FORMACIÓN EN NUEVAS TECNOLOGÍAS
JAVA 9-11: Novedades y buenas prácticas

1. Introducción

Nuevo ciclo de reléase de Java

Visión general de Java 9

Visión general de Java 10

Visión general de Java 11

1.1. Nuevo ciclo de release de Java

1.2. Visión general de Java 9

Novedades

- Modulos
- Métodos privados en interfaces
- Anotacion @deprecated mejorada
- Métodos de fábrica de conveniencia para colecciones
- Mejoras try-with-resources
- Nueva API HTTP
- Formato TIFF para Image I/O
- Logging API

1.3. Visión general de Java 10

Novedades:

- Inferencia de tipos
- Mejoras en Garbage Collection
- Nuevas extensiones de etiquetas de idioma Unicode

1.4. Visión general de Java 11

Novedades

- Ejecuciones sin compilar con javac
- Nuevos métodos en String
- Variables Locale para los parámetros Lambda
- Cliente HTTP
- Nuevos métodos para lectura y escritura de archivos
- Acceso a clases internas

2. Trabajar con módulos en Java 9

- ClassPath Shortcomings
- Introducción a los módulos Java 9
- Descriptores, requires, exports de módulos
- Tipos de módulos
- Crear y utilizar módulos
- Migración – Trabajar con Jar y Classpath
- Servicios

2.1. ClassPath Shortcomings

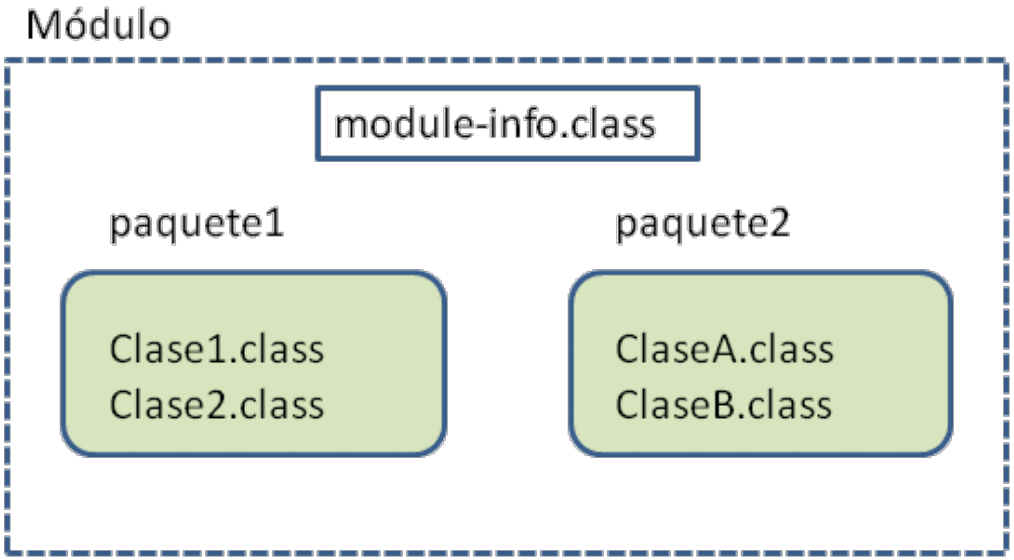
- Deficiencias con el classpath
  - Me permite acceder a recursos de la propia aplicacion

