Icono *Training Consulting*

Formación

Curso JAVA: Novedades y Buenas Prácticas versiones LTS

2024

Formador

Ana Isabel Vegas

INGENIERA INFORMÁTICA con Master Máster Universitario en Gestión y Análisis de Grandes Volúmenes de Datos: Big Data, tiene la certificación PCEP en Lenguaje de Programación Python y la certificación JSE en JavaScript. Además de las certificaciones SCJP Sun Certified Programmer for the Java 2 Platform Standard Edition, SCWD Sun Certified Web Component Developer for J2EE 5, SCBCD Sun Certified Business Component Developer for J2EE 5, SCEA Sun Certified Enterprise Architect for J2EE 5.

Desarrolladora de Aplicaciones FULLSTACK, se dedica desde hace + de 20 años a la CONSULTORÍA y FORMACIÓN en tecnologías del área de DESARROLLO y PROGRAMACIÓN.

training@iconotc.com

JAVA: Novedades y Buenas Prácticas versions LTS

❑ **Duración:** 20 horas

❑ **Modalidad:** Presencial / Remoto

❑ **Fechas/Horario:**

- Días 16, 17, 18 y 19 Junio 2025
- Horario de 15:00 a 20:00 hs

❑ **Contenidos:**

- Java 8
  - Expresiones lambda
  - Interfaz collection: forEach
  - Streams
  - Optional
  - Java Time
  - Interfaces genéricas de Java 8
  - Interfaces
  - Base 64
  - Nashorn
  - CompletableFuture
- Java 11
  - Mejoras de seguridad
  - Integración con la plataforma de contenedores
  - Closures
  - Pattern Matching
  - JVM Constants
  - java.net.http
  - Métodos Predeterminados en Interfaces
  - Mejoras en el Manejo de Cadenas String: isBlank(), lines(), stripLeading(), y stripTrailing()
  - Nuevos métodos en Optional: orElseThrow() y stream()
- Java 17
  - Inferencia de tipos mejorada
  - Rendimiento mejorado
  - Mejor soporte para Modern Java
  - Patrones de Switch: mejorados para dmitir casos múltiples
  - jdk.incubator.pattern: nuevo Módulo de Patrones
  - Mejoras en la API de E/S

Novedades en JAVA 8

Visión general de Lambdas

- Una interface funcional es una interfaz que proporciona un único método abstracto

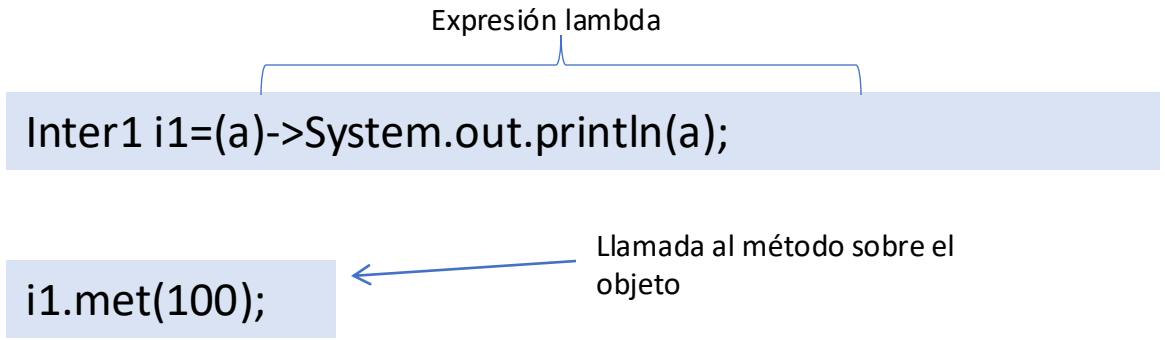
```
public interface Runnable{
   void run();
}
```

```
public interface Inter2{
   boolean process(int n, String pt);
   static void print(){}
}
```

```
public interface Inter1{
   void met(int data);
   default int res(){return 1;}
}
```

Visión general de Lambdas

- **Que es una expresión lambda?**
- Implementación de una interfaz funcional
- Proporciona el código del único método abstracto de la interfaz, a la vez que genera un objeto que implementa la misma



Visión general de Lambdas

- Una expresión lambda tiene dos partes, la lista de parámetros del método y la implementación:

```
parametros->implementación
```

- Los parámetros pueden indicar o no el tipo
- La lista de parámetros se puede indicar o no entre paréntesis (obligatorio si hay dos o más) y también si se indica el tipo
- En caso de devolver un resultado, la implementación puede omitir la palabra return si consta de una sola instrucción

Visión general de Lambdas

- Ejemplos

CORRECTO

```
()->3

(int a)->System.out.println("hello")

x->x*x

(n1,n2)->{
   n1+=20;
   System.out.println(n1+n2);
}
```

INCORRECTO

```
->3

int a->System.out.println("hello")

x->return x*x //se requieren llaves con return

n1,n2->System.out.println(n1+n2)
```

Sintaxis Local-Variable para Lambdas

- Es posible inferir el tipo en los parámetros de las expresiones lambda:

```
(var a)->System.out.println(a)
```

- Aunque no se puede combinar inferencia de tipos y tipos específicos en una misma expresión:

```
(var a, int c)->a+c  //error de compilación
```

- ¿Qué utilidad tiene si ya es posible no indicar el tipo en los parámetros?

```
(@NotNull var c)->...  //ok

(@NotNull c)->...  //error de compilación
```

Sintaxis Local-Variable para Lambdas

- **Comparator con Lambdas**
- Interfaz utilizada para la ordenación de colecciones y arrays
- Al ser funcional, se puede implementar con lambdas:

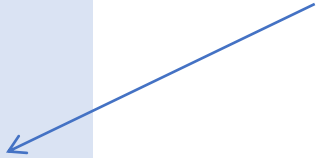
```
List<String> textos=new ArrayList<>();
textos.add("mi texto"); textos.add("hello");textos.add("es el más largo");
//ordenación de la lista de textos por longitud
textos.sort((a,b)->a.length()-b.length());
//recorrido y presentación de datos
for(String s:textos){
   System.out.println(s));
}                                                    //hello
                                                     //mi texto
                                                     //es el más largo
```

forEach

- void forEach(Consumer<? super T> action). Realiza una acción para cada elemento del stream.

```
Stream st=Stream.of(2,5,8,3,6,2,10);

//muestra todos los elementos
st.forEach(n->System.out.println(n));
System.out.println(st.count()); //Error!!
```



