Ingeniería en Desarrollo de Software
3er semestre

Programa de la asignatura:
Programación orientada a objetos I

Unidad 3. Características de POO y excepciones

Clave:
Ingeniería: **TSU:**
15142316 / 16142316

Universidad Abierta y a Distancia de México

Índice

Unidad 3. Características de POO y excepciones

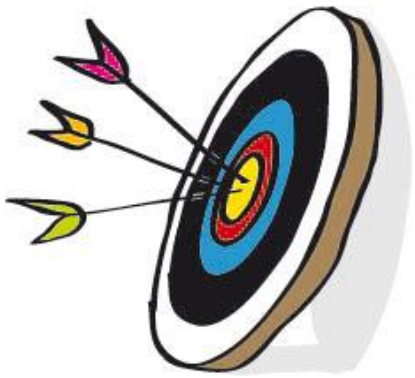
Presentación de la unidad

En esta tercera unidad de Programación Orientada a Objetos I (POO I) aprenderás los conceptos de **herencia y polimorfismo** que son muy utilizados en la programación, ya que, por ejemplo, cada que se utiliza una librería o se invocan paquetes predefinidos por JAVA se hace uso de estas características de la orientación a objetos. Dichas características te serán especialmente útiles cuando manejes aplicaciones con componentes gráficos (lo cual estudiarás durante la asignatura POO II).

También, durante el desarrollo de esta unidad se verá el tema de **excepciones,** es importante para evitar que al usuario final “le truene el sistema”, es decir, que tenga un cierre inesperado y se controlen mediante programación los posibles errores en que pueda incurrir el sistema.

Para comprender mejor los temas es muy importante que captures todos los programas de ejemplo y analices su sintaxis.

Propósito



Al término de esta unidad lograrás:

- Identificar el uso de la herencia para reutilizar código y construir clases basadas en algunas creadas con anterioridad.
- Determinar el uso del polimorfismo para implementar un mismo método en distintas clases de diferentes maneras.
- Identificar el uso de las excepciones en la programación, lo cual permitirá controlar posibles errores de ejecución.

Competencia específica

Desarrollar programas modulares para solucionar problemas diversos, aplicando la herencia, el polimorfismo y el manejo de excepciones mediante la sintaxis que JAVA ofrece para su implementación.

3.1. Herencia

La **herencia** es una propiedad del lenguaje que nos permite definir objetos con base en otros ya existentes, y así poder añadirles nuevas características (extenderlos). La nueva clase es llamada subclase o clase extendida, y la clase que hereda sus métodos y atributos se llama superclase.

Esta característica del lenguaje es muy importante, ya que permite la reutilización de código preexistente, de manera que si se tiene la certeza de la funcionalidad de código creado con anterioridad, y se requieren funcionalidades similares, mediante el uso de la herencia (realizando una extensión del código preexistente) el programador se ahorra el tiempo de crear; así como el de probar que el código funcione correctamente.

Título: Vestidos iguales
Tomada de: http://galeria.dibujos.net/fiestas/dia-del-padre/vestidos-iguales-pintado-por-padre-e-hijo-8843343.html

La herencia en programación permite reutilizar código creado con anterioridad, al modificar sólo de manera mínima las clases ya existentes, para que una nueva clase cumpla especificaciones diferentes. Como ya se ha mencionado de manera general, esta característica funciona de manera similar a la herencia genética (de ahí se tomó el concepto): así como los hijos se parecen a sus padres, las subclases se parecen a las superclases. Y así como los hijos pueden, además de parecerse, tener otras características que los hagan diferentes de sus padres, a las subclases es posible añadirles características que las diferencien de la superclase.

Observa la siguiente imagen, en ella se puede apreciar que el hijo se parece al padre, pero no son exactamente iguales, pues el hijo además de las características que tomó de su padre tiene otras que son propias.

En los siguientes subtemas abordarás con mayor profundidad el tema de la herencia.