2.2. 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



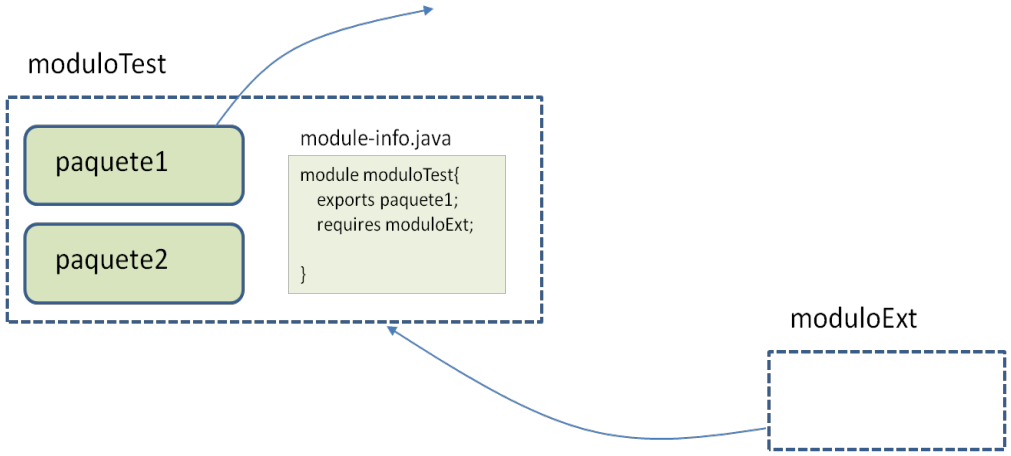
El propio JDK está organizado de forma modular

2.2. 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.

2.3. 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



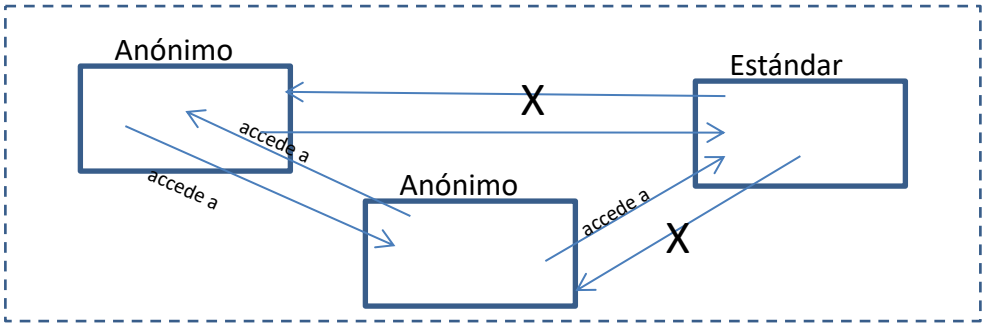
2.4. Tipos de módulos

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

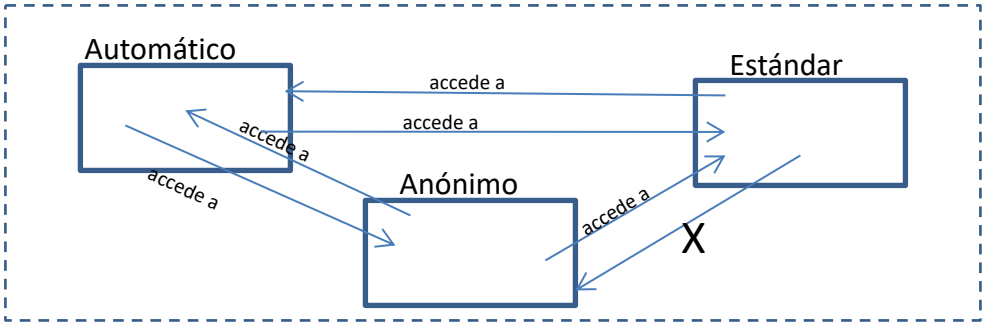
Solo pueden acceder a las clases de un módulo anónimo las clases de otros módulos anónimos (o automáticos)



