base.

```
//No error
class Node1 : BaseNodeGeneric<int> { }

//Generates an error
//class Node2 : BaseNodeGeneric<T> {}

//Generates an error
//class Node3 : T {}
```

Las clases genéricas que heredan de tipos construidos abiertos deben proporcionar argumentos de tipo para cualquier parámetro de tipo de clase base que no se comparta mediante la clase heredada, como se demuestra en el código siguiente:

```
class BaseNodeMultiple<T, U> { }

//No error
class Node4<T> : BaseNodeMultiple<T, int> { }

//No error
class Node5<T, U> : BaseNodeMultiple<T, U> { }

//Generates an error
//class Node6<T> : BaseNodeMultiple<T, U> {}
```

Las clases genéricas que heredan de tipos construidos abiertos deben especificar restricciones que son un superconjunto de las restricciones del tipo base, o que las implican:

```
class NodeItem<T> where T : System.IComparable<T>, new() { }
class SpecialNodeItem<T> : NodeItem<T> where T : System.IComparable<T>, new() { }
```

Los tipos genéricos pueden usar varios parámetros de tipo y restricciones, de la manera siguiente:

```
class SuperKeyType<K, V, U>
    where U : System.IComparable<U>
    where V : new()
{ }
```

Tipos construidos cerrados y construidos abiertos pueden usarse como parámetros de método:

```
void Swap<T>(List<T> list1, List<T> list2)
{
    //code to swap items
}

void Swap(List<int> list1, List<int> list2)
{
    //code to swap items
}
```

Si una clase genérica implementa una interfaz, todas las instancias de esa clase se pueden convertir en esa interfaz.

Las clases genéricas son invariables. En otras palabras, si un parámetro de entrada especifica un List<BaseClass>, obtendrá un error en tiempo de compilación si intenta proporcionar un List<DerivedClass>.

TEMA 3: Métodos genéricos

os genéricos presentan el concepto de parámetros de tipo en Java, lo que permite diseñar clases y métodos que aplazan la especificación de uno o varios tipos hasta que el código de cliente declara y crea instancias de la clase o método. Por ejemplo, al usar un parámetro de tipo genérico T, puede

escribir una clase única que otro código de cliente puede usar sin incurrir en el costo o riesgo de conversiones en tiempo de ejecución u operaciones de conversión boxing, como se muestra aquí:

```
// type parameter T in angle brackets
public class GenericList<T>
{
    // The nested class is also generic on T.
    private class Node
    {
        // T used in non-generic constructor.
        public Node(T t)
        {
            next = null;
            data = t;
        }

        private Node? next;
        public Node? Next
        {
            get { return next; }
            set { next = value; }
        }

        // T as private member data type.
        private T data;

        // T as return type of property.
        public T Data
        {
            get { return data; }
            set { data = value; }
        }
    }

    private Node? head;

    // constructor
    public GenericList()
    {
        head = null;
    }

    // T as method parameter type:
    public void AddHead(T t)
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node? current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }
}
```

En el ejemplo de código siguiente se muestra cómo el código de cliente usa la clase genérica GenericList<T> para crear una lista de enteros. Simplemente cambiando el argumento de tipo, el código siguiente puede modificarse fácilmente para crear listas de cadenas o cualquier otro tipo personalizado:

```
class TestGenericList
{
    static void Main()
    {
        // int is the type argument
        GenericList<int> list = new GenericList<int>();

        for (int x = 0; x < 10; x++)
        {
            list.AddHead(x);
        }

        foreach (int i in list)
        {
            System.Console.Write(i + " ");
        }
        System.Console.WriteLine("\nDone");
    }
}
```

Introducción a los genéricos

- Use tipos genéricos para maximizar la reutilización del código, la seguridad de tipos y el rendimiento.
- El uso más común de los genéricos es crear clases de colección.
- La biblioteca de clases de .NET contiene varias clases de colección genéricas en el espacio de nombres System.Collections.Generic. Las colecciones genéricas se deben usar siempre que sea posible en lugar de clases como en ArrayList el espacio de System.Collections nombres.
- Puede crear sus propias interfaces, clases, métodos, eventos y delegados genéricos.
- Puede limitar las clases genéricas para habilitar el acceso a métodos en tipos de datos determinados.

- Puede obtener información sobre los tipos que se usan en un tipo de datos genérico en tiempo de ejecución mediante la reflexión.