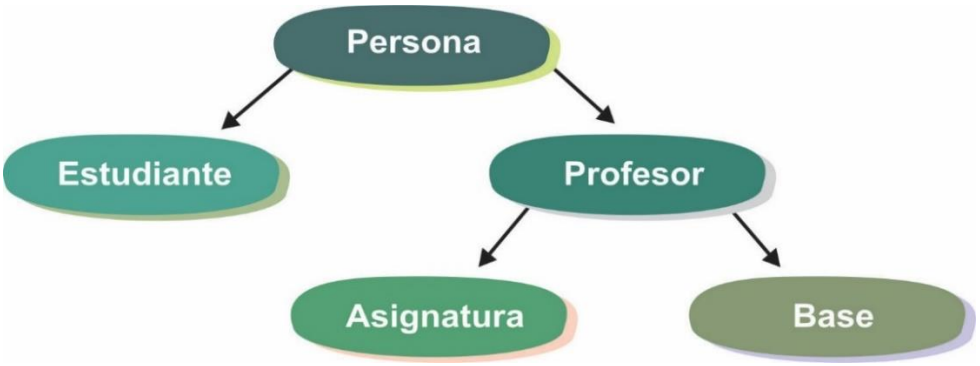
3.1.1. Subclases y superclases

En el lenguaje Java existen dos categorías en las que pueden agruparse las clases: **superclase**, que es la clase ya existente, y **subclase**, que es la clase derivada de la primera; es decir, la clase de nueva creación que está basada en la que ya existía. Otro término usado comúnmente para la superclase es **clase padre** y a la subclase también se le conoce como **clase hija**.

Para entender mejor la clase derivada, o hija, se dirá que es aquella que añade variables de instancia y métodos a partir de los heredados (obtenidos) desde la clase padre.

En la siguiente imagen se ejemplifica el modo en que se heredan los métodos y atributos entre las clases. La clase padre es Persona y sus subclases son Estudiante y Profesor, debe notarse de igual manera que la clase Profesor a su vez funge con dos roles distintos: es subclase de la superclase Persona y es superclase de las subclases Asignatura y Base.

Esto significa que tanto el estudiante como el profesor son personas, por lo tanto todos los atributos y acciones que tiene Persona también los tienen Estudiante y Profesor. Pero además, éstos tienen sus propias características que los hacen diferentes, por lo que Estudiante y Profesor pueden tener otros atributos y métodos además de los que ya hayan sido definidos en Persona.



Ejemplo de herencia

A continuación se muestra el código del ejemplo de herencia revisado; en sombreado azul se encuentra la clase Persona, en morado la clase Estudiante y en verde la clase Main, desde donde se llamarán a Persona y a Estudiante.

```
package ejemplo;
/**
 * @author ESAD
 */
publicclass Persona {
intedad;
    float peso;
    String nombre;
publicvoidcolocarEdad(int e){
        edad=e;
}
    public void colocarPeso(float
p){
peso=p;
    }
publicvoidcolocarNombre(String n){
        nombre=n;
    }
publicintobtenerEdad(){
return edad;
    }
```

```
publicfloatobtenerPeso(){
return peso;
    }
publicStringobtenerNombre(){
return nombre;
    }
}
```

La clase Persona cuenta con tres atributos (edad, peso, nombre) y con seis métodos para colocar y obtener los valores de los atributos.

Observa que los tres atributos se encuentran a nivel de clase, lo que significa que podrán ser utilizados en toda ella.

```
package ejemplo;
/**
 * @author ESAD
 */
publicclass Estudiante extends
Persona{
int matricula;
String carrera;
publicvoidcolocarMatricula(int m){
        matricula=m;
    }
publicvoidcolocarCarrera(String c){
        carrera=c;
    }
publicintobtenerMatricula(){
return matricula;
    }
    public String obtenerCarrera(){
        return carrera;
}
    }
```

La clase Estudiante cuenta con dos atributos (matrícula y carrera) y con cuatro métodos para colocar y obtener los valores de los atributos. En esta clase también se declararon los atributos a nivel de la clase.

Observa que se colocó la palabra reservada **extends** en la declaración de la clase, seguida de Persona, lo que significa que esta clase está siendo heredada o extendida de la clase Persona.

Gracias a esta herencia, los objetos de Estudiante contarán, además de los atributos y métodos declarados en esta clase, con los atributos y métodos de Persona, por lo que cuenta con cinco atributos y diez métodos.

```
package ejemplo;

/**
 * @author ESAD
 */

public class Main {

    public static void main(String[] args){
//primero se crean los objetos de Persona y Estudiante.
        Persona per=new Persona();
        Estudiante est=new Estudiante();

        //Colocar  datos dentro del objeto persona
per.colocarEdad(20);
per.colocarNombre("Juan Perez");
per.colocarPeso(80);
        //Se obtienen los datos de la persona para imprimirlos
System.out.println("Nombre:"+per.obtenerNombre()+"
Edad:"+per.obtenerEdad() +" Peso:"+per.obtenerPeso());

        //Se colocan datos dentro del objeto estudiante
```