Tras llamar a un método el stream se cierra y **no puede** volver a utilizarse

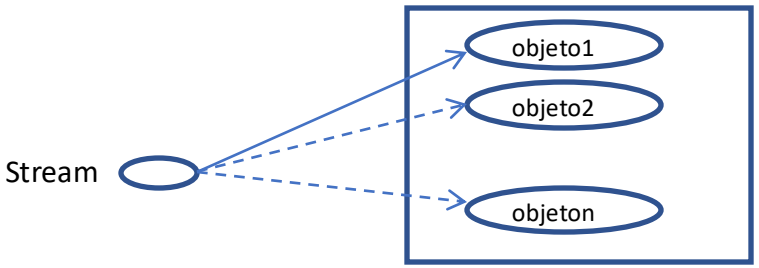
Streams

- **Que es un stream?**

- Objeto que permite realizar de forma rápida y sencilla operaciones de búsqueda, filtrado, recolección, etc. sobre un grupo de datos (array, colección o serie discreta de datos)
- Para manipular un Stream utilizamos la interfaz Stream de java.util.stream
- Otras variantes como IntStream, LongStream o DoubleStream se emplean para trabajar con tipos primitivos

Streams

- **Funcionamiento**
- Recorre los datos desde el principio hasta el final y durante el recorrido realiza algún tipo de cálculo u operación



- Una vez realizado el recorrido, el stream se cierra y no puede volver a utilizarse

Streams

- **Creación de un stream**
  - A partir de una colección:

```
ArrayList<Integer> nums=new ArrayList<>();
nums.add(20);nums.add(100);nums.add(8);
Stream<Integer> st=nums.stream();
```

  - A partir de un array:

```
String[] cads={“a”,”xy”,”jk”,”mv”};
Stream<String>st= Arrays.stream(cads);
```

  - A partir de una serie discreta de datos:

```
Stream<Double> st=Stream.of(2.4, 7.4, 9.1);
```

  - A partir de un rango de datos:

```
IntStream stint=IntStream.range(1,10);
IntStream stint2=IntStream.rangeClosed(1,10);
```

Streams

- **Tipos de métodos de Stream**

  - Métodos intermedios. El resultado de su ejecución es un nuevo Stream. Ejemplos: filtrado y transformación de datos, ordenación, etc.

  - Métodos finales. Generan un resultado. Pueden ser void o devolver un valor resultado de alguna operación. Ejemplos: calculo (suma, mayor, menor, …), búsquedas, reducción, etc.

Optional

- Encapsula resultados de una operación final de un Stream
- Podemos utilizar los siguientes métodos para manipularlo:
  - T get(). Devuelve el valor encapsulado. Si no hay ningún valor, lanza una NoSuchElementException
  - T orElse(T other). Devuelve el valor encapsulado. Si no hay ninguno, entonces devuelve el valor pasado como parámetro.
  - boolean isPresent(). Permite comprobar si contiene o no algún valor.
- Existen las variantes OptionalInt y OptionalDouble que encapsulan tipos primitivos

Problemas con java.util.Date

- La instancia es mutable – no compatible con Lambda.
- No es thread safe.
- Gestiona fechas y horas de forma conjunta.

Nuevo java.time

- La principal novedad es la separación de fechas y horas.
- toString() es entendible.
- Resuelve los problemas vistos en la diapositiva anterior.
- Implementan TemporalAccesor.

Nuevas clases

- LocalDate; Para manejar fechas
- LocalTime; Para tratar horas.
- LocalDateTime; Combinacion de las dos anteriores.

LocalDate

- Solo fechas, sin date ni zone information.
- No incluye horas
- Representa: año-mes-dia

- Métodos mas utilizados:
  - LocalDate.now(); Fecha de hoy
  - LocalDate.of(año,mes,dia); Crea la fecha con los datos facilitados
  - unaFecha.isBefore(otraFecha); Devuelve true si unaFecha es anterior a la otraFecha.
  - unaFecha.isLeapYear(); Devuelve true si es año bisiesto.
  - unaFecha.getDayOfWeek(); Devuelve el dia de la semana como un valor del enumerado DayOfWeek.
  - unaFecha.plusMonths(1); Suma un mes a la fecha.
  - unafecha.with(next(TUESDAY)); Devuelve la fecha del siguiente jueves.

LocalTime

- Basado en 24 horas. Las 13:30 es la 1:30 PM.

- Los métodos más utilizados son:
  - LocalTime.now(); Devuelve la hora actual
  - unaHora.plusHours(1).plusMinutes(15); Suma 1 hora y 15 minutos a unaHora.
  - unaHora.truncatedTo(ChronoUnit.MINUTES); Devuelve solo hora:minutos.
  - unaHora.toSecondOfDay(); Transforma a segundos el dia.
  - unaHora.isAfter(otraHora); Devuelve true si unaHora es posterior a otraHora.
  - unaHora.until(otraHora, ChronoUnit.MINUTES); Devuelve el tiempo que queda en minutos.

LocalDateTime

- Es una combinación de LocalDate y LocalTime.

- Los métodos mas utilizados son:
  - LocalDateTime.of(año,mes,dia,hora,min,seg,nanoseg); Crea el objeto con esos datos.
  - LocalDateTime.of(localDate, localTime);
  - obj.plusDays(4).plusHours(8);

Interfaces

- Es una colección de:
- declaraciones de métodos (sin definirlos)
- declaraciones de constantes.
- Las clases que implementen (implements) el interface han de definir obligatoriamente las funciones declaradas en él.
- Una clase puede implementar uno o más interfaces.

Interfaces

- Una interfaz se puede comparar a una clase abstracta que tiene todos sus métodos abstractos, ya que en una interfaz no puede existir ningún método con código.

- Una interfaz puede heredar de otra.

- Una clase puede implementar varias interfaces.