UNIDAD 4: Persistencia de Objetos

TEMA 1: Construcción objetos I

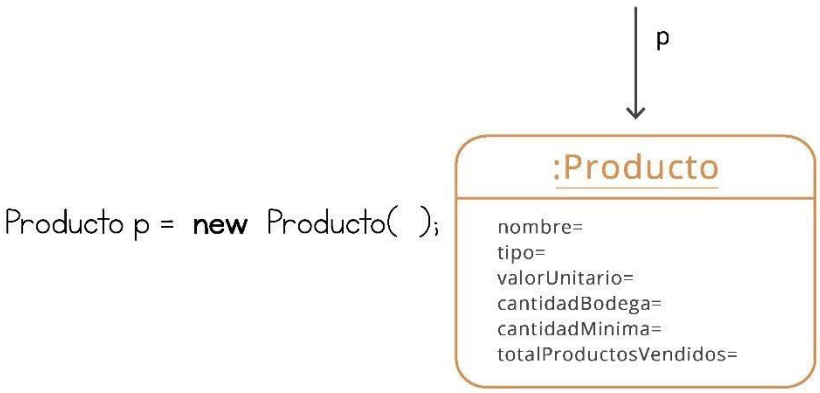
En programación, la persistencia es la acción de preservar la información de un objeto de forma permanente (guardado), pero a su vez también se refiere a poder recuperar la información del mismo (leerlo) para que pueda ser nuevamente utilizado.
De forma sencilla, puede entenderse que los datos tienen una duración efímera; desde el momento en que estos cambian de valor se considera que no hay persistencia de los mismos. Sin embargo, en informática hay varios ámbitos donde se aplica y se entiende la persistencia.

En el caso de persistencia de objetos la información que persiste en la mayoría de los casos son los valores que contienen los atributos en ese momento, no necesariamente la funcionalidad que proveen sus métodos. La persistencia de objetos puede ser fácilmente confundida con la persistencia en memoria; incluso con la persistencia de aplicación.1
La persistencia de objetos consiste en la inicialización de objetos con sus atributos predeterminados o atributos por defecto. Esto es posible con dos maneras de proceder.

1. Sobre un medio (de almacenamiento) fijo se guarda (cuando el objeto fue definido) un conjunto de datos que son recuperados cuando el tipo de objeto en cuestión es creado; dichos datos son transferidos a las propiedades del objeto.
2. Otro objeto mantiene los datos que serán transferidos a las propiedades del nuevo objeto creado. En este caso los datos están en memoria.

TEMA 2: Construcción objetos II

Recordemos la creación de objetos visto en el nivel anterior. Un objeto se crea utilizando la instrucción new y dando el nombre de la clase de la cual va a ser una instancia. Todas las clases tienen un método constructor por defecto, sin necesidad de que el programador tenga que crearlo. Como no es responsabilidad del computador darle un valor inicial a los atributos, cuando se usa este constructor, éstos quedan en un valor que se puede considerar indefinido. En la figura se muestra el resultado del llamado a este constructor.



El resultado de ejecutar la instrucción del ejemplo es un nuevo objeto, con sus atributos no inicializados.

Dicho objeto está "referenciado" por p, que puede ser un atributo o una variable de tipo Producto.

Para inicializar los valores de un objeto, se debe definir en la clase un constructor propio. En el siguiente ejemplo trabajaremos los conceptos vistos en el capítulo anterior, usando el caso de la tienda.

Ejemplo

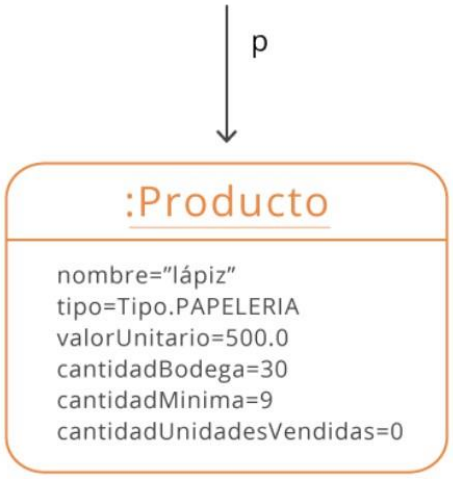
Se hace la inicialización de los atributos de los objetos de la clase. En este ejemplo mostramos los constructores de las clases Tienda y Producto, así como la manera de pedir la creación de un objeto de cualquiera de esos dos tipos.

```
public Producto(Tipo pTipo, String pNombre, double pValorUnitario, int pCantidadBodega, int pCanti
{
    tipo = pTipo;
    nombre = pNombre;
    valorUnitario = pValorUnitario;
    cantidadBodega = pCantidadBodega;
    cantidadMinima = pCantidadMinima;
    cantidadUnidadesVendidas = 0;
}
```

- El constructor exige 5 parámetros para poder inicializar los objetos de la clase Producto.
- En el constructor se asignan los valores de los parámetros a los atributos.

```
Producto p=new Producto(Tipo.PAPELERIA, "lápiz", 500.0, 30, 9);
```

Este es un ejemplo de la manera de crear un objeto cuando el constructor tiene parámetros.