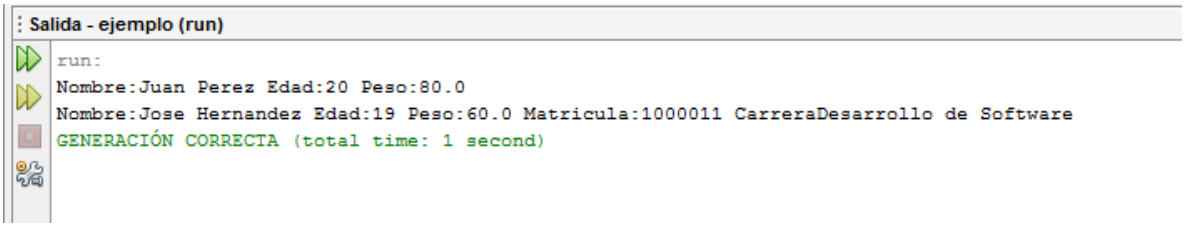
```
est.colocarEdad(19);
est.colocarNombre("JoseHernandez");
est.colocarPeso(60);
est.colocarMatricula(1000011);
est.colocarCarrera("Desarrollo de Software");
System.out.println("Nombre:"+est.obtenerNombre()+"
Edad:"+est.obtenerEdad() +" Peso:"+est.obtenerPeso()+"
Matricula:"+est.obtenerMatricula()+"
Carrera"+est.obtenerCarrera());
    }
}
```

Ahora observa la clase Main, donde se están llamando las otras dos clases, y observa que el objeto de la clase Estudiante asigna valores a sus cinco atributos y de la misma manera obtiene los datos y los imprime.

*Nota 1: recuerda que los valores numéricos se envían a los métodos tal cual, y en el caso de las cadenas (*String*) se envían entre comillas”” indicando precisamente que son una cadena.

*Nota 2: recuerda que en la impresión se tienen “cadenas”+ valores, y esto es llamado concatenación, es decir, se está construyendo una cadena más grande que será la que se imprima.

Se muestra a continuación la imagen de salida de la ejecución del código presentado.



Para contar con un ejemplo más sobre la herencia de la superclase hacia la subclase, imagina que se cuenta con la clase Circulo, que es perfecta para hacer algunas operaciones

y abstracciones matemáticas, que servirán para más de un propósito y más de una aplicación donde se utilice un círculo y las operaciones básicas inherentes a él. Para algunas otras aplicaciones tal vez se precise manipular la clase Circulo (con sus métodos y atributos básicos) de otra manera y se necesite imprimirlo a pantalla, en ese caso, para no modificar por completo la clase Circulo, que ya es usada por cientos de aplicativos más que resultarían seriamente perjudicados, se debe crear una subclase de la clase padre Circulo, se llamaría Dibujacirculo y tendría la misma funcionalidad que su clase padre, más un método añadido: imprimeCirculo.

Una manera de hacerlo es como sigue:

```
public class Dibujacirculo {
// esta es la abstracción del circulo.
public Circle c;
// Métodos anteriores.
public double area( ) { return c.area( ); }
public double circunference( ) { return c.circunference( ); }
// Las nuevas variables y métodos irán aquí.
public Color outline, fill;
public void imprimeCirculo(DrawWindow dw) { /* código obviado */ }
}
```

La solución descrita funciona; sin embargo, no es una solución óptima, y siendo un poco más exigentes puede decirse que carece de elegancia. El problema que se presenta en esta solución es el doble trabajo que involucra escribir los métodos presentados y que no tienen injerencia en el dibujado del círculo.

Otra forma de resolver el problema es heredando (extendiendo) desde la clase Circulo a la subclase Dibujacirculo, como ya se había explicado:

```
public class Dibujacirculo extends circulo {
// Se heredan automáticamente los métodos y atributos de
// circulo, por lo que no es necesario volver a reescribirlos aquí.
Color outline, fill;
public void imprimeCirculo(DrawWindow dw) {
dw.drawCircle(x, y, r, outline, fill);
}
}
```

Deberás notar que se obvia la declaración de dos clases: Color y *DrawWindow,* se toma sólo como ejemplo ilustrativo. La palabra reservada *extends* le indica al compilador de Java que Dibujacirculo es una subclase de circulo, lo que significa que ésta heredará por completo sus métodos y atributos con los modificadores de acceso que se le indicaron en la clase padre.