Interfaces

- Sintaxis para crearla:

```
public interface nombreInterfaz [extends interface1]
```

- Sintaxis para implementarla:

```
public class nombreClase implements interf1, interf2, ...
```

Ejemplo de interface

EL INTERFACE

```
import java.awt.Graphics;
public interface Geometria
{
    public double area();
    public void dibujar(Graphics g);
}
```

LAS CLASE QUE IMPLEMENTAN EL INTERFACE

```
public class Circulo extends Figura implements Geometria
{
    //...
}
```

```
public class Rectangulo extends Figura implements Geometria
{
    //..
}
```

Ejemplo de interface

```
import java.awt.Graphics;
public class Circulo extends Figura implements Geometria
{
    //VARIABLES
    private float radio;
    public final static double  PI = 3.14;
    //CONSTRUCTORES
    // ...
    //METODOS
    //....

    public double area(){
        double a;
        a = 2*PI* radio * radio;
        return a;
    }
    public  void dibujar(Graphics g){
        g.fillOval(this.getCoordX(),this.getCoordY(),(int)radio, (int)radio);
    }

}
```

Métodos en interfaces

- **Métodos estáticos**
  - Desde Java 8, las interfaces pueden incluir métodos estáticos al igual que las clases.
  - El método está asociado a la interfaz, no es heredado por las clases que la implementan.

```
interface InterA{
   static void m(){
      System.out.println("estático InterA");
   }
}
public class Test implements InterA{

}
```

```
public class Prueba{
   public static void main(String[] args){
      Test ts=new Test();
      ts.m(); //error de compilación
      Test.m(); //error de compilación
      InterA.m(); //correcto, muestra estático InterA
   }
}
```

Métodos en interfaces

- **Métodos default**
  - Proporciona una implementación por defecto, que puede ser utilizada por las clases que implementan la interfaz.
  - Se definen con la palabra reservada default:

```
public interface Operaciones{
   default void girar(int grados){
      System.out.println("gira "+grados+" grados);
   }
   int invertir();
}
:
public class Test implements Operaciones{
   //solo tiene que implementar el abstracto
   //aunque, si se quiere, se puede sobrescribir
   //también el default
   public int invertir(){
      :
   }
}
```

```
public class Prueba{
   public static void main(String[] args){
      Test ts=new Test();
      //utiliza la implementación por defecto
      ts.girar(30); //muestra gira 30 grados
   }
}
```

Base 64

- Introducción del soporte para la codificación Base64 en la biblioteca estándar de Java con el lanzamiento de Java 8.

- Esta característica facilita la codificación y decodificación de datos en formato Base64 de manera sencilla y eficiente.

- Java 8 incluye clases como Base64.Encoder y Base64.Decoder que permiten realizar estas operaciones de codificación y decodificación de manera fácil y efectiva.

Nashorn

- Nashorn es el nuevo motor de JavaScript integrado en Java 8, que reemplaza al anterior motor Rhino.
- Algunas de las principales novedades de Nashorn en Java 8 son:
  - Soporte completo de ECMAScript 5.1: Nashorn implementa completamente la especificación ECMAScript 5.1, lo que le permite ejecutar código JavaScript de manera más completa y compatible.
  - Mejor rendimiento: Nashorn utiliza la nueva característica InvokeDynamic introducida en Java 7, lo que le permite tener un mejor rendimiento en comparación con el antiguo motor Rhino.
  - Integración con Java: Nashorn permite una integración más fluida entre Java y JavaScript, permitiendo pasar objetos Java a JavaScript y viceversa de manera transparente.
  - Soporte para interfaces funcionales y características de Java 8: Al estar integrado en Java 8, Nashorn también soporta las nuevas características del lenguaje Java, como las expresiones lambda y las interfaces funcionales.
  - Consola interactiva: Nashorn viene con una consola interactiva que permite ejecutar código JavaScript directamente desde la línea de comandos, facilitando la experimentación y pruebas.

CompletableFuture

- CompletableFuture es una clase introducida en Java 8 que implementa la interfaz Future y la interfaz CompletionStage, ofreciendo alrededor de 50 métodos para simplificar el manejo y control de trabajos asíncronos.
- Esta clase permite combinar, componer y ejecutar tareas de forma asíncrona, incluyendo el manejo de errores.
- Para ejecutar procesos paralelos de manera eficiente, se recomienda utilizar un ExecutorService para crear un pool de threads independientes.
- Esto garantiza que los procesos no saturen el hilo principal de Java y se ejecuten de manera eficiente.
- CompletableFuture facilita la programación asíncrona en Java al ofrecer métodos para combinar llamadas, gestionar errores y ejecutar tareas de forma concurrente.

CompletableFuture

- Crear un CompletableFuture básico:

  CompletableFuture<String> completableFuture = new CompletableFuture<String>();

- Para completar manualmente el CompletableFuture, se usa el método complete():

  completableFuture.complete("finish").

- Crear CompletableFuture con runAsync:

  CompletableFuture<Void> voidCompletableFuture = CompletableFuture.runAsync(() -> { /* código asíncrono */ });

  Esto ejecuta una tarea de manera asíncrona sin esperar un resultado.

- Crear CompletableFuture con supplyAsync:

  CompletableFuture<String> supplier = CompletableFuture.supplyAsync(() -> { /* código asíncrono */ return "Value"; });

  Esto crea un CompletableFuture que ejecuta una tarea asíncrona y devuelve un resultado.

CompletableFuture

- Encadenar CompletableFutures: Usando métodos como thenCompose() y thenApply() se pueden encadenar múltiples CompletableFutures.
- Esto permite ejecutar tareas de manera asíncrona y componer los resultados.
- Manejar errores:Métodos como exceptionally() y handle() permiten capturar y manejar excepciones que ocurran durante la ejecución de las tareas asíncronas.
- Esperar por múltiples CompletableFutures:Métodos como allOf() y anyOf() permiten esperar por la finalización de múltiples CompletableFutures.

