
TEMARIO Y PREGUNTAS

By Anthony Martin Rosas Quispe

ÍNDICE

● Portabilidad.....	0
○ Definición.....	0
○ JDK.....	0
■ javac - Compilador Bytecode.....	0
○ JRE.....	0
○ JVM.....	0
○ Cómo se relacionan JDK vs JRE vs JVM.....	0
○ Es compilado y luego interpretado.....	0
○ Bytecode.....	0
● Ediciones de Java.....	0
○ Java ME.....	0
○ Java SE.....	0
○ Java EE.....	0
● Versiones de Java.....	0
○ Qué cambios ocurrieron con el traspaso de Java a Oracle con respecto a Java.....	0
○ Java 1.8 o 8 - Características.....	0
■ Expresiones Lambda.....	0
● Consumer, Biconsumer.....	0
● Supplier.....	0
● Function, Bifunction.....	0
○ Unary & Binary.....	0
● Predicate, Bipredicate.....	0
■ Interfaces Funcionales.....	0
● @FunctionalInterface.....	0
■ Referencia a métodos.....	0
● Referencia a métodos estáticos.....	0
● Referencia a un método de instancia de un objeto particular.....	0
● Referencia a un método de instancia de un tipo particular.....	0
● Referencia a un constructor.....	0
■ Default Method en Interfaces y Static Method en Interfaces.....	0
■ Java.Time.....	0
● LocalDate.....	0
● LocalTime.....	0
● LocalDateTime.....	0
● ZonedDateTime.....	0
● Period and Duration.....	0
■ ForEach() en Iterable.....	0

■ Streams.....	0
● filter(Predicate<T> predicate).....	0
● map(Function<T, R> mapper).....	0
● sorted(Comparator<T> comparator).....	0
● distinct().....	0
● collect(Collector<T, A, R> collector).....	0
● reduce(T identity, BinaryOperator<T> accumulator).....	0
● anyMatch(Predicate<T> predicate).....	0
● allMatch(Predicate<T> predicate).....	0
● noneMatch(Predicate<T> predicate).....	0
■ Completable Future.....	0
● runAsync().....	0
● supplyAsync().....	0
● handle().....	0
● exceptionally().....	0
■ Optionals.....	0
○ Java 1.11 o 11.....	0
■ Modularización.....	0
● open module.....	0
● exports.....	0
● requires transitive.....	0
■ Inferencia de Tipos.....	0
■ Nuevos métodos de Collections.....	0
● List.of & Set.of & Stream.of & Map.of.....	0
● List.copyOf & Set.copyOf & Map.copyOf.....	0
● toUnmodifiableList(), toUnmodifiableSet() y toUnmodifiableMap().....	0
■ Nuevos métodos de Streams.....	0
● takeWhile(), dropWhile(), iterate() y ofNullable().....	0
■ Nuevos métodos de Optional.....	0
● ifPresentOrElse(), or() y stream().....	0
■ Métodos privados en Interfaces.....	0
■ Jshell.....	0
■ Nuevo Garbage Collector.....	0
● Java en general.....	0
○ Qué es POO.....	0
○ Objeto.....	0
○ Clase.....	0
○ Paquetes.....	0
○ Modificadores de acceso.....	0
■ private.....	0
■ protected.....	0
■ public.....	0
■ default o sin modificador.....	0
○ Heap y Stack.....	0

○ Garbage Collector.....	0
■ Serial GC.....	0
■ Parallel GC.....	0
■ Concurrent Mark Sweep GC.....	0
■ G1 GC.....	0
○ Pilares de POO.....	0
■ Polimorfismo.....	0
■ Herencia.....	0
■ Abstracción.....	0
■ Encapsulamiento.....	0
○ Clases Abstractas.....	0
○ Clases Genéricas.....	0
○ Colecciones.....	0
■ Arrays.....	0
■ Clase Wrapper.....	0
■ List.....	0
■ Map.....	0
■ Stack.....	0
■ Queue.....	0
○ Manejo de Excepciones.....	0
■ Try, Catch, Finally.....	0
■ throws.....	0
■ Crear una excepción.....	0
○ Overloading, Overriding.....	0
○ Nested Class.....	0
■ Static Nested Class.....	0
■ Non-Static Nested Class / Inner Class.....	0
● Anonymous Inner Class.....	0
● Local Inner Class.....	0
○ Operador InstanceOf.....	0
○ Java Reactivo.....	0
■ RxJava.....	0
■ Project Reactor.....	0
■ Akka Streams.....	0
■ Testing Java Reactivo.....	0
■ R2DBC.....	0
■ Backpressure.....	0
■ Future & Promises.....	0
■ Callbacks.....	0
■ Completable Future.....	0
● Spring Initializer.....	0
● Swagger, OpenApi.....	0
● Spring vs Spring Boot.....	0
● Bean.....	0