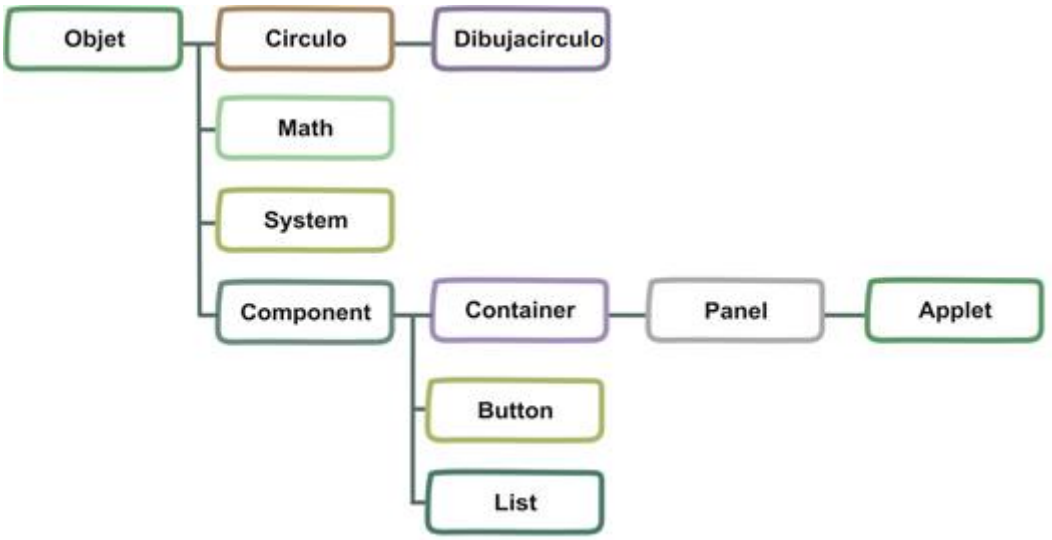
3.1.2. Jerarquía de la herencia

En el lenguaje Java cada clase que se escribe es por definición una superclase. Si no se escribe específicamente la palabra reservada *extends* (que es la que indica la herencia), la superclase será la clase *Object,* que por varias razones es un tipo especial de clase, las cuales se enlistan a continuación:

1. Desde la definición de Java, es la única clase que no tiene superclases.
2. Los métodos definidos en la clase *Object* pueden ser invocados desde cualquier clase Java.

Por lo descrito anteriormente, todas las clases en Java pueden ser superclases.

Se deberá marcar una jerarquía de clases en Java, donde su raíz deberá ser la clase *Object,* se presenta una imagen de su posible representación retomando el ejemplo antes citado de la clase Circulo.



Ejemplo de la jerarquía de la herencia en la clase circulo

3.1.3. Clases y métodos abstractos

Con la salida a la luz del paradigma de programación orientada a objetos surgieron muchos problemas inherentes a la comprensión sobre la forma de representar cosas, y más aún conceptos en forma de un objeto en lenguaje de programación Java.

Al respecto, en muchas ocasiones se desea representar conceptos abstractos en forma de una clase que pueda ser manejada por Java, pero por la misma definición de su abstracción no se puede instanciar a partir de ellas un nuevo objeto.

Considera por ejemplo la comida. ¿Se puede tener una instancia (un ejemplar) de comida? No, aunque sí se pueden tener instancias de manzanas, chocolates o algún otro tipo de comida. El termino comida representa el concepto abstracto de cosas que se pueden comer, lo que nos indica que no es un objeto concreto, sólo una abstracción (conceptualización) de un concepto más amplio (Joyanes, 2001).

Bajo la definición anterior no se puede comprender la utilidad de tener una clase abstracta de manera pronta, sin embargo su utilidad se ve definida cuando sirven para heredar a otras subclases características generales del concepto abstracto que representan, pero de ellas no se puede instanciar objetos con el operador *new*. Las clases a partir de las cuales se pueden crear instancias se denominan **clases concretas** (Niemeyer y Peck, 1997).

Una **clase abstracta** es una clase que contiene los nombres de los comportamientos sin las implementaciones que ejecutan esos comportamientos. Los objetos no se pueden instanciar de una clase abstracta.

Cuando se hace la definición (declaración) de una clase abstracta, deberá ser antecedida por el modificador *abstract.* Este tipo de clases puede contener métodos abstractos, que son sólo la declaración de los mismos, pero sin la implementación de las instrucciones que se esperaría para cumplir el propósito para los cuales fueron escritos. La implementación de dichos métodos se hace en las subclases. Veamos el siguiente ejemplo:

```
package ejemplo;
/**
 * @author ESAD
 */

public abstract class AbstractaAreas {

    public abstract void areaCirculo(int
d);

    public abstract void areaCuadrado(int
l);

    public abstract void
areaRectangulo(int b, int a);
}
```

Observa que, como su definición lo indica, tanto la definición de la clase como de los métodos requiere colocar la palabra reservada **abstract** para indicar que ambos son abstractos. Y sólo se realiza su declaración, sin llegar a realizar la implementación de éstos (entiéndase por implementación el hecho de codificar la clase y los métodos para su ejecución).