- Este es el objeto que se crea con la llamada anterior.
- El objeto creado se ubica en alguna parte de la memoria del computador.

Dicho objeto es referenciado por el atributo o la variable llamada "p".

```
public Tienda(  )
{
    producto1 = new Producto( Tipo.PAPELERIA, "Lapiz", 550.0, 18, 5 );
    producto2 = new Producto( Tipo.DROGUERIA, "Aspirina", 109.5, 25, 8 );
    producto3 = new Producto( Tipo.PAPELERIA, "Borrador", 207.3, 30, 10 );
    producto4 = new Producto( Tipo.SUPERMERCADO, "Pan", 150.0, 15, 20 );
    dineroEnCaja = 0;
}
```

- Puesto que es necesario que la tienda tenga 4 productos, su método constructor debe ser como el que se presenta. Supone que en la caja de la tienda no hay dinero al comenzar el programa.

Vamos a practicar la creación de escenarios usando los métodos constructores de las clases Tienda y Producto. En el programa del caso de estudio, la responsabilidad de crear el estado de la tienda sobre la cual se trabaja está en la clase principal del mundo (clase Tienda). En una situación real, dichos valores deberían leerse de un archivo o de una base de datos, pero en nuestro caso se utilizará un escenario predefinido. Si quiere

modificar los datos de la tienda sobre los que trabaja el programa, puede darle otros valores en el momento de construir las instancias.

TEMA 3: Construcción objetos III

Paradigma

Un paradigma es una forma de afrontar la construcción de código de software

- No hay paradigmas mejores ni peores
- Todos tienen sus ventajas e inconvenientes

Hay distintos paradigmas:

- POO, Estructurado, funcional, Lógico, etc.

La programación orientada a objetos permite:

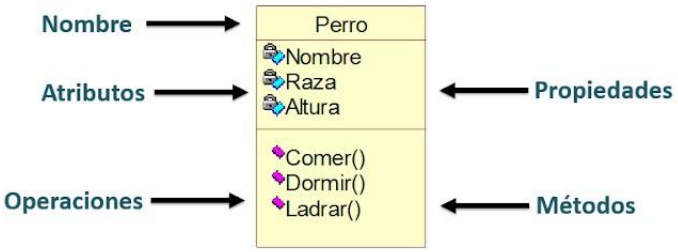
- Facilidad de diseño y relación con el mundo real (UML)
- Reutilización de piezas de código (no copy/paste)
- Encapsulamiento (ocultar el estado de los objetos)

Elementos de la POO

Los elementos principales son:

Clases

- Especificación de un conjunto de elementos ➢ Plantillas para definir elementos (objetos).
- Pueden estar directamente relacionadas unas con otras.



Objetos

- Elemento autónomo y con una funcionalidad concreta

Elementos con comportamiento y estado.

- Métodos y atributos concretos.
- Instancias concretas de una clase.
- Interactúan por medio de mensajes.

Clase

Objeto

Encapsulación

Puede (y suele) haber distintos niveles de visibilidad:

public: se puede acceder desde cualquier lugar. private: sólo se puede acceder desde la propia clase.

protected: sólo se puede acceder desde la propia clase o desde una clase que herede de ella.

Instanciación de Objetos

Antes de utilizar un objeto debemos de crearlo.

```
package Clases;

public class Perro {
    //Propiedades
    public String nombre;
    public String raza;
    public String altura;

    //Métodos
    public String comer(String carne)
    {
        return nombre + " mide " + altura + " y comerá " + carne;
    }

    public void dormir()
    {

    }
    public void ladrar()
    {

    }
}
```

Métodos

Definen el comportamiento de los objetos de una clase.

1. Métodos Habituales *Constructor*:

- Sirve para inicializar un objeto al crearlo.
- Existe sobrecarga (distintos parámetros) (para cualquier método).
- Coincide con el nombre de la clase y no devuelve nada por definición.

```
public Perro( String nombre,String raza, String altura)
    {
        this.nombre = nombre;
        this.raza = raza;
        this.altura = altura;
    }
```

2. Get y Set

```
public String getNombres() {
    return nombres;
}
public void setNombres(String nombres) {
    this.nombres = nombres;
}
```

3. Destructor

No es tan típico, no se suele usar (se hace de manera automática).