○ Dependency Injection / DI / Inyección de dependencia.....	0
○ Tipos de DI.....	0
○ Ciclo de vida de un Bean / Life-cycle Bean.....	0
■ Aware Interfaces.....	0
■ BeanPostProcessor.....	0
● postProcessBeforeInitialization.....	0
● postProcessAfterInitialization.....	0
■ @PostConstruct.....	0
■ InitializingBean.....	0
■ @PreDestroy.....	0
■ Disposable Bean.....	0
■ Orden de Ejecución del ciclo de vida de un bean.....	0
■ JSR-250.....	0
○ Ambito de un Bean / @Scope.....	0
■ Singleton.....	0
■ Prototype.....	0
■ Request.....	0
■ Session.....	0
■ Application.....	0
■ WebSocket.....	0
○ Explicit Bean Declaration.....	0
○ Ambigüedad de un Bean / @Qualifier.....	0
○ @Primary.....	0
○ @Autowired.....	0
■ En Listas.....	0
● Stereotypes / Estereotipos.....	0
○ @Component.....	0
○ @Service.....	0
○ @Controller.....	0
○ @Repository.....	0
● Perfiles.....	0
○ Perfiles por Default.....	0
● Cómo funciona el @SpringBootApplication.....	0
● Properties in Spring.....	0
○ Carga de properties / @PropertySource.....	0
● Spring Expression Language / SpEL.....	0
● AOP / Programación orientada a aspectos.....	0
○ crosscutting.....	0
○ Aspect.....	0
○ Joinpoint.....	0
○ Advice.....	0
■ Before.....	0
■ After returning.....	0
■ After throwing.....	0

■ After finally.....	0
■ Around.....	0
○ Pointcut.....	0
○ Target object.....	0
○ Proxy.....	0
○ Weaving.....	0
● HTTP.....	0
○ Verbos HTTP / Metodos HTTP.....	0
■ GET.....	0
■ HEAD.....	0
■ POST.....	0
■ PUT.....	0
■ PATCH.....	0
■ DELETE.....	0
■ TRACE.....	0
■ OPTIONS.....	0
■ CONNECT.....	0
○ URI.....	0
■ Schema.....	0
■ Host.....	0
■ Path.....	0
○ Petición HTTP.....	0
■ Método HTTP.....	0
■ URI.....	0
■ Headers.....	0
■ Body.....	0
○ Respuesta HTTP.....	0
■ Versión.....	0
■ Status code.....	0
■ Headers.....	0
■ Body.....	0
○ Connectionless and Stateless.....	0
○ REST.....	0
■ Resource.....	0
○ Status HTTP.....	0
■ 1xx.....	0
■ 2xx.....	0
● 200 - OK.....	0
● 201 - CREATED.....	0
● 204 - NOT CONTENT.....	0
■ 3xx.....	0
■ 4xx.....	0
● 400 - BAD REQUEST.....	0
● 401 - UNAUTHORIZED.....	0

● 404 - NOT FOUND.....	0
● 405 - METHOD NOT ALLOWED.....	0
● 409 - CONFLICT.....	0
● 429 - TOO MANY REQUESTS.....	0
■ 5xx.....	0
● 500 - INTERNAL SERVER ERROR.....	0
● 501 - NOT IMPLEMENTED.....	0
● 502 - BAD GATEWAY.....	0
● 503 - SERVICE UNAVAILABLE.....	0
○ Versionamiento de API.....	0
■ Por URL.....	0
■ Por Header.....	0
■ Por Dominio.....	0
■ Por Request param.....	0
○ Paginación, filtros y ordenamiento con Query Params.....	0
● REST.....	0
○ Spring MVC.....	0
■ Model.....	0
■ View.....	0
■ Controller.....	0
○ Anotaciones.....	0
■ Controller.....	0
■ RestController.....	0
■ RequestMapping.....	0
■ GetMapping.....	0
■ PostMapping.....	0
■ PutMapping.....	0
■ DeleteMapping.....	0
■ PathMapping.....	0
■ PathVariable.....	0
■ RequestParam.....	0
■ ResponseStatus.....	0
■ Service.....	0
○ Otras clases útiles.....	0
■ ResponseEntity.....	0
■ HttpStatus.....	0
■ ResponseStatusException.....	0
● Spring Actuator.....	0
○ Configuración.....	0
○ Métricas.....	0
■ Timed.....	0
○ Prometheus.....	0
○ Grafana.....	0
● Spring Cache.....	0