Para que un método pueda declararse como abstracto debe cumplir con dos condiciones básicas:

1. No ser método privado.
2. No ser método estático.

Ahora bien, seguramente te preguntarás **¿para qué sirve programar clases y métodos abstractos, si sólo son declaraciones sin funcionalidad?** El principal uso de estos elementos es crear una estructura de un sistema, lo que resulta útil para los analistas y arquitectos de *software*; ya que sin necesidad de programar toda la funcionalidad pueden indicar cómo deberá estructurarse un sistema completo, y entregarlo a los programadores para que realicen su implementación. Esta implementación de clases y métodos abstractos se realiza mediante herencia, por lo tanto debe utilizarse la palabra clave **extends**, como se ve en la siguiente imagen, donde se realiza la implementación del ejemplo anterior.

```
package ejemplo;
/**
 * @author ESAD
 */
publicclassImplementacionextendsAbstractaAreas{

    @Override
publicvoidareaCirculo(int r) {
doubleareaCirculo= Math.PI *(r * r);
System.out.println("El area del circulo es:
"+areaCirculo);
    }

    @Override
publicvoidareaCuadrado(int l) {
```

```
intareaCuadrado= l * l;
System.out.println("El area del cuadrado es:
"+areaCuadrado);

}

    @Override
    public void areaRectangulo(int b, int a) {
intareaRectangulo= b * a;
System.out.println("El area del rectangulo es:
"+areaRectangulo);
    }

}
```

Ejemplo de implementación de clases y métodos abstractos.

Nótese que se utilizó herencia para la implementación de la clase abstracta y sus métodos (se utilizó **implements**), y para indicar esa implementación se añade la marca **@Override** antes de cada método implementado, lo que indica que éste había sido declarado en la clase abstracta y se está reescribiendo.

3.2. Polimorfismo

El polimorfismo consiste en la posibilidad de utilizar una misma expresión para invocar diferentes versiones de un mismo método, ya que permite hacer diferentes implementaciones de los mismos métodos.

Es decir, se pueden realizar diferentes implementaciones de un mismo método, para poder adaptarlo a las necesidades propias de cada clase.

En otras palabras: se pueden crear métodos con el mismo nombre dentro de diferentes clases heredadas, y proporcionales implementaciones diferentes.

El uso de esta propiedad de la programación orientada a objetos es similar a la de la herencia, pues es frecuentemente empleada para la reutilización de código.

3.2.1. Clases y métodos finales

Las **clases** se declaran como **finales** cuando se pretende que de ellas no se pueda derivar ninguna otra clase. Es decir, si se crea una clase y se quiere que no se creen clases hijas de ésta (que hereden), entonces la clase deberá ser declarada como final, para indicar que no se pueden crear otras clases a partir de ella y que terminan con una cadena de herencia.

Esto se declara de la siguiente manera:

```
final class ClaseFinal {
      //aquí va el cuerpo de
la clase
}
```

Como se muestra, sólo es necesario añadir la palabra *final* al inicio de la declaración de la clase.

Por su parte, los **métodos finales** son aquellos que no pueden ser redefinidos, es decir que en ellos no se podrá aplicar ni herencia, ni polimorfismo. Su declaración, al igual que la clase final, sólo consta de anteponer la palabra reservada *final* a la declaración de método en cuestión.

```
final public void metodoFinal(){
      //...aquí va el cuerpo
del método
}
```

Entonces, se usará la palabra *final* en las clases y métodos que ya no se puedan heredar, ni redefinir; es decir, aquellos que no deban ser modificados en ninguna otra clase.

3.2.2. Interfaces

Cuando se define o se pretende utilizar en Java una interface, en realidad se está describiendo un comportamiento. De manera general las interfaces lucen un tanto parecidas a una clase abstracta, sólo las diferencia la forma de declaración:

1. En una clase abstracta se utiliza la palabra reservada *abstract.*
2. En una interface se utiliza la palabra reservada *interface*.

Tratando de entender la definición general de interfaces, se dirá que es un sistema que hace la función de puente para unir entidades no relacionadas entre sí. Cuando se sigue el proceso normal de compilación, el compilador pone cada interface identificada en un archivo de *bytecode* por separado como si fuera una clase normal, además cabe apuntar que en las clases abstractas sólo se tiene la definición de métodos, pero no la implementación de ellos.

"En esencia una interface podría identificarse como lo mismo que una clase abstracta. La gran diferencia entre ellas y lo que justifica la existencia de las interfaces es el hecho de que pueden emular el comportamiento de la herencia múltiple, que no lo soportan las clases abstractas" (Froufe, 2008, p. 101).