2.4. Tipos de módulos

Módulos automáticos

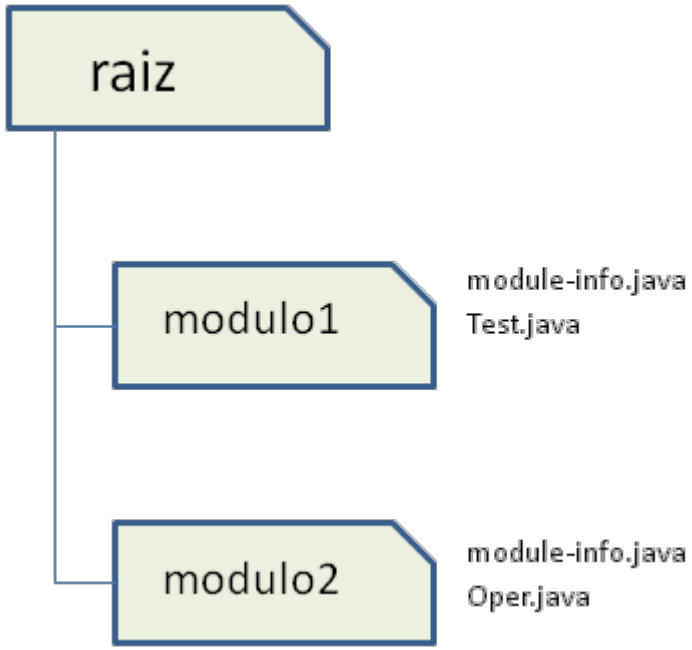
- Cuando un módulo anónimo se incluye en el module-path de una aplicación, se convierte en un módulo automático
- Desde estas clases, se puede acceder a cualquier paquete de clases, tanto de módulos anónimos/automáticos como de estándares.
- Exportan implícitamente todas sus clases, que podrán ser utilizadas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



2.5. Crear y utilizar módulos

Estructura de ejemplo

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



2.5. 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;
}
```

2.5. 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;
}
```

2.6. Migración – Trabajar con Jar y Classpath

Empaquetado en archivos .jar

Para empaquetar un módulo en un .jar debemos hacer uso del comando:

jar -c –file=*dir_destino*/*nombre_archivo*.jar –C *path_modulo* .

dir_destino es el directorio de destino del módulo path_modulo el directorio raíz del módulo. El punto "." indica que se incluya todo el contenido del directorio:

jar -c --file=ejecutables/modulo2.jar -C modulo2 .

jar -c --file=ejecutables/modulo1.jar -C modulo1 .

El directorio ejecutables debe existir

Para ejecutar modulo1 desde ejecutables:

java -p . -m modulo1/com.cliente.Test

2.6. Migración – Trabajar con Jar y Classpath

- **Empaquetado en archivos .jmod**

- Similar a .jar, si bien debe ser utilizado para librerías nativas en lugar de módulos en general:

- Sintaxis:

  jmod create --class-path *dir_modulo fichero*.jmod

- Ejemplo:

  jmod create –-class-path modulo2 ejecutables/modulo2.jmod

  jmod create –-class-path modulo1 ejecutables/modulo1.jmod

- No se puede utilizar el formato .jmod en la ejecución de módulos

2.6. Migración – Trabajar con Jar y Classpath

Comando jdeps

Se emplea para obtener información sobre la dependencia de módulos:

jdeps -s *dir_modulo/nombre_modulo*

Si depende de algún módulo extra:

jdeps --module-path *mod_dependiente* -s *dir_modulo/nombre_modulo*

Ejemplo:

jdeps --module-path ejecutables/modulo2.jar -s ejecutables/modulo1.jar

Resulta:

modulo1 -> java.base
modulo1 -> modulo2

2.7. Servicios

Servicio: Interfaz definida en un módulo :

```
module service{
   exports com.Interfaz1;
}
```

Proveedor de servicio: Módulo que implementa la interfaz:

```
module proveedor{
   provides com.Interfaz1 with com.Clase1;
}
```

Consumidor: Módulo que utiliza el servicio

```
module consumidor{
   uses com.Interfaz1;
}
```

3. JShell – un Java REPL

- Introducción a JShell
- Evaluar código (Snippets)
- Utilizar bibliotecas (Módulos, Jars, etc)
- Otras capacidades de jShell (comandos, scripts, etc)
- Inferencia de tipos
- Inferencia de tipos Local-Variable
- Visión general de Lambdas
- Sintaxis Local-Variable para Lambdas