B. Base de Consulta

TÍTULO	AUTOR	EDICIÓN	AÑO	IDIOMA	EDITORIAL
Curso de Programación	Ceballos, F	1	2004	Español	Alfaomega, Cuarta Edición.
Construcción de Software Orientado a Objetos	Meye,B	1	1998	Español	Prentice Hall, Segunda Edición.
Como programar en JAVA	Deitel, H. Deitel, P	1	2004	Español	Pearson educación
Lenguaje unificado de modelado	Booch, G. Rumbaugh, J. Jacobson, I. Saez Martinez, J. Garcia, M. Jesus J.Guillermo	1	2003	Español	Addison Wesley.

C. Base práctica con ilustraciones

TRABAJOS PARA PRESENTAR	DETALLE
	3- Presentación del Proyecto Impreso Debe contener carátula, encabezado y pie de página, Índice, inicio a normas APA

4. ESTRATEGIAS DE APRENDIZAJE

ESTRATEGIA DE APRENDIZAJE 1: Análisis y Planeación
Descripción: La complejidad de los programas que se desarrollan actualmente produce la necesidad de iniciar a los alumnos en un camino que los conduzca a utilizar efectivas técnicas de programación. Es importante para ello poner énfasis en el diseño previo. Como se ha comprobado, una estrategia valedera es comenzar a enseñar programación utilizando los algoritmos como recursos esquemáticos para plasmar el modelo de la resolución de un problema. Esto genera una primera etapa de la programación que resulta un tanto tediosa para los alumnos que están ávidos de utilizar la computadora. Si bien no aparecen

dificultades graves con el aprendizaje de esta técnica. El hecho de reescribir los algoritmos hasta ponerlos a punto es operativamente complicado cuando se trabaja con lápiz y papel. Además, comprobar la corrección del algoritmo presenta inconvenientes importantes. Es difícil, mental o gráficamente, representar las acciones del programa en ejecución de manera totalmente objetiva, sin dejarse llevar por la subjetividad, fundamentalmente cuando el que lo hace es el propio autor del algoritmo. Por otra parte, se ha comprobado que el uso del método global para el aprendizaje de la asignatura de programación orientada a objetos, ahorra tiempo y esfuerzo. Con el propósito de trabajar especialmente sobre los aspectos mencionados se creó un Ambiente de Aprendizaje con un editor interactivo de algoritmos, un constructor automático de trazas y un traductor. Se presentan en este trabajo, los resultados obtenidos en una experiencia de campo diseñada para comprobar la efectividad de la aplicación del entorno de programación mencionado.

Se pretende adquirir conocimiento, a través de la práctica participativa en temas relevantes como: Herencias, Clases, Polimorfismo, Encapsulación y Examen Final. Los ejercicios en clase guiados por el maestro ayudan al estudiante a comprender el lenguaje de mejor manera y genera a su vez la necesidad de poder crear nuevo conocimiento, investigar nuevos conceptos complementarios al buen desempeño del trabajo que se está realizando. Se persigue que el estudiante debe mostrar interés en el desarrollo del curso, asumir un compromiso en las pautas metodológicas, participar activamente en modo personal y grupo.

Finalmente, el estudiante se identifica con la carrera al plantear sus propias aplicaciones para su solución. Participa activamente con responsabilidad y respeto.

Ambiente(s) requerido:

Aula amplia con buena iluminación.

Material (es) requerido: ✓ Infocus. ✓ Laboratorios de computación ✓ Parlante
Docente: Con conocimiento de la materia.

5. ACTIVIDADES

- Controles de lectura
- Exposiciones
- Presentación del Trabajo final

6. EVIDENCIAS Y EVALUACIÓN

Tipo de Evidencia	Descripción (de la evidencia)
De conocimiento:	Creación de un sistema informático aplicando una metodología de enseñanza proporcionada basada en normas APA, que permitirá al estudiante comprender como comercializar y promocionar un producto apoyado de las TIC's.
Desempeño:	Trabajo individual presentación del trabajo sobre la creación de un sistema de control para usuarios finales. Exposición individual del proyecto educacional
De Producto:	✓ Trabajo de realizado consiste en la presentación de un programa completo desarrollado en un lenguaje de programación orientado a objetos. ✓ Defensa del proyecto mediante una práctica de los estudiantes ✓ Exposiciones orales sobre los temas de investigación individuales asignados a los estudiantes, sobre hojas de cálculo

Criterios de Evaluación (Mínimo 5 actividades por asignatura)	✓ Identificar la lógica matemática aplicada en un sistema informático.

Compilado por: **Ing. Byron Giovanny Cholca MSc.**	**Revisado Por: (Dirección Investigación)**	**Aprobado Por: (Rectorado)**