Una interface se puede considerar una clase abstracta, de hecho en ella se toma en cuenta lo siguiente (Flanagan, 1997):

1. Todos los miembros son públicos.
2. Todos los métodos son abstractos.
3. Todos los campos son *static* y final.

Una interface consta de dos elementos básicos:

1. La **declaración,** que es donde se colocan los diversos atributos acerca de la interface, como el nombre o si se extiende de otra interface.
2. El **cuerpo,** que contiene la declaración de los métodos y las constantes que conforman la propia interface

Revisa el siguiente ejemplo:

- Se marca en sombreado azul la definición de una interface con 3 métodos.
- Se marca en sombreado morado otra interface con 3 métodos.
- Por último se marca la implementación de la interface en verde. Observa que para hacer esto se utiliza la palabra reservada *implements*, la cual, como se mencionó anteriormente, emula la herencia múltiple, pues como se muestra en el ejemplo, esto permite que se coloque la implementación de varias interfaces en una misma clase, colocando una coma (,) entre el nombre de las interfaces que se están implementado. Y al final la implementación cuenta con seis métodos, tres tomados de la primera interface y tres de la segunda.

```
package ejemplo;

/**
 * @author ESAD
 */

public interface InterfaceAreas {

    public abstract void areaCirculo(int r);

    public abstract void areaCuadrado(int l);

    public abstract void areaRectangulo(int b, int a);

}
```

```
package ejemplo;

/**
 * @author ESAD
 */
public interface InterfacePerimetros {
```

```
publicabstractvoidperimetroCirculo(int r);

publicabstractvoidperimetroCuadrado(int l);

publicabstractvoidperimetroRectangulo(int b, int a);
}
```

```
package ejemplo;

/**
 * @author ESAD
 */
publicclassImplementacionInterfaceimplementsInterfaceAreas,
InterfacePerimetros{

@Override
publicvoidareaCirculo(int r) {
doubleareaCirculo= Math.PI *(r * r);
System.out.println("El area del circulo es: "+areaCirculo);
    }

    @Override
publicvoidareaCuadrado(int l) {
intareaCuadrado= l * l;
System.out.println("El area del cuadrado es: "+areaCuadrado);

}

    @Override
    public void areaRectangulo(int b, int a) {
intareaRectangulo= b * a;
System.out.println("El area del rectangulo es:
"+areaRectangulo);
}
```

```
    @Override
    public void perimetroCirculo(int r) {
throw new UnsupportedOperationException("Not supported
yet.");
}

    @Override
publicvoidperimetroCuadrado(int l) {
throw new UnsupportedOperationException("Not supported
yet.");
}

    @Override
    public void perimetroRectangulo(int b, int a) {
throw new UnsupportedOperationException("Not supported
yet.");
}

}
```

3.2.3. Sobrecarga de métodos

La **sobrecarga de métodos** es un beneficio más que provee el paradigma de la programación orientada a objetos. Esta característica permite declarar varios métodos con el mismo nombre dentro de una sola clase (Martín, 2010, p. 189).

Esto es útil cuando se requieren métodos que realizan prácticamente las mismas acciones, pero de manera diferente. Por ejemplo: imagina una clase llamada imprimir, y que dentro de ella se ha definido un método llamado imprimeResultado, que lo que hace es crear una apariencia de salida para un resultado, no es necesario nombrar un método para cada tipo de datos a imprimir, pues serían muchos nombres y podría llegar a ser

confuso, en lugar de ello se declararían métodos con el mismo nombre, pero diferente recepción de parámetros (se recibirán tantos tipos de datos como se requiera).

Considera el siguiente ejemplo:
Se tiene una clase, donde se han declarado cuatro métodos con el mismo nombre (suma), si todos los métodos tuviesen exactamente la misma declaración esto marcaría un error.
Ahora observa el siguiente ejemplo, donde:

- El primer método recibe dos parámetros enteros.
- El segundo método recibe tres.
- El tercer método recibe dos parámetros de tipo flotante, y
- El cuarto recibe tres flotantes.

La causa de que esto no provoque un error de compilación es que cada uno de los métodos, a pesar de que en apariencia son iguales (todos suman), realmente no lo son, pues manejan diferentes tipos de datos y número de parámetros. El polimorfismo provee al lenguaje de la capacidad de distinguir a que método se está invocando, dependiendo de la llamada que se realice.