Novedades en JAVA 11

Mejoras de seguridad

- **Fundamentos**
  - Conjunto de buenas prácticas a la hora de conseguir generar código seguro, que evite efectos indeseados ante posibles ataques externos.
  - Se organizan en nueve secciones:

Mejoras de seguridad

- **Denegación de servicio**
  - Debemos comprobar la entrada de datos en un sistema para evitar que cause un consumo excesivo de recursos y que pueda afectar a la CPU o la memoria
  - Liberar recursos en todas las situaciones
  - Definir un sistema robusto de gestión de errores y excepciones

```
try(resource1;resource2){


}
catch…
```

Mejoras de seguridad

- **Confidencialidad**
  - Los datos confidenciales solo deberían ser visibles dentro de un contexto limitado
  - Eliminar información sensible de volcados de error
  - No realizar registros de log de información sensible
  - Considerar eliminar información sensible de la memoria después de su uso

Mejoras de seguridad

- **Inyección**

  - Evitar la inyección de SQL mediante el uso de PreparedStatement. Utilizar parámetros en la instrucción SQL , en lugar de concatenar valores:

Incorrecto

```
String sql="select * from alumnos where nombre=`"+nombre+"´";
Statement st=con.createStatement();
st.execute(sql);
```

Correcto

```
String sql="select * from alumnos where nombre=?";
PreparedStatement st=con.prepareStatement(sql);
st.setParameter(1,nombre);
st.execute();
```

  - Evitar inyección de valores excepcionales en punto flotante:

```
if (Double.isNaN(untrusted_double_value)) {
    // specific action for non-number case
}

if (Double.isInfinite(untrusted_double_value)){
    // specific action for infinite case
}

// normal processing starts here
```

Mejoras de seguridad

- **Accesibilidad**
  - Limitar la accesibilidad de clases, métodos y atributos al mínimo requerido por nuestro código
  - Limitar la extensibilidad de clases y métodos. Para evitar herencias y sobrescrituras maliciosas, debemos definirlos como final. Si una clase debe usar otra, preferible composición a herencia
  - Evitar cambios en una superclase para que las subclases no se vean afectadas

Mejoras de seguridad

- **Validación de datos**
  - Validar siempre datos de entrada, a fin de evitar que puedan alterar el flujo de nuestro código o corromper el estado de objetos:

```
public void process(String name, int value){
    if(name.equals("")){…}

    if(value<0&&value>100){…}
}
```

  - No se debe confiar solo en la validación del lado cliente
  - Definir envoltorios alrededor de métodos nativos

Mejoras de seguridad

- **Mutabilidad**
- Crear copias de valores de salida mutables. Cuando un método devuelve un dato que es mutable, preferible devolver una copia:

```
public class CopyOutput {
    private final java.util.Date date;
    public java.util.Date getDate() {
        return (java.util.Date)date.clone();
    }
}
```

- No exponer colecciones modificables
- Definir atributos públicos estáticos como finales

Mejoras de seguridad

- **Construcción de objetos**
  - Evitar constructores públicos de clases sensibles
  - No establecer valores de atributos en el constructor, hasta que todas las comprobaciones se hayan realizado
  - Evitar desde los constructores llamadas a métodos que pueden ser sobrescritos

```
public class Test{
    private int data;
    public Test(int x){
        if(x>10){…}
        method();//llamada no segura
    }
    public void method(){..}
}
```

validación de parámetros antes de asignación

Mejoras de seguridad

- **Serialización**
  - Evitar la serialización de clases sensibles
  - Proteger datos sensibles durante la serialización
  - Evitar serializar determinados datos durante el proceso de serialización, definiéndolos como transient:

```
public class Profile implements Serializable {
    private transient String password;
}
```

  - Ser consciente de que, durante el proceso de deserialización, se crea un objeto de la clase sin invocar a constructor alguno

Mejoras de seguridad

- **Control de acceso**
  - Invocaciones seguras mediante doPrivileged:

```
public class LibClass {

    private static final String OPTIONS = "xx.lib.options";

    public static String getOptions() {
        return AccessController.doPrivileged(
            new PrivilegedAction<String>() {
                public String run() {
                    // this is checked by SecurityManager
                    return System.getProperty(OPTIONS);
                }
            }
        );
    }
}
```

El valor debe estar "controlado", no permitir cualquier entrada

Las llamadas a propiedades de sistema se realizan desde doPrivileged para que, quien utilice la clase acceda con los permisos de la misma

Integración con la plataforma de contenedores

- La plataforma de contenedores en Java 11 se utiliza para crear y ejecutar contenedores de manera eficiente y escalable.
- Estos contenedores permiten a las aplicaciones Java ejecutarse de manera aislada y portátil, lo que facilita su desarrollo, pruebas y despliegue en diferentes entornos.

Integración con la plataforma de contenedores

- Uso de contenedores en Java 11:
  - **Docker**: Java 11 admite el uso de contenedores Docker, lo que permite a las aplicaciones Java ejecutarse en entornos aislados y portátiles. Docker proporciona una forma de empaquetar la aplicación y sus dependencias en un contenedor que puede ser ejecutado en cualquier máquina que tenga Docker instalado.
  - **Testcontainers**: Testcontainers es una biblioteca que facilita la creación de contenedores para pruebas de integración. Permite a los desarrolladores crear contenedores para pruebas de integración de manera eficiente y escalable.
  - **Java EE**: La plataforma Java EE (Enterprise Edition) admite el uso de contenedores para ejecutar aplicaciones Java. Esto permite a las aplicaciones Java EE ejecutarse en entornos aislados y portátiles, lo que facilita su desarrollo, pruebas y despliegueen diferentes entornos.
  - **Swing**: La biblioteca Swing de Java proporciona contenedores y componentes gráficos de usuario para crear interfaces de usuario. Estos contenedores y componentes permiten a los desarrolladores crear interfaces de usuario interactivas y funcionales.

Integración con la plataforma de contenedores

- Ventajas de utilizar contenedores en Java 11:
  - **Portabilidad**: Los contenedores permiten a las aplicaciones Java ejecutarse en diferentes entornos sin necesidad de cambios en el código.
  - **Aislamiento**: Los contenedores aíslan la aplicación de los demás procesos en el sistema, lo que mejora la seguridad y la estabilidad.
  - **Escalabilidad**: Los contenedores permiten a las aplicaciones Java ejecutarse en diferentes máquinas y entornos, lo que facilita la escalabilidad.
  - **Flexibilidad**: Los contenedores permiten a los desarrolladores elegir el entorno de ejecución de la aplicación, lo que facilita la adaptación a diferentes entornos.

