PROGRAMACIÓN ORIENTADA A OBJETOS

CON JAVA 17

ACADEMIA CONSULTEC

www.consultec-ti.com

VALENTÍA | AGILIDAD | EVOLUCIÓN | OPTIMIZACIÓN

Agenda **del día**

2

- Clases y métodos
- Gestión avanzada de errores
- Herencia de Interfaces y Enumerations
- Administración, manipulación y manejo de fechas.
- Programación genérica

ACADEMIA CONSULTEC

Clases y métodos en **JAVA**

ACADEMIA CONSULTEC

Clases

Una clase básicamente está compuesta por:

1. **Modificadores**: Especifique el acceso para la clase.
2. **Nombre de la clase**: Nombre de la clase.
3. **Palabras clave**: Palabras clave que sugieren si una clase se extiende o implementa una clase o interfaz.
4. **El cuerpo** de la clase entre llaves {}

Dependiendo de los diferentes modificadores utilizados, el cuerpo de la clase especificado y las palabras clave utilizadas nos podemos encontrar como usar o determinar su uso.

La clase `Object` (`java.lang.Object`) es la clase principal o la primera clase en Java, está en el nivel más alto de la jerarquía de Java por lo tanto es un objeto público.

ACADEMIA CONSULTEC

Clases

Concrete class:

Es una clase que tiene una implementación para todos sus métodos. No puede tener ningún método sin implementar. También puede extender una clase abstracta o implementar una interfaz siempre que implemente todos sus métodos. Es una clase completa y puede instanciarse.

Abstract class:

Es una clase que está incompleta o cuya implementación no está completa. No se puede crear una instancia de una clase abstracta. Necesita ser extendido por las otras clases e implementar sus métodos para formar una clase concreta.

Una clase abstracta se declara utilizando la palabra clave **`abstract`**. Una clase abstracta puede tener métodos estáticos y finales, así como constructores.

Clases

final class:

Se puede declarar una clase como **`final`**, cuando no nos interesa crear clases derivadas de dicha clase, por tanto, no se puede heredar una clase final.

Declarar una clase **`final`** es generalmente útil para escribir clases inmutables o constante, por ejemplo, la clase **`String`** que se hace generalmente por seguridad.

static class :

Similar a una clase **`final`** no se pueden crear instancias de una clase estática. Se utiliza como una unidad de organización para métodos no asociados a objetos particulares y separa datos y comportamientos que son independientes de cualquier identidad del objeto, por lo tanto todos sus métodos son también estáticos.

Las clases estáticas son adecuadas cuando no tienen que almacenar información, sino sólo realizar cálculos o algún proceso que no cambie.

Clases

POJO class:

POJO significa ***Plain Old Java Object***. En términos simples, usamos ***POJO*** para hacer un modelo de programación para declarar entidades objeto, por lo que es usado como un clase modelo a base de datos. Las clases son fáciles de usar y no tienen ninguna restricción.

Propiedades de las clases ***POJO***:

1. Las clases **POJO** deben ser públicas para que podamos usarlas fuera de la clase.
2. La clase **POJO** debe tener un método **`getter`** y **`setter`** público.
3. Todos los miembros o variables de instancia deben ser privados.
4. La clase **POJO** no extiende ni implementa clases o interfaces que están preespecificadas.
5. No contienen anotaciones predefinidas.
6. No tiene un constructor sin argumentos.

Clases

Internal class (Clase Anidada):

Una clase anidada tiene por objetivo favorecer el encapsulamiento, por tanto, es una clase incluida dentro de otra clase. Al igual que una clase tiene variables y métodos como miembros, también puede tener una clase interna como miembro

Las clases internas tienen los siguientes subtipos:

1. **Clase interna anidada:** Una clase interna anidada tiene acceso a variables miembro privadas de una clase externa. También podemos aplicar modificadores de acceso a la clase interna anidada.
2. **Método de clase interna local:** Esta es la clase interna que se declara dentro de un método de clase externa.
3. **Clase interna anónima:** La clase interna anónima es una clase interna declarada dentro de una clase externa y no tiene nombre.
4. **Clase anidada estática:** De la misma manera que una clase tiene una variable miembro estática, también puede tener una clase estática como miembro.

ACADEMIA CONSULTEC

Clases

Clase Singleton:

Es una clase que está diseñada para restringir la creación de objetos pertenecientes a esta clase o el valor de un tipo a un único objeto. Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella y se puede reconocer por un método de creación estático, que devuelve el mismo objeto guardado en caché.

```
public final class MyClass {
    private static MyClass instance;
    public String value;
    private MyClass(String value) {
        // The following code emulates slow initialization.
        try { Thread.sleep(1000); }
        catch (InterruptedException ex) { ex.printStackTrace(); }
        this.value = value;
    }
    public static MyClass getInstance(String value) {
        if (instance == null) { instance = new MyClass(value); }
        return instance;
    }
}
```

ACADEMIA CONSULTEC

Clases

Clase Sealed (clases selladas) :

Son una mejora al lenguaje incluido con la versión **Java LTS 17** (incluidas desde la versión ***16***). Las clases o interfaces selladas nos permiten restringir que otras clases e interfaces puede extender de ellas.

Una clase o interface es sealed (sellada) aplicando el modificador **`sealed`** a su declaración, luego de cualquier **`extends`** o **`implements`** se debe agregar la cláusula **`permits`** seguida de las clases que permitimos sean extendidas.

```
public abstract sealed class Shape permits Circle, Rectangle, Square, WeirdShape { ... }

public final class Circle extends Shape { ... }
public final class Rectangule extends Shape { ... }
public final class Square extends Shape { ... }
public final class WeirdShape extends Shape { ... }
```

ACADEMIA CONSULTEC

Clases

Sub-classes Sealed permitidas y sus reglas:

Cada subclase permitida debe usar un modificador para describir como propagar el sellado iniciado por su super-clase, por lo que debe de cumplir las siguientes reglas:

1. Una sub-clase permitida puede aplicar el modificar final para evitar extenderse más.
2. Una sub-clase permitida puede aplicar el modificador sealed seguido de la cláusula permits para extender a otras sub-clases en su jerarquía.
3. Una sub-clase permitida puede aplicar el modificar non-sealed de manera que revierte en su propia jerarquía el “sellado” de la clase super clase, y abriendo la extensión a otros clases desconocidas por la super clase.

Clases

Clase Record:

A partir de la versión **Java 14** del **JDK** y en modo *preview* disponemos de los *Java Records*, pero introducidas formalmente en ***Java 15,*** es un concepto que nos permite simplificar de forma fuerte la construcción de DTOs.

Las clases **`record`**, son implícitamente **`final`** por lo cual es consistente con las reglas para las sub-clases permitidas mencionadas en las clases Sealed.