3.1. Introducción a JShell

DURACIÓN

15 horas

3.2. Evaluar código (Snippets)

DURACIÓN

15 horas

3.3. Utilizar bibliotecas (Módulos, Jars, etc)

DURACIÓN

15 horas

3.4. Otras capacidades de jShell (comandos, scripts, etc)

DURACIÓN

15 horas

3.5. 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

3.6. 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
```

3.6. 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
```

3.7. Visión general de Lambdas

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

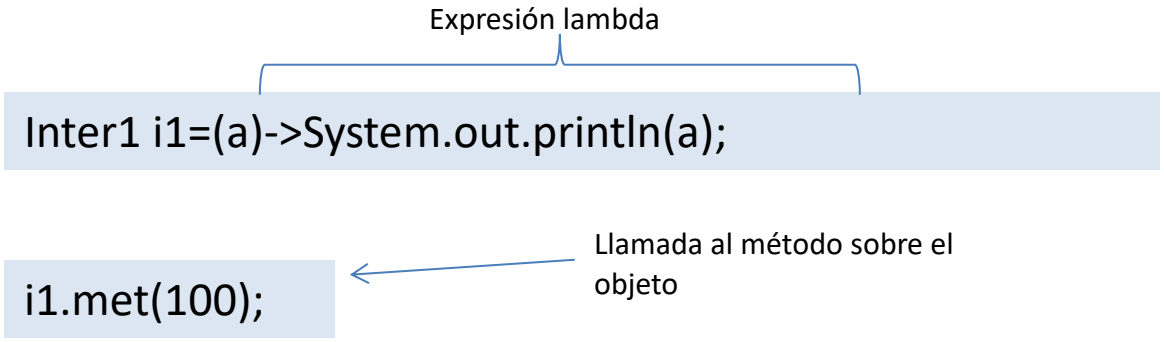
```
public interface Runnable{
   void run();
}
```

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

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

3.7. 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



3.7. 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

3.7. 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)
```

3.8. 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
```

3.8. 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
```

4. HttpClient

- Visión general
- API
- Uso y características

4.1. Visión general

- 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.

4.2. API

- 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.

4.3. 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.

5. Cambios / Adiciones en la API

- Colecciones
- String
- Streams
- Reactive Streams
- Flow API
- Métodos privados en interfaces
- Logging
- Módulos
- Optional
- Otros cambios de la API
- Características marcadas para eliminar (deprecated) / eliminadas

5.1. Colecciones

- Una colección es una agrupación de objetos sin tamaño fijo
- Se puede añadir y eliminar objetos de una colección dinámicamente
- Para gestionar colecciones disponemos de clases e interfaces específicas en java.util
- Tipos:
  - Listas
  - Tablas
  - Conjuntos

5.1. Colecciones

- **Listas**
- Cada elemento tiene una posición asociada a partir del orden de llegada, siendo 0 la posición del primero
- Las listas implementan la interfaz List, que a su vez implementa Collection.
- Son colecciones de tipo genérico (preparadas para admitir cualquier objeto Java)
- La principal clase de colección es ArrayList.

5.1. Colecciones

- **Conjuntos**
- Los elementos no tienen posición ni clave asociada, si bien cada elemento es único, no se pueden repetir
- Emplea internamente los métodos equals y hashcode para determinar la igualdad de objetos
- Los conjuntos implementan la interfaz Set, que es de tipo genérico
- La principal clase de conjuntos es HashSet

5.1. Colecciones

- **Mapas**

- Cada elemento tiene asociada una clave única
- No hay un orden o posición
- Las tablas implementan la interfaz Map.
- Tanto el tipo de la clave como del valor son genéricos
- La principal clase de colección es HashMap.

5.2. 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

5.3. 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)

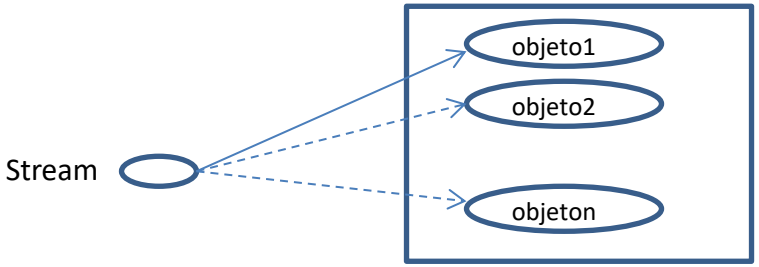
- Para manipular un Stream utilizamos la interfaz Stream de java.util.stream

- Otras variantes como IntStream, LongStream o DoubleStream se emplean para trabajar con tipos primitivos

5.3. 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

5.3. 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);
```