Así pues, las llamadas a los métodos en el mismo orden en que están declarados, quedarían como sigue:

- suma(5, 5);
- suma(5, 5, 7);
- suma(5.2, 5.1);
- suma(5.0, 5.0, 6.2);
-

```
package ejemplo;
/**
 * @author ESAD
 */
publicclass Sobrecarga {
public intsuma(int a, int b){
```

```
        return a+b;
    }

    public intsuma(int a, int b, int c){
        return a+b+c;
    }
    public float suma(float a, float b){
        return a+b;
    }
    public float suma(float a, float b, float c){
return a+b+c;
    }
}
```

3.2.4. Sobrecarga de operadores

La sobrecarga de los operadores permite redefinir las funciones que se le asignan por definición a un operador, por ejemplo el operador "+" (más) y el operador "." (punto), se pueden utilizar para realizar otras funciones adicionales a las regulares. Como ejemplo práctico y conocido, el operador "+" en el mundo Java sirve para dos cosas, al estar sobrecargado:

1. Para realizar la **operación de suma**, cuando se acompaña de datos numéricos.
2. Para realizar la **operación de unión**, cuando uno de sus operandos es un dato de tipo *String.*

El compilador decide qué uso se le dará con base en los operandos de los cuales esté acompañado.

La idea de la sobrecarga de operadores también es poder programar cómo debería comportarse un operador, pero en Java esto no es posible, solamente pueden utilizarse los operadores tal como se han definido y nada más.

3.3. Excepciones

Cuando un programa Java viola las restricciones semánticas del lenguaje (se produce un error), la máquina virtual Java comunica este hecho al programa mediante una **excepción**. Por tanto, la manera más simple de explicarlo es que una excepción es un error que ha ocurrido en un programa en ejecución.

Muchas clases de errores pueden provocar una excepción, desde un desbordamiento de memoria o un disco duro estropeado, un intento de dividir por cero o intentar acceder a un arreglo fuera de sus límites.

Cuando algún error como los mencionados ocurre, la máquina virtual Java crea un objeto de la clase *exception*, se notifica el hecho al sistema de ejecución y se dice que se ha lanzado una excepción. Lo que acontece a continuación es que el programa en ejecución es interrumpido por dicho error mostrándolo, como típicamente se suele decir, “el programa truena”.

Un buen control de excepciones es necesario para evitar que esto suceda, de manera tal que si una excepción ocurre al ejecutar un programa éste no muestre el error al usuario, sino por el contrario que le informe la acción que se ha realizado incorrectamente, para que pueda corregirla, esto es mucho más elegante en lugar de que el programa simplemente no se cierre o no responda.

3.3.1. Sentencia Try-catch

Como se ha dicho, si al ejecutar un programa se provoca un error en su ejecución, se generará una excepción interrumpiendo la ejecución del programa. La manera de evitar esto es realizando un manejo de dicha excepción. Esto se efectúa mediante la sentencia *try-catch*.

La sentencia llamada *try-catch* traducida al español sería algo como intenta-atrapa, esto es realmente lo que hace la sentencia, pues dentro del *try* se coloca el código vulnerable a errores, para que el programa lo “intente”, y en caso de que ocurra un error lo “atrape” en la sentencia *catch*. El bloque de sentencias que pudiera tener un error se coloca dentro del bloque *try*{ }, y en el bloque *catch*(){} se coloca el tipo de error y el nombre dado, que por lo general siempre es *e*, entonces una vez que atrapa el error lo imprime a pantalla para que el usuario conozca el error, todo esto sin provocar un cierre inesperado del código.

Observa el siguiente ejemplo:

```
class EjemploExcepcion {
public static void main(String argumentos[ ] ) {
    try{
        int i=5, j=0;
        int k=i/j; // División por cero
}
    catch(java.lang.ArithmeticException e){
        System.out.print("Se produjo el error"+e);
    }
    }
}
```

En el código se puede ver que se declaró la clase llamada *EjemploExcepcion*, un método *main* y dentro del método el bloque *try* tiene declarados dos enteros y una división por cero (intencionalmente, para que al ejecutarlo observes qué pasa), este código es vulnerable a errores, después está declarado el bloque *catch*, donde se atrapa la excepción que se genera y se imprime el error ocurrido.

*Nota: ejecuta el código sin el bloque *try-catch*, y observa el resultado de la ejecución, después coloca el bloque tal como en el ejemplo y date cuenta de que el bloque *try-catch* controla el error.

3.3.2. Tipos de errores

Al controlar las posibles excepciones del código debes tomar en cuenta los posibles errores que puedan ocurrir, para que en la medida de lo posible sean evitados al programar, o en su caso se prevea el manejo de las excepciones que se deberán realizar.