Las clases Record evitan tener que declarar explícitamente los métodos **`getter`** para cada una de las propiedades así como los métodos **`hashCode`** y **`equals`**, métodos que hay implementar correctamente, también evitan declarar el método **`toString`** muy útiles para el correcto funcionamiento cuando las clases se añaden en colecciones.

Object Class

La clase Object proporciona un cierto número de métodos de utilidad general que pueden utilizar todos los objetos

Constructor / Método	**Descripción**
[NombreDeClase].getClass()	Devuelve la clase de tiempo de ejecución del objeto.
Constructor de clases de objetos	
Object()	Constructor de clases de objetos
Métodos de clase de objeto	
protected Object clone()	Devuelve un clon o copia del objeto.
public boolean equals(Object obj);	Comprueba si un objeto determinado obj es igual a este objeto.
protected void finalize();	Lo llama el recolector de basura cuando se destruyen todas las referencias al objeto.
int hashCode ()	Devuelve el código hash del objeto.
public final native void notify();	Úselo para despertar un solo hilo en espera.
public final native void notifyAll();	Despierta todos los hilos en espera.
public String toString();	Devuelve String equivalente al objeto.
public final void wait();	Hace que un hilo espere hasta que otro hilo lo despierte mediante el método de notificación.
public final native void wait(long timeout);	Hace que el hilo espere hasta que transcurra el tiempo dado por el 'tiempo de espera' o se despierte mediante los métodos notificar o notificar a todos.
public final native void wait(long timeout, int nanos)	Hace que el hilo espere hasta que haya transcurrido el tiempo especificado o hasta que otro hilo invoque notificar () o notificar a todos ().

ACADEMIA CONSULTEC

Modificadores de Clases

Modificador	Description
`public`	Indica que la clase es accesible desde cualquier otro lugar en el código, tanto dentro como fuera del paquete en el que se encuentra definida.
`private`	Indica que la clase solo es accesible desde dentro de la misma clase. Esta palabra clave se utiliza para crear clases internas o clases de ayuda que no deben ser visibles desde fuera de la clase que las contiene.
`protected`	Indica que la clase es accesible desde cualquier clase en el mismo paquete y desde cualquier subclase, independientemente del paquete en el que se encuentren.
`default`	Si no se especifica ninguna palabra clave antes de ***class***, la clase tiene un nivel de acceso predeterminado, también conocido como default. En este caso, la clase es accesible solo dentro del mismo paquete en el que se encuentra definida.

Métodos

Los métodos son una herramienta indispensable en JAVA, ya que nos ayuda a reducir la complejidad del código y simplificar la ejecución de determinadas tareas, dividiendo el programa en subrutinas, de esta manera reforzar el concepto de la modularidad y encapsulación.

En Java, un programa tiene muchas subrutinas, pero en términos generales a **todas estas subrutinas del programa se les menciona como métodos**, pero conceptualmente existen 3 tipos de subrutinas: *funciones*, *métodos* y *procedimientos*.

```
[acceso] [modificador] tipoDatoRetorno nombreMetodo([tipoDato nombreArgumento,[tipoDato nombreArgumento]...])
{
    /*
    * Bloque de instrucciones
    */
    return valor;
}
```

ACADEMIA CONSULTEC

Métodos

Funciones:

Son un conjunto de instrucciones, encapsulados en un bloque, usualmente reciben parámetros, cuyos valores utilizan para efectuar operaciones y adicionalmente retornan un valor, si no retorna algo, entonces no es una función. En java las funciones usan el modificador **`static`**.

```
//Método con dos parámetros
static boolean metodoBoolean(boolean n, String mensaje)
{
    //Usamos el parámetro en el método
    if(n) {
        //Mostramos el mensaje
        System.out.println(mensaje);
    }
    //Usamos el parámetro como valor a retornar
    return n;
}
```

ACADEMIA CONSULTEC

Métodos

Métodos:

Los métodos y las funciones en Java están en capacidad de realizar las mismas tareas, son funcionalmente idénticos, pero su diferencia radica en que un método está asociado a un objeto, **SIEMPRE**, básicamente un método es una función que pertenece a un objeto o clase, mientras que una función existe por sí sola, sin necesidad de un objeto para ser usada.

```
//Función sin parámetros
int metodoEntero() {
    int suma = 5+5;
     //Acá termina la ejecución del método
    return suma;
}
```

```
//método con un parámetro
public String metodoString(int n)
{
    //Usamos el parámetro en la función
    if(n == 0){
        //Si n es cero retorna a
        return "a";
    }
    //Este return sólo se ejecuta cuando n NO es cero
    return "x";
}
```

ACADEMIA CONSULTEC

Métodos

Procedimientos:

Un procedimiento es básicamente un método cuyo tipo de retorno es **`void`** que no nos obliga a utilizar una sentencia **`return`**. Por tanto son únicamente instrucciones que se ejecutan sin retornar ningún valor.

```
//Notar el void
void procedimiento(int n, String nombre)
{
    //usamos los dos parámetros
    if(n > 0 && !nombre.equals("")) {
        System.out.println("hola " + nombre);
        //Si no ponemos este return se mostraría hola y luego adiós
       return;
    }
    //También podríamos usar un else en vez del return
    System.out.println("adios");
}
```

ACADEMIA CONSULTEC

Métodos Especiales

Cuando se coloca un atributo privado en una Clase, para poder acceder a él se hacen métodos que dan permisos de Leer (**`GET`**) y Escribir (**`SET`**).

Métodos GET:

Este método devuelve es el atributo deseado, para que lo puedan ver desde fuera de la clase pero no puedan modificarlo.

Métodos SET:

Este método lo que hace es darle valor a un atributo deseado, para que lo puedan ver desde fuera de la clase pero no puedan modificarlo.

Constructor

Es un método que evita tener que llenar los valores de cada objeto. Suele ser el primer método que se encuentra en cada clase, justo después de la declaración de atributos.