Stream de tipos primitivos

5.3. 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.

5.4. Reactive Streams

La programación reactiva es un paradigma enfocado en el trabajo con flujos de datos finitos o infinitos de manera asíncrona.

La motivación detrás de este nuevo paradigma procede de la necesidad de responder a las limitaciones de escalado presentes en los modelos de desarrollo actuales, que se caracterizan por su desaprovechamiento del uso de la CPU debido al I/O, el sobreuso de memoria (enormes thread pools) y la ineficiencia de las interacciones bloqueantes.

5.4. Reactive Streams

- La programación reactiva es un paradigma enfocado en el trabajo con flujos de datos finitos o infinitos de manera asíncrona.
- conjunto mínimo de interfaces para lograr esos fines:
  - org.reactivestreams.Publisher es un proveedor de datos que publica datos para los suscriptores en función de su demanda
  - org.reactivestreams.Subscriber es el consumidor de datos: puede recibir datos después de suscribirse a un editor
  - org.reactivestreams.Subscription se crea cuando un editor acepta un suscriptor
  - org.reactivestreams.Processor es tanto un suscriptor como un editor: se suscribe a un editor, procesa los datos y luego pasa los datos procesados al suscriptor

5.5. Flow API

- Flow API en JDK 9 corresponden a la especificación de secuencias reactivas.
- Con Flow API, si la aplicación inicialmente solicita N elementos, el editor envía como máximo N elementos al suscriptor.

- Las interfaces de Flow API están todas en la interfaz java.util.concurrent.Flow.
- Son semánticamente equivalentes a sus respectivas contrapartes de Reactive Streams.

5.6. Métodos en interfaces

Métodos privados

A partir de la versión Java 9 se pueden incluir métodos privados en las interfaces. Son utilizados desde métodos default

```
interface Inter1{
   //uso interno en la interfaz
   private int mayor(int a, int b){
      return (a>b)?a:b;
   }
   private int menor(int a, int b){
      return (a<b)?a:b;
   }
   default int suma(int a, int b){
      int s=0;
      //llamada a métodos privados
      for(int i=menor(a,b);i<mayor(a,b);i++){
         s+=i;
      }
      return s;
   }
}
```

```
public class Prueba{
   public static void main(String[] args){
      ClasePrueba cp=new ClasePrueba();
      System.out.println("suma "+cp.suma(10, 5));
   }
}
```

Los métodos privados también pueden ser estáticos, pudiendo ser llamados desde otros métodos estáticos o default de la interfaz

5.6. 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
   }
}
```

5.6. 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
   }
}
```