Integración con la plataforma de contenedores

- Diferencias entre Java 8 y Java 11

  - Nashorn: Java 8 incluye el motor de JavaScript Nashorn, que permite ejecutar código JavaScript en la plataforma Java. Nashorn es una mejora significativa en comparación con Rhino, el motor de JavaScript anterior. Nashorn tiene un soporte total de la especificación ECMAScript 5.1 y compila JavaScript a bytecode utilizando las nuevas características de la plataforma, incluyendo invokedynamic.

  - Java 11: **Java 11 elimina el motor de Nashorn**, que servía para ejecutar JavaScript. En su lugar, Java 11 incluye el estándar HTTPClient, que se encontraba disponible desde la versión 9 pero en fase de pruebas. Esto indica que Java 11 se enfoca más en la programación de aplicaciones web y en la gestión de peticiones HTTP.

  - Contenedores: Java 8 y Java 11 comparten la capacidad de ejecutar contenedores, pero Java 11 tiene una mayor integración con contenedores Docker. Java 11 admite el uso de contenedores Docker, lo que permite a las aplicaciones Java ejecutarse en entornos aislados y portátiles.

  - Swing: Java 8 y Java 11 comparten la biblioteca Swing para crear interfaces gráficas de usuario. Swing es una biblioteca de dases que proporciona contenedores y componentes gráficos de usuario para crear interfaces de usuario interactivas y funcionales.

  - Java EE: Java 8 y Java 11 comparten la plataforma Java EE, que admite el uso de contenedores para ejecutar aplicaciones Java. Esto permite a las aplicaciones Java EE ejecutarse en entornos aislados y portátiles.

Closures

- Los closures son un concepto de programación que se encuentra en diferentes lenguajes de programación, como Java, Groovy, Scala, Python, entre otros.
- Un closure es una función que hereda el contexto de otra función, lo que le permite acceder a las variables utilizadas en la función externa.
- En Java 11, si bien no hay una implementación nativa de closures como en otros lenguajes, se pueden simular utilizando clases anónimas.
- Esto permite pasar un bloque de código como argumento a una función, lo que facilita la implementación de consultas y operaciones sobre colecciones.

Pattern Matching

- El Pattern Matching en Java 11 se introdujo a través de dos JEPs (JEP 305 y JEP 394):
- Pattern Matching para el operador instanceof:
  - Permite realizar pruebas de tipo y extraer datos de objetos de manera más concisa y segura.
  - Elimina la necesidad de realizar conversiones explícitas después de las comprobaciones de tipo.
  - Permite utilizar variables de patrón en las expresiones de los condicionales.
- Pattern Matching para las expresiones y sentencias switch:
  - Permite realizar coincidencias de patrones más complejas, como coincidir con campos de registros o rangos de valores.
  - Introduce nuevas sintaxis como los "patrones guardados" y los "patrones entre paréntesis" para refinar las condiciones de coincidencia.
  - Hace que el código sea más legible y preciso al poder coincidir con valores reales, en lugar de solo tipos.

Pattern Matching

- Pattern Matching para el operador instanceof:

```
public class PatternMatchingInstanceOf {
    public static void main(String[] args) {
        Object obj = "Hola, Mundo";

        if (obj instanceof String str) {
            System.out.println("La variable 'obj' es una instancia de String");
            System.out.println("El contenido de la cadena es: " + str);
        } else {
            System.out.println("La variable 'obj' no es una instancia de String");
        }
    }
}
```

Pattern Matching

- Pattern Matching para las expresiones y sentencias switch:

```
public interface PatterMatchingSwitch {

    public static void main(String[] args) {

        int monthNumber = 3;
        int quarter = switch (monthNumber) {
            case 1, 2, 3 -> 1;
            case 4, 5, 6 -> 2;
            case 7, 8, 9 -> 3;
            case 10, 11, 12 -> 4;
            default -> throw new IllegalArgumentException("Invalid month: " + monthNumber);
        };
        System.out.println("Quarter: " + quarter);
    }

}
```

JVM Constants

- Los JVM Constants en Java 11 representan una mejora en la forma de manejar y representar los valores constantes en las clases Java, lo que se traduce en una mayor eficiencia, rendimiento y flexibilidad para los desarrolladores.
- En Java 11, podemos utilizar la API de constantes de la JVM para acceder a estas constantes de manera más eficiente:

```
import java.lang.constant.ClassDesc;
import java.lang.constant.ConstantDesc;
import java.lang.constant.ConstantDescs;
import java.lang.constant.DirectMethodHandleDesc;
import java.lang.constant.DynamicConstantDesc;
import java.lang.constant.MethodTypeDesc;
```

HTTP

- Mejorado en JDK 11
- Esta API existe en Java desde versiones antiguas pero presentaba los siguientes inconvenientes:
  - La API de URLConnection se diseñó con varios protocolos que ahora ya no funcionan (FTP, gopher, etc.).
  - La API es anterior a HTTP/1.1 y es demasiado abstracta.
  - Funciona solo en modo de bloqueo (es decir, un hilo por solicitud/respuesta).
  - Es muy difícil de mantener.

HTTP

- Las nuevas API HTTP se pueden encontrar en java.net.HTTP.*
- La API consta de tres clases principales:

- HttpRequest representa la solicitud que se enviará a través de HttpClient.
- HttpClient se comporta como un contenedor de información de configuración común a varias solicitudes.
- HttpResponse representa el resultado de una llamada HttpRequest.

Uso y características

- La versión más nueva del protocolo HTTP está diseñada para mejorar el rendimiento general del envío de solicitudes por parte de un cliente y la recepción de respuestas del servidor.
- A partir de Java 11, la API ahora es completamente asíncrona (la implementación anterior de HTTP/1.1 estaba bloqueada).
- Las llamadas asíncronas se implementan utilizando CompletableFuture.
- La implementación de CompletableFuture se encarga de aplicar cada etapa una vez finalizada la anterior, por lo que todo este flujo es asíncrono.
- La nueva API de cliente HTTP proporciona una forma estándar de realizar operaciones de red HTTP con soporte para funciones web modernas como HTTP/2, sin necesidad de agregar dependencias de terceros.
- Las nuevas API brindan soporte nativo para HTTP 1.1/2 WebSocket.