Aquí te coloco sus características mas comunes:

1. Tiene el mismo nombre que la clase
2. Generalmente es público
3. Su objetivo (su motivo en la vida) es inicializar el estado de los atributos de una clase
4. Se puede sobrecargar

```
public class Construtor {

    String cadena = "";
    int number = 0;
    ObjectX object = null;

    public Construtor (String cadena, ObjectX object){
        this.cadena = cadena;
        this.number = object.number;
        this.object = new ObjectX(object.number);
    }
}
```

ACADEMIA CONSULTEC

Modificadores de los métodos

public:

Se utiliza para indicar que el método es accesible desde cualquier parte del programa.

```
public int suma (int numberA, int numberB){
    return numberA + numberB;
}
```

private:

Se utiliza para indicar que el método solo es accesible desde dentro de la misma clase en la que se ha definido

```
private boolean validarIdentificacion (String numberId){
    //código validacion
}
```

protected:

Se utiliza para indicar que el método es accesible desde dentro de la misma clase y también desde las subclases que hereden de esta clase.

```
protected double calcularArea(){
    //código de calculo
}
```

ACADEMIA CONSULTEC

Modificadores de los métodos

default :

No se especifica modificador, y el método es accesible desde cualquier clase que se encuentre en el mismo paquete, pero no desde clases que estén fuera de este paquete.

```
void imprimirSaludo (){
    system.out.printl(“Hola Mundo");
}
```

static:

Se utiliza para indicar que el método es un método de clase que se puede llamar sin necesidad de crear una instancia de la clase en la que se ha definido.

```
public static int calcularfactorial(int n) {
    int resultado = 1;
    for (int i = 1; i <= n; i++) {
        resultado *= i;
    }
    return resultado;
}
```

ACADEMIA CONSULTEC

Modificadores de los métodos

final :

Se utiliza para indicar que el método no puede ser sobrescrito por las subclases que hereden de esta clase.

```
public final void saludo (){
    system.out.printl(“Saludo a todos");
}
```

abstract:

Se utiliza para indicar que el método no tiene implementación en la clase en la que se ha definido, sino que debe ser implementado en las subclases que hereden de esta clase.

```
public abstract double calcularPerimetro();
```

ACADEMIA CONSULTEC

Modificadores de los métodos

synchronized:

Se utiliza para garantizar que solo se puede acceder a un bloque de código o un método mediante un hilo (thread) a la vez. Es decir, si un hilo está ejecutando un bloque de código o un método sincronizado, ningún otro hilo podrá acceder a ese mismo bloque de código o método hasta que el hilo actual haya terminado de ejecutarlo.

```
public void sentenciaSincronizada (){
    // CÓDIGO NO SINCRONIZADO
    synchronized(this) {
        // CÓDIGO SINCRONIZADO
        system.out.printl(“Solo me muestro una vez por hilo de ejecucion");
    }
    // CÓDIGO NO SINCRONIZADO
}
```

ACADEMIA CONSULTEC

Sobrecarga

Entendiendo que la firma de un método es la combinación del nombre y los tipos de los parámetros o argumentos, la sobrecarga hace referencia a un método al cual se le pueden pasar diferentes tipos o números de argumentos, por tanto, el método mantiene el mismo nombre pero cambia el valor de retorno o los valores de entrada.

Y la sobrecarga existen de 2 maneras: *Sobrecarga de Métodos* y *Sobrecarga de Constructores*.

```
public class Catalogo {
    private List<Producto> productlist;

    public Catalogo(List<Producto> productoList) {
        this.productlist = productoList;
    }

    public Catalogo(Producto producto) {
        this.productlist = new ArrayList<Producto>();
        this.productlist.add(producto);
    }
}
```

```
addProducts(List<Producto> productos)
```

```
addProducts(Producto productos)
```

ACADEMIA CONSULTEC

Sobrecarga vs sobreescritura de métodos

Sobrecarga

Permite declarar métodos que se llamen igual pero que reciban parámetros diferentes (no pueden haber 2 métodos con el mismo nombre y los mismos parámetros), por esta razón lo que define a qué método se ingresa, son los argumentos que se envían como parámetros.

Sobreescritura

Es la forma por la cual una clase que hereda puede redefinir los métodos de su clase Padre, de esta manera puede crear nuevos métodos con el mismo nombre de su superclase.

ACADEMIA CONSULTEC

Sobreescritura de métodos

Este concepto se genera a partir de la concepción de una herencia de una clase, por tanto, una subclase hereda todos los métodos de su superclase que son accesibles a dicha subclase a menos que la subclase sobrescriba los métodos.

Una subclase sobrescribe un método de su superclase cuando define un método con las mismas características (nombre, número y tipo de argumentos) que el método de la superclase. Las subclases emplean la sobreescritura de métodos la mayoría de las veces para agregar o modificar la funcionalidad del método heredado de la clase padre.

```
class ClaseB extends ClaseA {
    /* Estos métodos sobreescriben a los métodos de la clase
padre */
    void miMetodo (int var1 ,int var2)
    { ... }

    String miOtroMetodo( )
    { ... }
}
```

```
class ClaseA {
    void miMetodo(int var1, int var2)
    { ... }

    String miOtroMetodo( )
    { ... }
}
```

ACADEMIA CONSULTEC

Gestión Avanzada de **Errores**

ACADEMIA CONSULTEC

Gestión de Errores

Durante la ejecución de un programa es posible se genere una situación que puede provocar un fallo, estos fallos se denominan ***excepciones*** en JAVA**.**

Las ***excepciones*** son un mecanismo diseñado para gestionar la aparición de una condición que cambia el flujo de ejecución normal de los *programas* y permitiendo tratar los errores en un código o flujo separado.

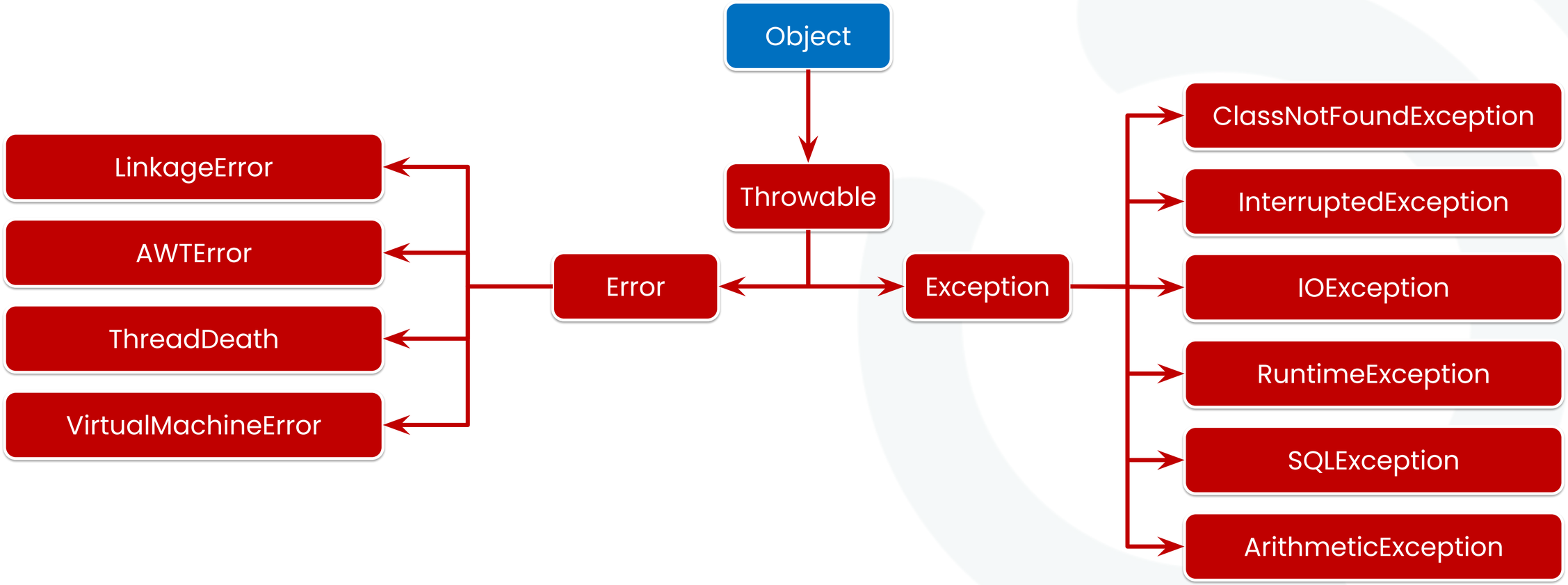
Superclass Throwable:

Java define una excepción como un objeto que es una instancia de la clase **`Throwable`**, y existe diferentes tipo de excepciones dependiendo del error generado.