○ Configuración.....	0
○ Uso.....	0
○ CacheEvict.....	0
○ Caché en memoria.....	0
○ Redis.....	0
■ Configuración.....	0
■ Uso.....	0
● Spring Security.....	0
○ Resuelve.....	0
■ Impersonators / Imitadores.....	0
● Ser Alguien.....	0
● Tener Algo.....	0
● Saber Algo.....	0
■ Upgraders / Mejoradores.....	0
■ Eavesdropper / Espía.....	0
○ Autenticación en memoria.....	0
○ Manejo de Usuarios.....	0
■ UserDetails.....	0
■ GrantedAuthority.....	0
■ UserDetailsService.....	0
■ UserDetailsServiceManager.....	0
● JdbcUserDetailsManager.....	0
○ Protección de Contraseñas.....	0
■ PasswordEncoder.....	0
■ Spring Security Crypto.....	0
■ BCrypt.....	0
○ Autenticación.....	0
■ AuthenticationProvider.....	0
■ SecurityContext.....	0
● DelegatingSecurityContextRunnable.....	0
● DelegatingSecurityContextCallable.....	0
● DelegatingSecurityContextExecutor.....	0
○ Restricción de Acceso.....	0
■ Authorities.....	0
■ Roles.....	0
■ Filters.....	0
○ Protección.....	0
■ CSRF.....	0
■ CORS.....	0
■ Tokens.....	0
● JWT.....	0
○ OAuth2.....	0
○ Spring Security Reactive.....	0
● Spring Data.....	0

○ Spring Java Persistence Api / Spring JPA.....	0
■ Diferencias con EJB.....	0
■ Diferencias con Hibernate.....	0
○ JDBC.....	0
○ JNDI.....	0
○ ORM.....	0
○ Transactional.....	0
● Microservicios.....	0
○ Definición.....	0
○ Ventajas y desventajas de aplicar microservicios.....	0
○ Teorema Gold Hammer.....	0
○ Configuración mediante repositorio remoto.....	0
○ Patrones de diseño de microservicios.....	0
■ Saga.....	0
■ CQRS (Command and Query Responsibility Segregation).....	0
■ Circuit Breaker.....	0
■ Chassis.....	0
■ Api Gateway.....	0
■ Service Discovery.....	0
■ Service Registry.....	0
○ Descomposición de negocio en Microservicio.....	0
■ Capacidades de negocio.....	0
■ Subdominios.....	0
■ etc.....	0
● Docker.....	0
○ Definición.....	0
○ Cómo se genera una imagen.....	0
○ Que es un container.....	0
● Kubernetes.....	0
○ Definición.....	0
○ Relación con Docker.....	0
○ Componentes principales.....	0
■ Control Plane.....	0
■ Cluster.....	0
■ Nodo.....	0
■ Pod.....	0
■ Service.....	0
■ Container.....	0
■ Job.....	0
● Redis.....	0
○ Definición.....	0
○ Estructura key-value.....	0
○ Redis vs Hazelcast.....	0
● Kafka.....	0