Métodos Predeterminados en Interfaces

- Los métodos predeterminados en interfaces en Java 11 son una característica que permite agregar métodos con implementaciones predeterminadas en las interfaces, lo que facilita la extensión de la funcionalidad de las interfaces sin romper la compatibilidad con las implementaciones existentes.

```
interface Calculadora {
    default int sumar(int a, int b) {
        return a + b;
    }
}

class CalculadoraImplementacion implements Calculadora {
    public static void main(String[] args) {
        CalculadoraImplementacion calculadora = new CalculadoraImplementacion(
        int resultado = calculadora.sumar(5, 3);
        System.out.println("El resultado de la suma es: " + resultado);
    }
}
```

String

- Nuevos métodos en Java 11:
- boolean isBlank(). Devuelve true si la cadena está vacía o contiene solamente espacios en blanco:
- String repeat(int n). Devuelve una cadena resultado de concatenar tantas veces la cadena actual como se indique en el parámetro
- String strip(). Devuelve una cadena resultante de eliminar espacios a izquierda y derecha. Similar a trim(), pero reconoce más caracteres en blanco

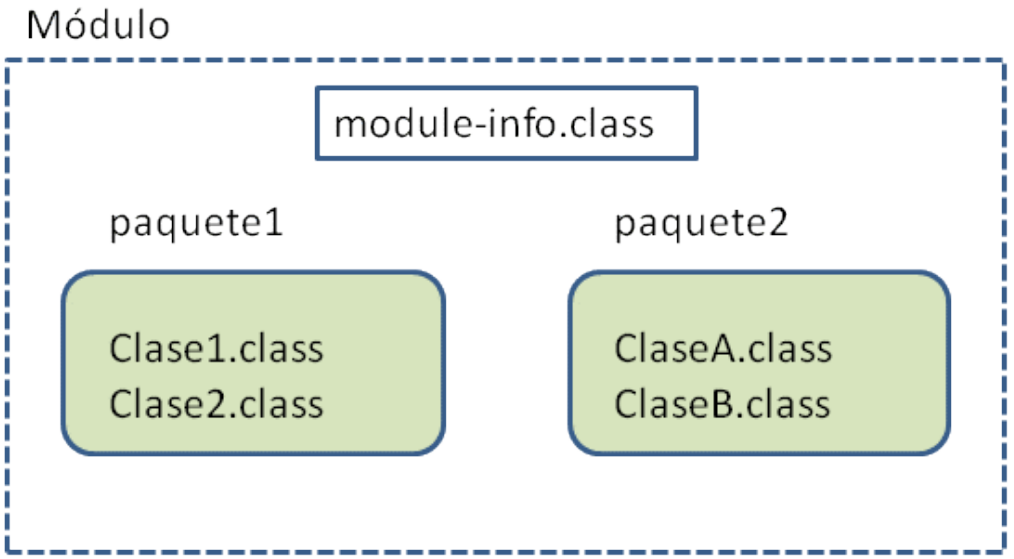
Nuevos métodos en Optional

- El método **orElseThrow()** es similar a orElse(), pero en lugar de devolver un valor predeterminado si el valor opcional no está presente, lanza una excepción si el valor opcional no está presente. Esto puede ser útil cuando se necesita manejar errores de manera explícita en lugar de devolver un valor predeterminado.

- El método **stream()** devuelve un flujo que contiene solo el valor presente en el Optional, o un flujo vacío si el valor no está presente. Esto permite transformar un flujo de valores opcionales en un flujo de valores presentes, lo que puede ser útil para procesar datos de manera más eficiente.

Introducción a los módulos Java 9

- **Que es un módulo?**
- Nivel de división superior al de paquete
- Agrupa un conjunto de paquetes e incluye información de dependencia de los mismos



El propio JDK está organizado de forma modular

Introducción a los módulos Java 9

- **Ventajas**

- Mejor control de acceso. Permite que sólo ciertos paquetes sean utilizados por otras aplicaciones.

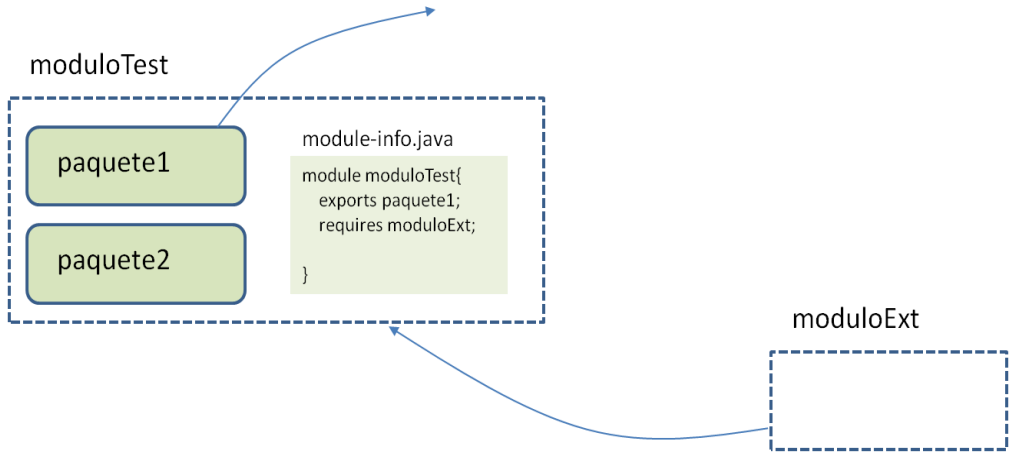
- Claridad en las dependencias. A través de module-info, se especifica claramente las dependencias entre módulos, que son evaluadas al compilar y al lanzar la aplicación.

- Paquetes de distribución más pequeños. Facilita la distribución de aplicaciones y mejora el rendimiento.

- Existencia de paquetes únicos. No puede haber dos módulos que expongan el mismo paquete.