La clase **`Throwable`** tiene 2 hijos directos: **`Error`** y **`Exception`**.

ACADEMIA CONSULTEC

La jerarquía de Throwable



ACADEMIA CONSULTEC

Constructores Throwable

Cualquier clase puede tener cualquiera de los tres o los tres tipos de constructores. Son constructores por defecto, parametrizados y no parametrizados.

Constructores Públicos:

1. **Throwable():** Es un constructor no parametrizado que construye un nuevo `Throwable` con `null` como mensaje detallado.
2. **Throwable(String mensaje)**: Es un constructor parametrizado que construye un nuevo `Throwable` con el mensaje detallado específico.
3. **Throwable(String mensaje, Throwable causa):** Es un constructor parametrizado que construye un nuevo `Throwable` con el mensaje detallado específico y una causa.
4. **Throwable(Throwable causa):** Es un constructor parametrizado que construye un nuevo `Throwable` con la causa específica y un mensaje detallado de la causa convirtiendo la causa a `String` mediante el método `toString()`.

ACADEMIA CONSULTEC

Constructores Throwable

Constructores protegidos :

Throwable(String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace): Construye un nuevo throwable con el mensaje detallado especificado, la causa, la supresión habilitada o deshabilitada y el seguimiento de pila escribible habilitado o deshabilitado.

Los parámetros son:

1. `message` - el mensaje detallado.
2. `cause` - la causa. (Se permite un valor nulo, e indica que la causa es inexistente o desconocida).
3. `enableSuppression` - si la supresión está habilitada o deshabilitada.
4. `writableStackTrace` - si el seguimiento de pila debe ser escribible o no.

Methods Throwable

Aparte de los constructores mencionados anteriormente, también hay muchos métodos predefinidos disponibles en la clase **`throwable`**. Pero se mencionaran los mas importantes y útiles:

1. **getMessage():** Devuelve la cadena del mensaje detallado del **`Throwable`** actual.

   Sintaxis: `public String getMessage()`

   Devuelve: la cadena de mensaje detallada de la instancia **`Throwable`** actual (también puede devolver null).

2. **printStackTrace():** Imprime el **`Throwable`** actual y su seguimiento en el flujo de error estándar.

   Sintaxis: `public void printStackTrace()`

   Devuelve: Este método no devuelve nada.

3. **toString():** Este método devuelve una breve descripción del objeto **`Throwable`** actual.

   Sintaxis: `public String toString()`

   Devuelve: una representación de cadena del lanzable actual..

ACADEMIA CONSULTEC

Tipos de Excepciones

Checked Exceptions:

Por motivos de comprobación de excepciones en tiempo de compilación, **`Throwable`** y cualquier subclase de **`Throwable`** que no sea también una subclase de **`Error`** o **`RuntimeException`** se consideran **checked exceptions**. Una excepción de tipo **checked** debe ser capturada o bien especificar que puede ser lanzada de forma obligatoria, y si no lo hacemos obtendremos un error de compilación.

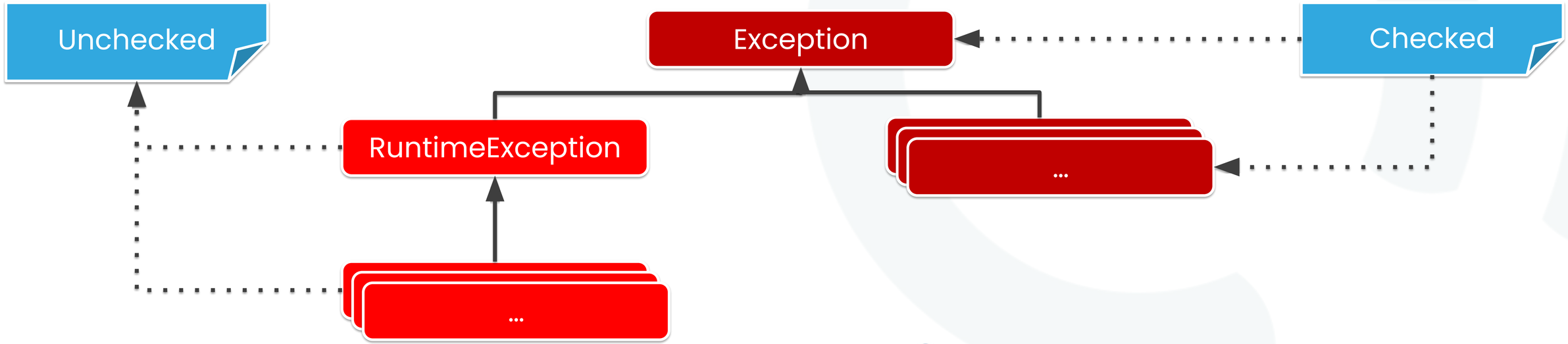
Todas las excepciones de este tipo son subclases que heredan desde **`Exception`**, como por ejemplo: **`SocketTimeoutException`** ,**`IOException`**, **`DataFormatException`**, **`FileNotFoundException`**.

ACADEMIA CONSULTEC

Tipos de Excepciones

Unchecked Exceptions:

Generalmente este tipo de excepciones son lanzadas por la aplicación y se generan a partir de errores en tiempo de **Runtime** que representan errores en el código y que la aplicación no es capaz de controlar, por tanto, las excepciones de tipo *Unchecked* son subclases que heredan de este o de cualquiera otra clase que herede de ella.