○ Definición.....	0
○ Kafka vs RabbitMQ.....	0
○ Comunicacion Asincrona vs Sincrona.....	0
○ Conceptos Principales.....	0
■ Zookeeper.....	0
■ Topic.....	0
■ Productor/Consumidor.....	0
■ Partición.....	0
■ Offset.....	0
● Patrones de diseño de Software.....	0
○ Patrones creacionales.....	0
■ Singleton.....	0
■ Factory.....	0
■ Abstract Factory.....	0
○ Patrones estructurales.....	0
■ Adapter.....	0
■ Decorator.....	0
■ Proxy.....	0
○ Patrones de comportamiento.....	0
■ Observer.....	0
■ Strategy.....	0
■ Command.....	0
● Principios SOLID.....	0
○ Acoplamiento.....	0
○ Cohesión.....	0
○ Principio de responsabilidad unica.....	0
○ Principio de abierto y cerrado.....	0
○ Principio de sustitución de Liskov.....	0
○ Principio de segregación de interfaces.....	0
○ Principio de inversión de dependencia.....	0
● Estilos arquitectonicos.....	0
○ Monolítico.....	0
○ Cliente Servidor.....	0
○ Peer-to-peer (P2P).....	0
○ Arquitectura en Capas.....	0
○ Microkernel.....	0
○ Service-Oriented-Architecture (SOA).....	0
○ Microservicios.....	0
○ Event Driven Architecture (EDA).....	0
○ Representational State Transfer.....	0
○ Hexagonal Architecture.....	0
○ Clean Architecture.....	0
● Patrones arquitectónicos.....	0
○ Data Transfer Object (DTO).....	0

○ Data Access Object.....	0
○ Polling.....	0
○ Webhook.....	0
○ Load Balance.....	0
○ Service Registry.....	0
○ Service Discovery.....	0
○ Api Gateway.....	0
○ Access Token.....	0
○ Single Sign On (inicio de sesión único).....	0
○ Store and forward.....	0
○ Circuit Breaker.....	0
○ Log aggregation.....	0
● Teorías.....	0
○ Don't Repeat Yourself (DRY).....	0
○ Separation of Concerns (SOC).....	0
○ Inversion of control (IoC).....	0
○ Code Smell - Código espagueti.....	0
○ Load Balance.....	0
○ Fault Tolerance.....	0
○ Escalabilidad.....	0
○ Config Server.....	0
○ Distributed Tracing.....	0
○ Orquestación / Coreografía.....	0
○ Semántica y taxonomía de apis.....	0
○ Backpressure.....	0
○ Cómo medir el nivel de madurez de una api rest.....	0
○ Patrón Strangler.....	0
○ Patrón sub/pub.....	0
● Enfoque y Metodologías de desarrollo.....	0
○ TDD - Test Driven Development.....	0
○ DDD - Domain Driven Design.....	0
○ BDD - Behavior Driven Development.....	0
● Devops.....	0
○ Definición.....	0
○ Jenkins.....	0

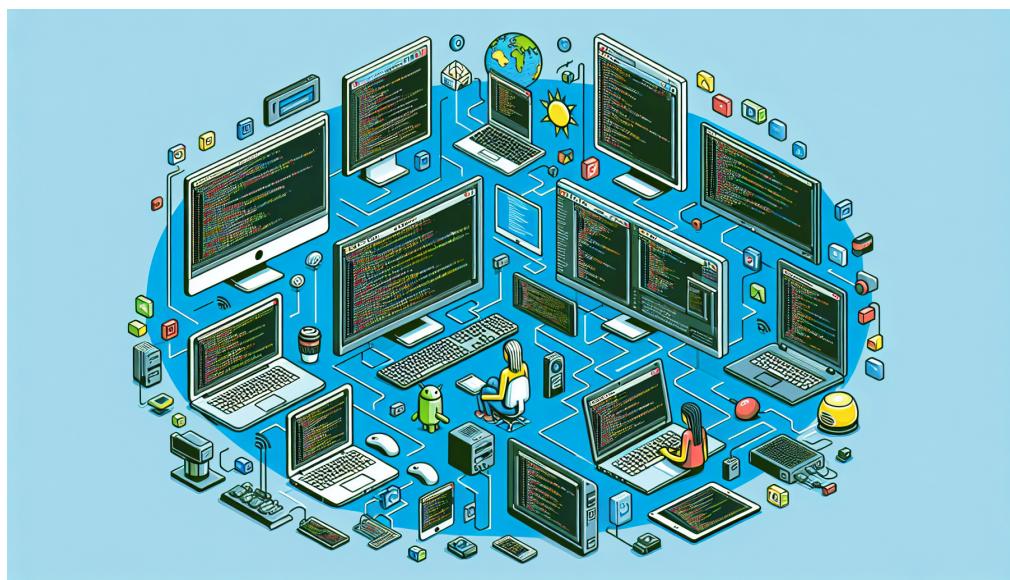
Java

- **Portabilidad**

- **Definición**

La capacidad de un programa Java de ejecutarse en diferentes plataformas o sistemas operativos sin necesidad de realizar cambios en el código fuente.