Descriptores, requires, exports de módulos

- **Descriptor de módulo**
- Se trata del archivo module-info.java
- Debe estar en el directorio raíz del módulo
- Indica los módulos requeridos por nuestro módulo y los paquetes a exportar para otros módulos



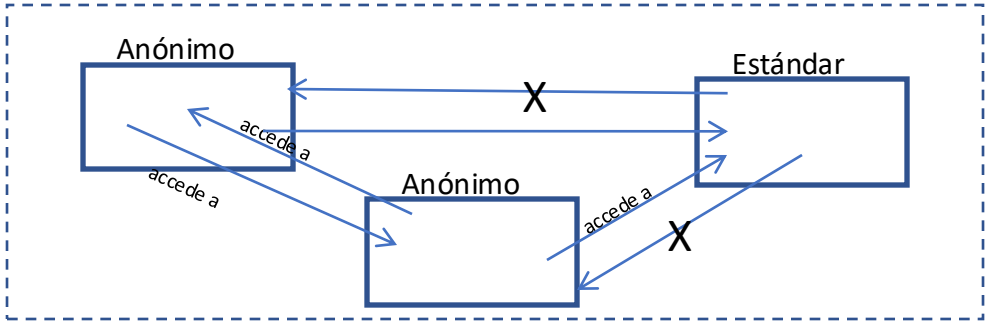
Tipos de módulos

- **Módulos anónimos**

- Conjunto de paquetes de clases de una aplicación que no forman parte de un módulo.
  Habitualmente, se distribuyen en un .jar

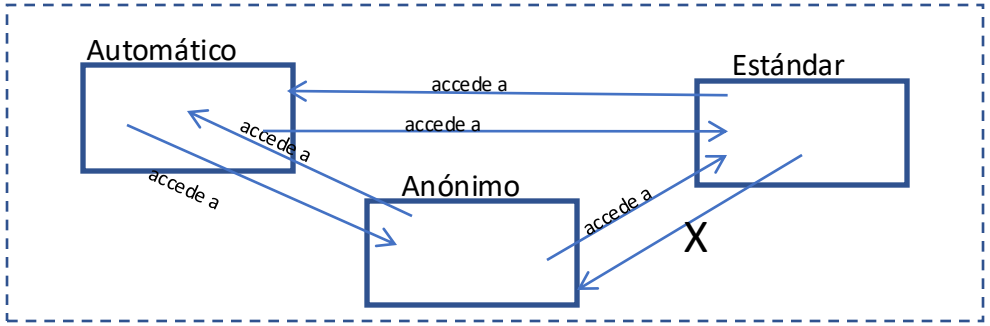
- Desde estas clases, se puede acceder a cualquier paquete de clases que se encuentre en el classpath.
  En el caso de paquetes modularizados, a exportados y no exportados

- Solo pueden acceder a las clases de un módulo anónimo las clases de otros módulos anónimos (o automáticos)



Tipos de módulos

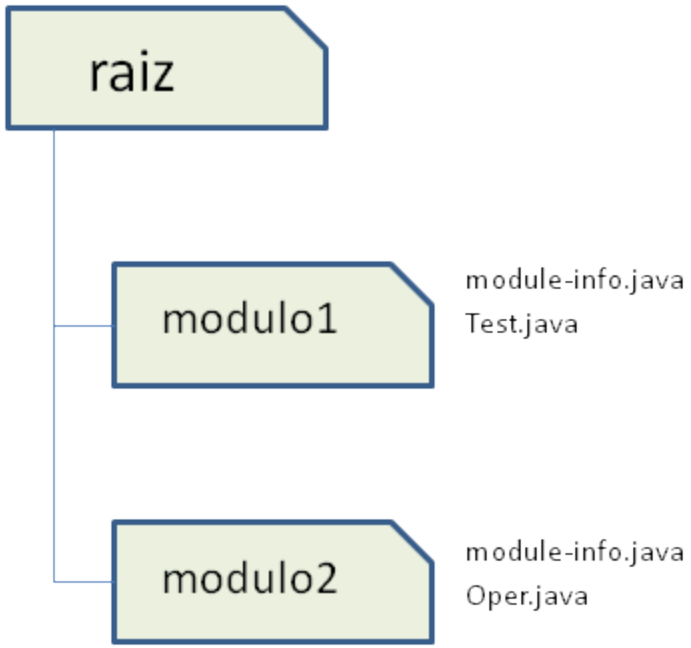
- **Módulos automáticos**
- Cuando un módulo anónimo se incluye en el module-path de una aplicación, se convierte en un módulo automático
- Desde estas clases, se puede acceder a cualquier paquete de clases, tanto de módulos anónimos/automáticos como de estándares.
- Exportan implícitamente todas sus clases, que podrán ser utilizadas por otros módulos que lo requieran



Crear y utilizar módulos

- **Estructura de ejemplo**

- Partimos de la siguiente estructura de módulos y clases de ejemplo:



Crear y utilizar módulos

- **Modulo 2**

- Contiene una clase que va a ser utilizada desde otro módulo (modulo1)

Oper.java

```
package com.operaciones;
public class Oper{
   public int sumar(int a, int b){
      return a+b;
   }
   public int multiplicar(int a, int b){
      return a*b;
   }
}
```

module-info.java

```
module  modulo2{
   exports com.operaciones;
}
```

Crear y utilizar módulos

- **Modulo 1**

- Incluye una clase que hace uso del paquete expuesto por el módulo2

Test.java

```
package com.cliente;
import com.operaciones.Oper;
public class Test{
   public static void main(String[] args){
      Oper op=new Oper();
      System.out.println(op.sumar(2,9));
      System.out.println(op.multiplicar(2,9));
   }
}
```

module-info.java

```
module  modulo1{
   requires modulo2;
}
```

Novedades en JAVA 17

Inferencia de tipos

- Característica incorporada en Java 10, consistente en declarar variables locales sin indicar explícitamente el tipo.
- Se emplea la palabra var:

```
var num=100; //entero
var datos=new ArrayList<Integer>(); //ArrayList de enteros
```

- El tipo es inferido por el compilador a partir del valor asignado a la variable
- Simplifica la escritura de código, ni mejora ni empeora el rendimiento de la aplicación