ACADEMIA CONSULTEC

Control de Excepciones

Para capturar y manejar las excepciones, Java proporciona las siguientes sentencias: **`try`**, **`catch`** y **`finally`**

1. **Bloque `try`:** Contiene el código regular de nuestro programa que puede producir una excepción en caso de error.
2. **Bloque `catch`:** Contiene el código con el que trataremos el error en caso de producirse.
3. **Bloque `finally`:** Este bloque contiene el código que se ejecutará al final tanto si se ha producido una excepción como si no lo ha hecho. Este bloque se utiliza para, por ejemplo, cerrar algún fichero que haya podido ser abierto dentro del código regular del programa, de manera que nos aseguremos que tanto si se ha producido un error como si no este fichero se cierre. El bloque **`finally`** no es obligatorio ponerlo

```
try {
    ... // Aqui va el codigo que puede lanzar una excepcion
} catch (Exception e) {
    System.out.println ("El error es: " + e.getMessage());
    e.printStackTrace();
} finally {
    // Código de finalización (opcional)
}
```

ACADEMIA CONSULTEC

Gestión automática de recursos

Java proporciona una función para hacer el código más robusto y reducir las líneas de código. Esta característica se conoce como **Automatic Resource Management(ARM)** utilizando ***try-with-resources*** desde Java 7 en adelante.

Esta sentencia asegura que cada recurso se cierra al final de la sentencia, lo que facilita el trabajo con recursos externos que necesitan ser eliminados o cerrados en caso de errores o de finalización con éxito de un bloque de código. Cualquier objeto que implemente **`java.lang.AutoCloseable`**, que incluye todos los objetos que implementan **`java.io.Closeable`**, puede ser utilizado como recurso.

Sintaxis:

```
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

ACADEMIA CONSULTEC

Receso

15 min

Herencia en **Interfaces**

ACADEMIA CONSULTEC

Herencia de Interfaces

En Java la herencia de interfaz se presenta cuando heredamos una lista de propiedades y métodos vacíos para que su codificación sea personalizada en dicha clase.

Este tipo de herencia es una solución elegante para solventar la no posibilidad de herencia múltiple, por el cual una interfaz es una especie de contrato donde se definen una serie de funciones vacías sin aportar su implementación.

Existen tres reglas clave para poder aplicar la herencia de interfaces sin problemas:

1. Las variables en las interfaces son siempre **`public`**, **`static`** y **`final`**.
2. Los métodos de las interfaces en Java son siempre **`public`**.
3. Los métodos estáticos de las interfaces NO se heredan.

Composición de Interfaces en Clases

Las interfaces se crean utilizando la palabra reservada **interface** y se implementan en nuestras clases con **implements**.

```
public interface IMove {
    void moveOn(int velocidad);
}
```

```
public class Car implements IMove {
    @Override
    public void moveOn(int velocidad)
    {
        // ...
    }
}
```

```
public class Dog implements IMove {
    @Override
    public void moveOn(int velocidad)
    {
        // ...
    }
}
```

ACADEMIA CONSULTEC

Enumerations

Java 5 introdujo por primera vez la palabra clave **`enum`** y denota un tipo especial de clase que siempre extiende la clase **`java.lang.Enum`**, y se limitan a la creación de objetos a los especificados explícitamente en la implementación de la clase.

Las constantes definidas de esta manera hacen que el código sea más legible, permiten la verificación en tiempo de compilación, documentan la lista de valores aceptados por adelantado y evitan comportamientos inesperados debido a que se pasan valores no válidos.

Puntos a recordar para Java Enum

1. Mejora la seguridad de tipos.
2. Se puede usar fácilmente en switch.
3. Puede ser transversal.
4. Puede tener campos, constructores y métodos.
5. Puede implementar muchas interfaces pero no puede extender ninguna clase porque internamente extiende la clase Enum.

ACADEMIA CONSULTEC

Métodos heredados de Enum

Todo tipo **`enum`** hereda de la clase **`Enum`** de Java. Los métodos que hereda de ella son:

1. **public final boolean equals(Object other):** Devuelve TRUE si el objeto especificado es igual a esta constante. Sobrescribe el método **“equals”** de **Object**.

2. **public final String name():** Devuelve en nombre de esta constante, tal y como fue declarada.

3. **public final int ordinal():** Devuelve la posición de la constante según la declaración de éstas. A la primera constante declarada se le asigna 0.

4. **public String toString():** Devuelve lo mismo que **name()**, pero sobrescribe **“equals”** de **Object**.

5. **public static enumConstant[] values():** Devuelve un array con las constantes declaradas.

ACADEMIA CONSULTEC

Composición de Enumerations

Forma Simple:

```
public enum Operaciones { SUMA, RESTA, MULTIPLICACION, DIVISION }
```

Forma Simple dentro de una Clase:

```
public class Riesgo {
    public enum CalificacionRiesgo { AAA, AA, A, BBB, BB, B, C, D, F};

    private edadCliente edad;
    private CalificacionRiesgo clasificacion;

    Riesgo (int edad, CalificacionRiesgo clasificacion) {
        this.edad = edad;
        this.clasificacion = clasificacion;
    }

    public void imprimir() {
        System.out.println(edad + "-" + CalificacionRiesgo.toString().toLowerCase());
    }
}
```

ACADEMIA CONSULTEC

Composición de Enumerations

Constante por asociación:

```
public enum DiaSemana {

    DOMINGO(0), LUNES(1), MARTES(2),
    MIERCOLES(3), JUEVES(4), VIERNES(5),
    SABADO(6);

    public final int nroDia;

    private DiaSemana(int nroDia) {
        this.nroDia = nroDia;
    }

    public int getNroDia() {
        return nroDia;
    }

    public String getNombreDia() {
        return name();
    }
}
```

ACADEMIA CONSULTEC

Composición de Enumerations

Con sobrecarga de método:

```
public enum CoffeeSize {
    BIG(8), HUGE(10),
    OVERWHELMING(16){
          @Override
        public String getLidCode(){//sustituye el método definido
            return "A";
        }`
    };

    CoffeeSize(int ounces){
        this.ounces = ounces;
    }

    private int ounces;

    public int getOunces(){
        return ounces;
    }
    public String getLidCode(){ //Este es el método sustituido
        return "B"; //El valor por defecto que se retorna
    }
}
```

ACADEMIA CONSULTEC

Almuerzo

45 min

Administración de **Fechas**

ACADEMIA CONSULTEC

Manejo de fechas y tiempo

Antes de la versión de Java 8, se estuvo trabajando con la clase **`Date`** y la clase **`Calendar`**, y se conocen por generar muchos problemas a la hora de gestionar fechas, tiempo y zonas horarias con estas clases unidas a la clase **`DateFormat`**. Por lo que manejar fechas se dividía en tres responsabilidades fundamentales:

1. Almacenar el valor de la fecha (Java Date)
2. Aplicar diferentes formatos a la fecha (Java DateFormat y SimpleDateFormat)
3. Hacer cálculos de fechas (Java Calendar)

Por tanto Java 8 introdujo un conjunto de nuevas API dentro del paquete `java.time` para Fecha y Hora que ayudo a resolverlo sin incluir dependencias adicionales, con el fin de solventar las deficiencias.

ACADEMIA CONSULTEC

Manejo de fechas

Clase Instant

Esta clase modela un instante en el tiempo, es la encargada de guardar un instante con toda su definición (fecha, tiempo hasta el nanosegundo) en un campo de tamaño long.

La clase Instant la vamos a conseguir, la mayoría de las veces, desde otras clases de tiempo, normalmente a través del método instant() que casi todas proporcionan.