5.7. Logging

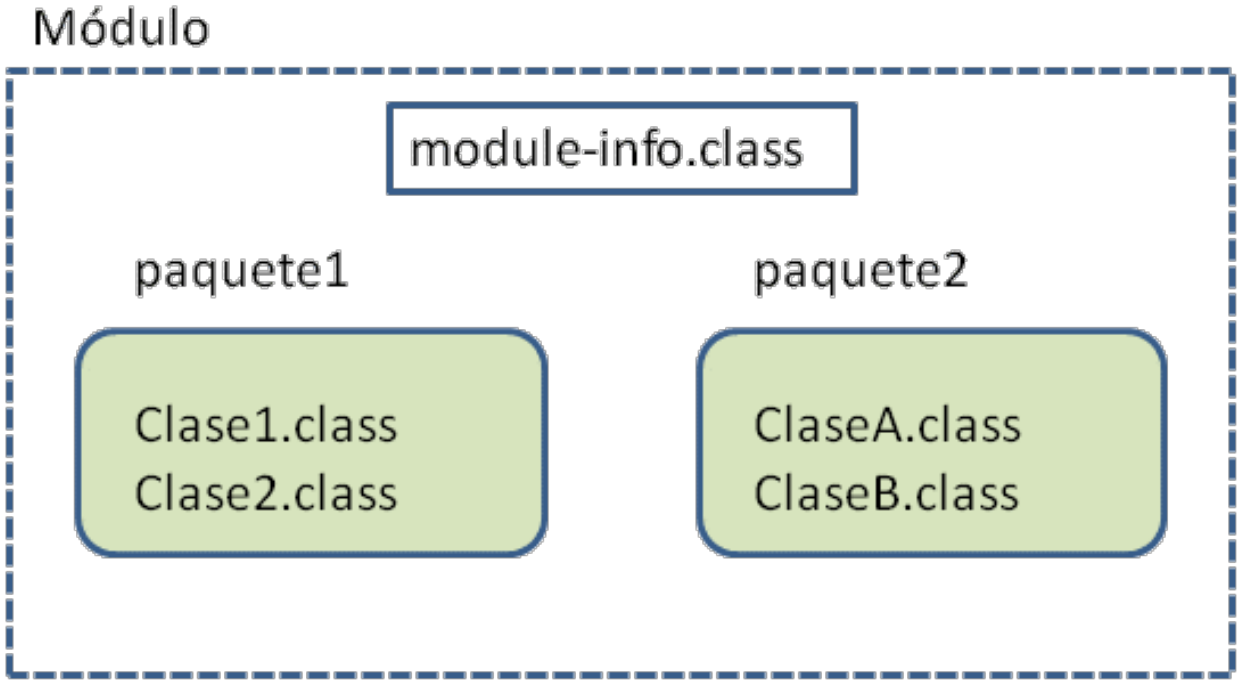
```
module Prueba_Login {
    requires java.logging;
}
```

```
import java.util.logging.Level;


// Create a Logger
Logger logger = Logger.getLogger(GFG1.class.getName());


logger.log(Level.INFO, "This is message 1");
logger.log(Level.WARNING, "This is message 2");
```

5.8. Módulos


Módulo
module-info.class
paquete1
Clase1.class
Clase2.class
paquete2
ClaseA.class
ClaseB.class

5.9. 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

5.10. Otros cambios de la API

Visto anteriormente

5.11. Características deprecated

- Nashorn JavaScript engine along with JJS tool is deprecated.
- Pack200 compression scheme for JAR files is deprecated.
- Java API for XML-Based Web Services (java.xml.ws)
- Java Architecture for XML Binding (java.xml.bind)
- JavaBeans Activation Framework (java.activation)
- Common Annotations (java.xml.ws.annotation)
- Common Object Request Broker Architecture (java.corba)
- JavaTransaction API (java.transaction)
- JDK Mission Control (JMC) is removed from standard JDK. It is available as standalone download.
- JavaFX is also removed from standard JDK. It is available as separate module to download.

DOCUMENTACIÓN

DOCUMENTACIÓN

- Características generales del lenguaje
- Clases esenciales
- El API de colecciones
- Nuevas características de Java 8

DOCUMENTACIÓN ADICIONAL

- Tutorial oficial de Java
- Libros gratuitos de Java en castellano
- Libros gratuitos sobre Java
- Libros recomendados: Java Language Especification y Programming Using JavaInformación sobre métodos Java
- Tutorial exhaustivo sobre Java 8

PROGRAMACIÓN ORIENTADA A OBJETOS

- Nociones fundamentales
- Diagramas de clases bajo UML

HERRAMIENTAS UTILIZADAS EN EL CURSO

- JSE 8-11
- Eclipse