Inferencia de tipos Local-Variable

- Únicamente puede utilizarse con variables locales:

```
class Test{
   var prueba=100; //error de compilación
   void print(){
      var res="success"; //correcto
   }
}
```

- Es obligatorio asignar explícitamente un valor a la variable, valor que no puede ser null:

```
var data; //error de compilación
var n=null; //error de compilación
```

Inferencia de tipos Local-Variable

- No es posible utilizar inferencia de tipos en declaraciones múltiples:

```
var a,c=10; //incorrecto
var b=5,x=30; //incorrecto
```

- Se puede utilizar inferencia de tipos en bucles de tipo for:

```
for(var i=0;i<10;i++){

}
```

```
for(var s:datos){

}
```

- En arrays, no puede utilizarse con inicialización abreviada:

```
var s={5,9,10}; //incorrecto
var d=new int[]{5,1,3}; //correcto
```

Rendimiento mejorado

- Algunas de las áreas en las que se ha trabajado para mejorar el rendimiento en Java 17 incluyen:
    - Optimizaciones en la JVM: Java 17 ha introducido mejoras en la Máquina Virtual de Java (JVM) para optimizar la ejecución de código Java, lo que puede resultar en una mayor eficiencia y velocidad de ejecución de las aplicaciones.
    - Mejoras en la recolección de basura: Se han realizado ajustes en el recolector de basura de Java para mejorar la gestión de la memoria y reducir el impacto de la recolección de basura en el rendimiento de las aplicaciones.
    - Optimizaciones en la compilación Just-In-Time (JIT): Java 17 puede incluir mejoras en el compilador JIT para generar un código más eficiente y optimizado, lo que puede acelerar la ejecución de las aplicaciones Java.
    - Mejoras en la gestión de recursos: Se han implementado mejoras en la gestión de recursos y en la eficiencia de las operaciones de E/S, lo que puede contribuir a un mejor rendimiento general de las aplicaciones Java.

Patrones en Switch: mejorados para admitir casos múltiples.

- En Java 17, esta estructura se ha mejorado con expresiones switch, lo que permite que la estructura switch pueda retornar valores y utilizar patrones en los casos "Pattern Matching for Switch”.

- En las versiones anteriores solo tenían la sentencia switch, que evaluaba un dato pero no retornaba un valor. Otra mejora importante es la introducción de las expresiones switch, que permiten que la estructura switch pueda retornar valores, a diferencia de las sentencias switch tradicionales que solo evaluaban un dato.

- En versiones anteriores, el switch lanzaba un NullPointerException cuando el valor era nulo. Sin embargo, en Java 17, tanto el switch statement como la expresión switch permiten el uso de null como parte del patrón

Patrones en Switch: mejorados para admitir casos múltiples.

```
switch (value) {
    case String s && (s.length() > 3) -> System.out.println("A short string");
    case String s && (s.length() > 10) -> System.out.println("A medium string"
    default -> System.out.println("A long string");
}
```

Pattern Matching para el operador instanceof

- Con esta novedad podemos realizar un casting automático con el operador instanceof

```
Object objeto = 5;

if (objeto instanceof Integer i) {
    System.out.println(i + " Es un numero");
}
```

Pattern Matching en switch

- Java nos permite hacer el casting automático de una variable al usar las palabras reservadas del lenguaje *instance of* en un *if/switch.*

```
public static void main(String[] args) {
    Object o = "test"; // any object
    String formatter = switch(o) {
        case Integer i -> String.format("int %d", i);
        case Long l    -> String.format("long %d", l);
        case Double d  -> String.format("double %f", d);
        case Object o  -> String.format("Object %s", o.toString());
    }
```

Mejoras en la API de E/S

- Java 17 ha introducido cambios significativos en el API java.io que impactan su funcionalidad y seguridad:
    - Clases y Métodos Sellados: En Java 17, se ha implementado la funcionalidad de clases e interfaces selladas en el API java.io. Esto implica restricciones sobre qué otras clases o interfaces pueden extender o implementar estas clases selladas, lo que contribuye a una mayor seguridad y control en el desarrollo de aplicaciones que utilizan este API.
    - Eliminación del Compilador Experimental de Compilación: Otra novedad importante en Java 17 es la eliminación del compilador experimental de compilación en el API java.io. Esta acción refleja la constante evolución y mejora del lenguaje Java, eliminando elementos que no han demostrado ser eficaces o necesarios en el desarrollo de software.

Mejoras en la API de E/S

- Clases y Métodos Sellados

```java
// Clase sellada en Java 17
public sealed class Animal permits Dog, Cat {
    // Implementación de la clase
}

// Clase que extiende la clase sellada Animal
public final class Dog extends Animal {
    // Implementación de la clase Dog
}

// Clase que extiende la clase sellada Animal
public final class Cat extends Animal {
    // Implementación de la clase Cat
}
```

Mejoras en la API de E/S

- Eliminación del Compilador Experimental de Compilación

```java
// Código de ejemplo que utilizaba el compilador experimental de compilación
public class Main {
    public static void main(String[] args) {
        var list = new ArrayList<String>();
        list.add("Ejemplo");
        System.out.println(list.get(0));
    }
}
```

Mejoras en la API de E/S

- Las principales funciones de la API java.io en Java 17 son:
  - Lectura y escritura de archivos: Clases como FileReader, BufferedReader, FileWriter y BufferedWriter permiten leer y escribir archivos de manera eficiente
  - Manejo de flujos de entrada/salida: Interfaces como InputStream y OutputStream proporcionan una forma estándar de trabajar con flujos de datos de entrada y salida
  - Serialización de objetos: Clases como ObjectInputStream y ObjectOutputStream facilitan la serialización y deserialización de objetos Java para su almacenamiento o transmisión
  - Manejo de sockets: Clases como Socket y ServerSocket permiten la comunicación a través de sockets TCP/IP para aplicaciones cliente-servidor
  - Compresión y descompresión de datos: Clases como GZIPInputStream y GZIPOutputStream brindan soporte para la compresión y descompresión de datos utilizando el formato GZIP

Icono *Training Consulting*

JAVA: Novedades y Buenas Practicas versiones LTS

Completa nuestra encuesta de satisfacción a través del QR

GRACIAS

2025