```
//Fijar momento actual
Instant ahora = Instant.now();
//Fijar instante segun hora PA
Clock clockPA = Clock.system(ZoneId.of("America/Panama"));
Instant ahoraPA = Instant.now(clockPA);
Instant parseado = Instant.parse("2022-11-10T08:00:00.00Z");
System.out.println("ahora :" + ahora);
System.out.println("ahoraPA :" + ahoraPA); System.out.println("parseado:" +
parseado);
```

En consola aparece:

```
ahora    :2023-03-30T20:10:19.402220400Z
ahoraPA :2023-03-30T20:10:19.427257500Z
parseado:2022-11-10T08:00:00Z
```

ACADEMIA CONSULTEC

Clase LocalDate

Es un objeto de tiempo, inmutable, que guarda la fecha como una etapa en formato ISO (yyyy-MM-dd). Es uno de los objetos que podemos proponer para cambiar por Date o Calendar.

Tomar en cuenta que no se puede considerar que pueda representar tiempo, porque no contiene ni el desplazamiento, ni la zona a que pertenece la fecha, sin embargo ya es útil para establecer la fecha actual.

```
// fecha actual en el momento
LocalDate date = LocalDate.now();
System.out.println(date);

// otras formas de crear un LocalDate
LocalDate date2 = LocalDate.of(2022, 03, 07);
System.out.println("Entrada por parametro " + date2);

LocalDate date3 = LocalDate.parse("2022-03-07");
System.out.println("Entrada por String " + date3);

// agrego 10 dias
LocalDate newDate = date3.plusDays(10);
System.out.println("Agrego 10 dias " + newDate);

// agrego 3 meses
newDate = date3.plusMonths(3);
System.out.println("Agrego 3 meses " + newDate);
```

En consola aparece:

```
2023-03-30
Entrada por parametro 2022-03-07
Entrada por String 2022-03-07
Agrego 10 dias 2022-03-17
Agrego 3 meses 2022-06-07
```

ACADEMIA CONSULTEC

Clase LocalTime

Es clase puede guardar y trabajar con horas, minutos y segundos, pero sin especificar la zona horaria a la que esos datos pueden pertenecer. La precisión con la que trabaja es de nanosegundos, con lo que podemos conservar en un campo de dicha clase un valor de tipo «13:45.30.123456789»

```
LocalTime time2 = LocalTime.parse("11:00:59.759");
LocalTime time3 = LocalTime.of(11, 14, 45, 200);
System.out.println("Entrada por cadena " + time2);
System.out.println("Entrada por parametro " + time3);

//valido si una hora es mayor / antes de
System.out.println("tiempo2 es antes del tiempo3?: " + time2.isBefore(time3));

//agrego una hora
LocalTime time4 = time3.plusHours(1);
System.out.println("Agrego 1hr " + time4);
```

En consola aparece:

```
Entrada por cadena 11:00:59.759
Entrada por parametro 11:14:45.000000200
tiempo2 es antes del tiempo3?: true
Agrego 1hr 12:14:45.000000200
```

ACADEMIA CONSULTEC

Clase LocalDateTime

Es capaz de guardar fecha y hora, pero sin especificación de zona ni offset. Como el resto de objetos de este package, es un objeto inmutable y thread-safe en que podemos guardar los datos de fecha y hora y acceder a todas las herramientas del paquete

```
LocalTime time2 = LocalTime.parse("11:00:59.759");
LocalTime time3 = LocalTime.of(11, 14, 45, 200);
System.out.println("Entrada por cadena: " + time2);
System.out.println("Entrada por parametron: " + time3);

//valido si una hora es mayor / antes de
System.out.println("tiempo2 es antes del tiempo3?: " + time2.isBefore(time3));

//agrego una hora
LocalTime time4 = time3.plusHours(1);
System.out.println("Agrego 1hr: " + time4);
```

En consola aparece:

```
Entrada por cadena: 11:00:59.759
Entrada por parámetro: 11:14:45.000000200
tiempo2 es antes del tiempo3?: true
Agrego 1hr 12:14:45.000000200
```

ACADEMIA CONSULTEC

Clase ZonedDateTime

ZonedDateTime se utiliza cuando queremos trabajar con fechas y tiempo pero agrega el factor de las zonas horarias, para esto utiliza un ZoneId el cual es un identificador para diferentes zonas.

```
ZoneId zoneIdPa = ZoneId.of("America/Panama");
ZoneId zoneIdAm = ZoneId.of("Europe/Amsterdam");
ZonedDateTime departureDate = ZonedDateTime.of(2018, 10, 31, 2, 39, 0, 0,
zoneIdPa);

System.out.println("Departure date: " + departureDate);
System.out.println("Arrival in Panama time: " + departureDate.plusHours(12));
System.out.println("Arrival in Amsterdam time: " +
ZonedDateTime.ofInstant(departureDate.plusHours(12).toInstant(), zoneIdAm));
```

En consola aparece:

```
Departure date:
2018-10-31T02:39-05:00[America/Panama]
Arrival in Panama time:
2018-10-31T14:39-05:00[America/Panama]
Arrival in Amsterdam time:
2018-10-31T20:39+01:00[Europe/Amsterdam]
```

ACADEMIA CONSULTEC

Clase Period

La clase Period se utiliza para modificar valores de una fecha u obtener la diferencia entre dos fechas.

```
LocalDate currentTime = LocalDate.now();

LocalDate myBirthDate = LocalDate.parse("2023-08-02");

Period period = Period.between(currentTime, myBirthDate);

System.out.println(
        String.format("Years %d Months %d Days %d",
                period.getYears(),
                period.getMonths(),
                period.getDays()
        )
);
```

En consola aparece:

```
Years 0 Months 4 Days 2
```

ACADEMIA CONSULTEC

Clase Duration

La clase Duration funciona de forma similar que Period la única diferencia es que en lugar de trabajar con fechas trabaja con tiempo.

```
LocalTime currentTime = LocalTime.of(8, 00);
LocalTime timeToLeave = LocalTime.of(17, 30);
Duration duration = Duration.between(currentTime, timeToLeave);
System.out.println(
        String.format("Minutes  %s", duration.toMinutes()
        )
);
```

En consola aparece:

```
Minutes  570
```

ACADEMIA CONSULTEC

Conversiones y Compatibilidad

Una de los detalles actuales son las migraciones de clases, existiendo muchas veces clases imposibles de migrar y manteniendo el comportamiento con las clase Date, Java 8 con la clase Instant nos permite dar una solución a este problema.