Esencialmente cuando se refiere a los **tipos de errores** en el lenguaje Java, éstos se clasifican dentro de dos tipos básicos:

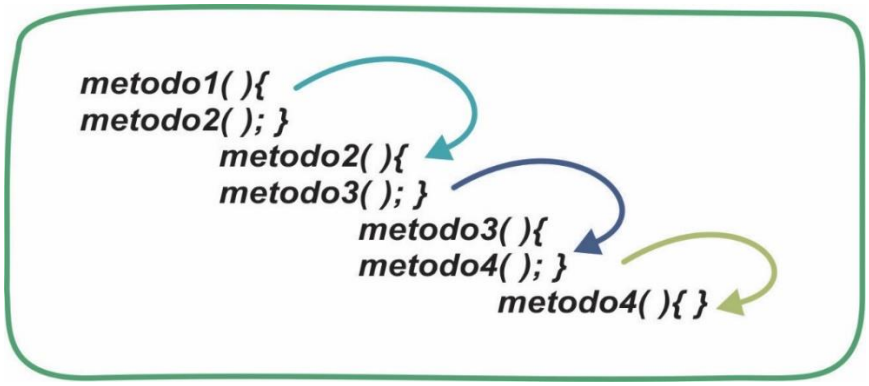
1. **Aquellos generados por el lenguaje Java**, éstos se generan cuando hay errores de ejecución, como al tratar de acceder a métodos de una referencia no asignada a un objeto, división por cero, etc. Es decir, que al programar no se codificó con la sintaxis correcta, y no se previeron estos posibles errores al crear el programa.
2. **Aquellos no generados por el lenguaje**, sino incluidos por el programador, como al tratar de leer un archivo o tratar de hacer conexión a una base de datos específica.

3.3.3. Jerarquía de las excepciones

Cuando un error ocurre y se genera una excepción, se busca donde se está controlando esa excepción, para evitar que el programa inminentemente falle; esto lo hace la máquina virtual Java, la cual recorre la pila de llamadas de métodos en busca de alguno que sea capaz de tratar la clase de excepción lanzada y controlar el error.

Para buscar dónde se controla la excepción, se comienza examinando el método donde se ha producido la excepción; si este método no es capaz de tratarla, entonces se examina el método desde el que se realizó la llamada al método donde se produjo la excepción, y así sucesivamente hasta llegar al último de ellos. En caso de que ninguno de los métodos de la pila sea capaz de tratar la excepción, la máquina virtual Java muestra un mensaje de error y el programa termina.

Para comprender mejor esto, examina el siguiente ejemplo: si se tiene un metodo1 que dentro hace una llamada al metodo2, y el metodo2 manda llamar al metodo3, y el metodo3 llama al metodo4; lo que pasaría si ocurre un error en el metodo1, es que se busca un manejo de excepciones en el metodo1, pero si no se encuentra se va al metodo2 para ver si en este método sí se realiza el manejo de la excepción, pero si ahí tampoco se hizo nada, se va al metodo3 en busca del manejo de la excepción, por último, si tampoco en el metodo3 se hizo nada, se va al metodo4 a buscar el manejo de la excepción. A esto se refiere el texto anterior sobre "ir buscando en las llamadas hasta encontrar donde se pueda manejar la excepción", pero si en ninguno de los métodos se realizó, entonces inevitablemente la excepción hará que el sistema falle.



Ejemplo. Jerarquía de las excepciones

Cierre de la unidad

Has concluido la tercera unidad del curso. A lo largo de ésta te has introducido al manejo de la herencia y el polimorfismo, así como de los conceptos y uso de subclases y superclases, también has visto el uso de métodos y clases abstractos y finales. Además de los temas acerca de interfaces y sobrecarga de métodos y operadores.

Finalizando con lo que respecta al manejo de excepciones mediante el bloque *try-catch*, y los tipos de errores más comunes.

Es recomendable que revises nuevamente la unidad en caso de que los temas que se acaban de mencionar no te sean familiares o no los recuerdes, de no ser el caso, ya estás

preparado(a) para seguir con la *Unidad 4. Arreglos*, en donde se abordará la revisión de arreglos, tanto unidimensionales como multidimensionales.

Para saber más

Consulta la página oficial del lenguaje Java, donde podrás encontrar manuales de referencia sobre el manejo de la herencia, el polimorfismo y las excepciones. Disponible en: http://www.java.com/es/

Fuentes de consulta

Flanagan, D. (1997). *Java in a NutShell, The Java Reference Library.* (Versión 1.2). México: O'Reilly&Associates.

Joyanes, L. y Fernández, M. (2001). *Java 2: Manual de programación*. México: McGraw-Hill.

Martín, A. (2010). *Programador certificado Java 2 curso práctico*. (3a. Ed.). México: Alfaomega.

Niemeyer, P. y Peck, J. (1997). *Exploring Java, The Java Reference Library*. (Versión 1.2). México: O'Reilly&Associates.

Froufe, A. (2008).*JAVA 2 Manual de usuario y tutorial*. (5a. Ed.). México: Alfaomega.