```
//America/Panama GTM -05:00
ZoneId defaultZoneId = ZoneId.systemDefault();
System.out.println("System Default TimeZone : " + defaultZoneId);

//toString() append +8 automatically.
Date date = new Date();
System.out.println("date : " + date);

//1. Convert Date -> Instant
Instant instant = date.toInstant();
System.out.println("instant : " + instant); //Zone : UTC+0
```

En consola aparece:

```
System Default TimeZone : America/Panama
date : Fri Mar 31 03:52:16 EST 2023
instant : 2023-03-31T08:52:16.884Z
```

ACADEMIA CONSULTEC

Conversiones y Compatibilidad

Luego de obtener una instancia de la clase Instant, solo debemos usar los métodos toLocalDate(), toLocalDateTime() o atZone(), para obtener un instancia de la clase correspondiente.

```
//2. Instant + system default time zone + toLocalDate() = LocalDate
LocalDate localDate = instant.atZone(defaultZoneId).toLocalDate();
System.out.println("localDate : " + localDate);

//3. Instant + system default time zone + toLocalDateTime() = LocalDateTime
LocalDateTime localDateTime =
instant.atZone(defaultZoneId).toLocalDateTime();
System.out.println("localDateTime : " + localDateTime);

//4. Instant + system default time zone = ZonedDateTime
ZonedDateTime zonedDateTime = instant.atZone(defaultZoneId);
System.out.println("zonedDateTime : " + zonedDateTime);
```

En consola aparece:

```
localDate : 2023-03-31
localDateTime : 2023-03-31T03:52:16.884
zonedDateTime :
2023-03-31T03:52:16.884-05:00[America/Panama]
```

ACADEMIA CONSULTEC

Receso

15 min

Programación **Genérica**

ACADEMIA CONSULTEC

Programación Genérica

La programación genérica surge a partir de la versión 5.0 de Java en el año 2004. Es un paradigma informático que nos indica que tenemos que crear un código que podamos reutilizar para otros objetos de diverso tipo, y su base es un tipo de programación que está mucho más centrada en los algoritmos que en los datos.

Ventajas de la programación genérica:

1. Hace nuestro código más sencillo de mantener y leer.
2. Nos evitan duplicar clases que administran tipos de datos distintos pero implementan algoritmos similares
3. Podemos reutilizar código de una forma más rápida.
4. Los errores nos acontecerán en la compilación y no en tiempo de ejecución, el IDE (entorno desarrollo) nos avisará y podremos rectificar antes de programar más líneas de código.
5. Al pasar un parámetro tipo queda más claro en nuestro código que tipo de dato estamos manejando.

ACADEMIA CONSULTEC

Raw Types

A pesar de que desde Java 5, el concepto de Genérico permitía parametrizar los argumentos de tipo para las clases, incluidas las de la API de Colecciones, al declarar y construir objetos, aun se presentaron algunos problema y es que los tipos brutos seguían existiendo por motivos de compatibilidad con versiones anteriores, por lo que el compilador debe diferenciar entre estos tipos brutos y los genéricos:

```
List<String> java5List = new ArrayList<String>();
Map<String, List<Map<String, Map<String, Integer>>>> java5Map
        = new HashMap<String, List<Map<String, Map<String, Integer>>>>();
List<String> java6List = new ArrayList<String>();
List<String> java4List = new ArrayList();
```

Aunque el compilador todavía nos permite utilizar tipos raw en el constructor, nos avisará con un mensaje de advertencia: ArrayList is a raw type. References to generic type ArrayList<E> should be parameterized

ACADEMIA CONSULTEC

Diamond Operator

El operador diamante fue introducido en Java 7, añade inferencia de tipos y reduce la verbosidad en las asignaciones cuando se utilizan genéricos.

```
List<String> java7List = new ArrayList<>();
```

El operador diamante es básicamente azúcar sintáctico (syntactic sugar) de Java, que nos va a permitir quitarnos cierta verbosidad a la hora de escribir nuestro código, y acorde al ejemplo el compilador sabe que la variable java7List va a ser de tipo String al estar declarado en la parte izquierda.

```
public interface Engine { }
public class Diesel implements Engine { }
public interface Vehicle<T extends Engine> { }
public class Car<T extends Engine> implements Vehicle<T> { }
Car<Diesel> myCar = new Car<>();
```

ACADEMIA CONSULTEC

Generic Class

Las clases genéricas nos evitan duplicar clases que administran tipos de datos distintos, pero implementan algoritmos o una lógica similar, en pocas palabras evitan la redundancia de código.

La regla principal es entender que puedes inferir sobre un tipo (objeto o atributo) a la vez y no 2 al mismo tiempo, este parámetro expuesto se define como una carácter entre el operador diamante<>.

```
import java.util.ArrayList;
public class BolsaGenerica < T > {
    private ArrayList < T > lista = new ArrayList < T >();
    public void add(T objeto){
        lista.add(objeto);
    }
    public ArrayList<T> getProducts(){
        return lista;
    }
}
```

ACADEMIA CONSULTEC

Generic Class

Para utilizar una clase genérica, al momento de instanciar se deberá de mandar como argumento el tipo de dato que vamos a querer almacenar en nuestra clase genérica.

```
BolsaGenerica<Soda> BolsaGenericaDeSodas = new BolsaGenerica<>();
```

En caso de presentarse un error en el momento de compilación se interpretara un error en el IDE, como se menciono anteriormente como se trata realmente de *“sintaxis sugar”* el mismo no existen realmente en la JVM, sino que se hace un construcción de la clase con el tipo de valor al momento de ejecución.

```
public class BolsaGenerica {
    private ArrayList < Soda > lista = new ArrayList < Soda >();
    public void add(Soda objeto){ lista.add(objeto); }
    public ArrayList<Soda> getProducts(){ return lista; }
}
```

ACADEMIA CONSULTEC

Generic Class Limitado

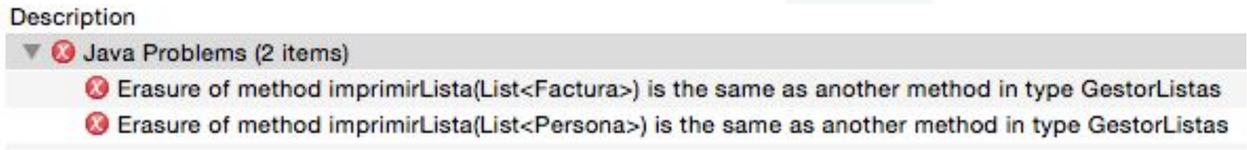
Las clases genéricas de tipos limitados se utilizan para añadir restricciones a los tipos de parámetros y se declaran utilizando la palabra clave extends.

```
public class BolsaGenerica < T extends Bebidas> {
    private ArrayList< T > lista = new ArrayList < T >();
    public void add(T objeto){ lista.add(objeto); }
    public ArrayList<T> getProducts(){ return lista; }
}
```

BolsaGenerica < T extends Bebidas> significa que la clase BolsaGenerica sólo puede instanciarse con el objecto o parámetro que extienda de un tipo Bebidas, en resumen, el único tipo parametrizado soportado en este caso es BolsaGenerica<Bebidas>. Si se intenta instanciar con otro tipo se producirá un error de compilación.

Generics Erasure

Uno de los puntos más importantes que hay que entender al trabajar con Genéricos es sobre la interpretación que realiza el JVM, estas sintaxis **(“sintaxis sugar”)** no se interpreta igual durante la ejecución de programa, esto es debido a que en su momento se llegó a la conclusión que implementarlos directamente en la máquina virtual era muy costoso.



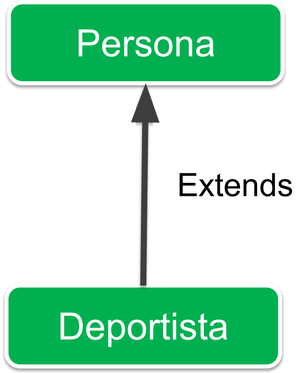

```java
// Estructura generado en codigo
public class GestorListas {
    public void imprimirLista(List<Factura> listaFacturas) {
        for (Factura p: listaFacturas) { System.out.println(p); }
    }
    public void imprimirLista(List<Persona> listaPersonas) {
        for (Persona p: listaPersonas) { System.out.println(p); }
    }
}
```

```java
//Interpretacion de la clase generada por JVM
public class GestorListas {
    public void imprimirLista( List listaPersonas) {
        for (Object p: listaPersonas) {
            System.out.println(((Persona)p).getNombre());
        }
    }
    public void imprimirLista(List listaFacturas) {
        for (Object f: listaFacturas) {
            System.out.println(((Factura)p).getConcepto());
        }
    }
}
```

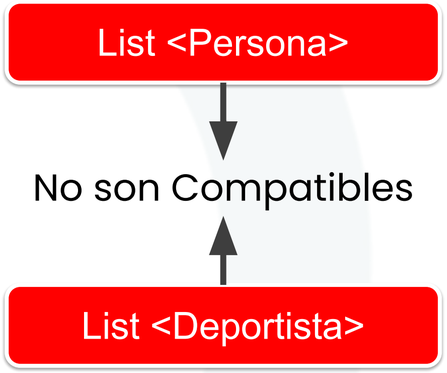
ACADEMIA CONSULTEC

Generic Class Wilcard

Cuando se establecen diferentes instancias de un tipo genérico, los mismos no son compatibles entre sí, ni siquiera explícitamente. Así pues, al tratarse de dos tipos de objetos que realmente no están relacionados por herencia no nos podemos apoyar en el **polimorfismo** para gestionarlos de forma común.




```
public static void main(String[] args) {
    List<Deportista> listaPersonas=new ArrayList<>();
    listaPersonas.add(new Deportista("pepe","futbol"));
     // se genera un error de compatibilidad
    imprimir(listaPersonas);
}
// El código no copila y responde el siguiente error: The method
imprimir(List<Persona>) in the type <Class> is not applicable for the
arguments (List<Deportista>)
public static void imprimir(List<Persona> lista) {
    for(Persona p:lista) { System.out.println ("Nombre de la persona :" p.getNombre()); }
}
```



ACADEMIA CONSULTEC

Generic Class Wilcard

Para esta incompatibilidad nos apoyamos con el carácter wilcard **(?)** conocido como comodín en programación genérica, de manera que añadiremos flexibilidad y podemos decirle lenguaje Java que cuando usemos un tipo genérico se puede aplicar cualquier tipo al parámetro.

```
public static void main(String[] args) {
    List<Deportista> listaPersonas=new ArrayList<>();
    listaPersonas.add(new Deportista("pepe","futbol"));
    // se genera un error de compatibilidad
    imprimir(listaPersonas);
}
public static void imprimir(List<? extends Persona> lista) {
    for(Persona p:lista) {
        System.out.println("Nombre de la persona: " + p.getNombre());
    }
}
```

En consola aparece:

```
Nombre de la persona: pepe
```

ACADEMIA CONSULTEC

Generic Methods

Cuando nos referimos a métodos genéricos estamos hablando de métodos que son genéricos sin pertenecer a una clase genérica, escribimos métodos genéricos con una única declaración de método, y podemos llamarlos con argumentos de distintos tipos.

Estas son algunas propiedades de los métodos genéricos:

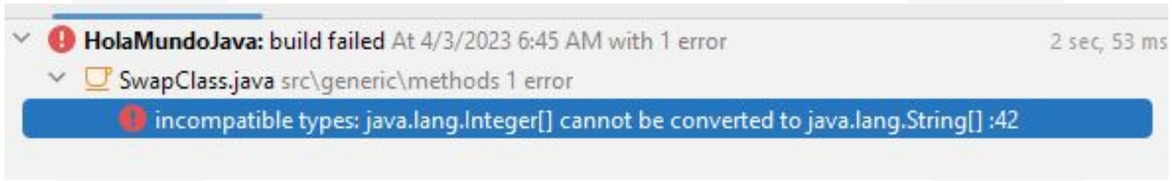
1. Los métodos genéricos tienen un parámetro de tipo (el operador rombo <> que encierra el tipo) antes del tipo de retorno de la declaración del método.
2. Los parámetros de tipo pueden estar identificados por el especificador de parámetro genérico colocado delante del tipo de retorno.
3. Los métodos genéricos pueden tener diferentes parámetros de tipo separados por comas en la firma del método.
4. El cuerpo de un método genérico es igual que el de un método normal.

Generic Methods

Método Común vs Método Genérico:

```
static void printList(String[] list){
    for(String i: list){
        System.out.println(i);
    }
}
public static void main(String[] args) {
    String[] animals={"Cat","Dog","Tiger","Elephant"};
    //call method; printList
    System.out.println("List of animals in zoo:\n");
    printList(animals);

    Integer[] number={1,4,8,5,2,6};
    //call method; printList
    System.out.println("List of number:\n");
    printList(number);
}
```



ACADEMIA CONSULTEC

Generic Methods

Método Común vs Método Genérico:

```
static <E> void printList(E[] list){
    for(E i: list){
        System.out.println(i);
    }
}
public static void main(String[] args) {
    String[] animals={"Cat","Dog","Tiger","Elephant"};
    //call method; printList
    System.out.println("List of animals in zoo:\n");
    printList(animals);

    Integer[] number={1,4,8,5,2,6};
    //call method; printList
    System.out.println("List of number:\n");
    printList(number);
}
```

En consola aparece:

```
List of animals in zoo:
Cat
Dog
Tiger
Elephant
List of number:
1
4
8
5
2
6
```

ACADEMIA CONSULTEC

Gracias

¡Nos vemos pronto!