Debido a la portabilidad de Java, los programas escritos en este lenguaje se pueden ejecutar en diferentes sistemas operativos, como Windows, Linux, MacOS, entre otros. Esto ha hecho que Java sea un lenguaje muy popular para el desarrollo de aplicaciones empresariales y de software de sistemas.



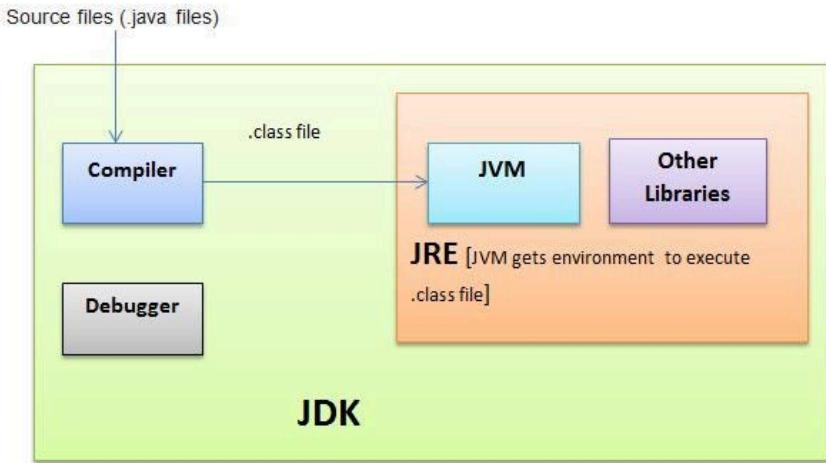
- **JDK**

Por sus siglas "Java Development Kit". Es un conjunto de herramientas de desarrollo para crear aplicaciones Java. Incluye el compilador de Java (javac), el intérprete de bytecode de Java (java), la biblioteca de clases de Java (rt.jar) y otras herramientas que son necesarias para desarrollar y ejecutar aplicaciones Java.

En resumen, el JDK es una herramienta fundamental para cualquier programador Java, ya que proporciona todo lo necesario para desarrollar, compilar y ejecutar aplicaciones Java.

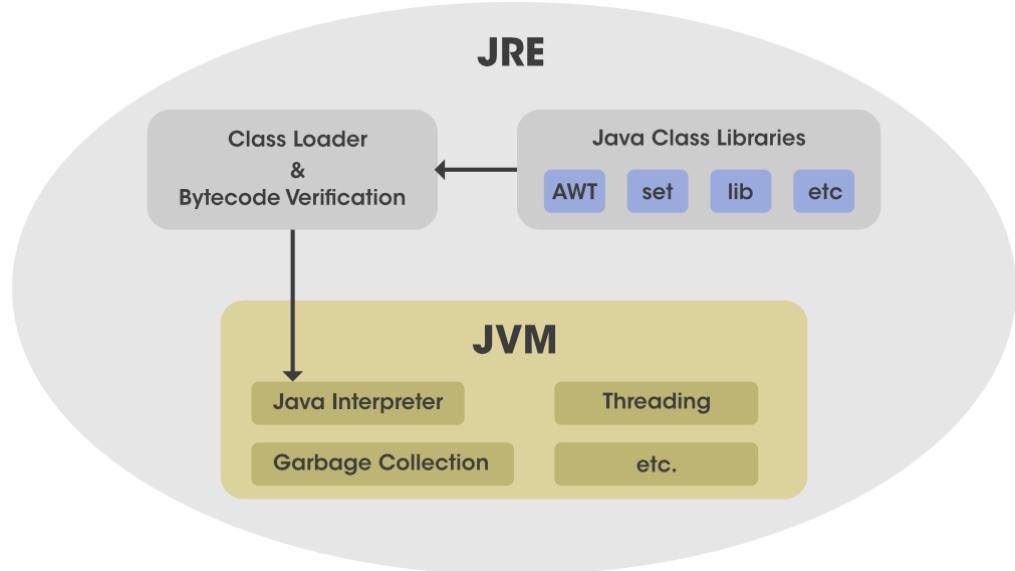
- **javac - Compilador Bytecode**

Javac es el compilador de Java, que es parte del conjunto de herramientas de desarrollo de Java conocido como JDK (Java Development Kit). Javac se utiliza para compilar el código fuente escrito en el lenguaje de programación Java en archivos de clase de Java (bytecode), que pueden ser interpretados y ejecutados por la máquina virtual de Java (JVM).



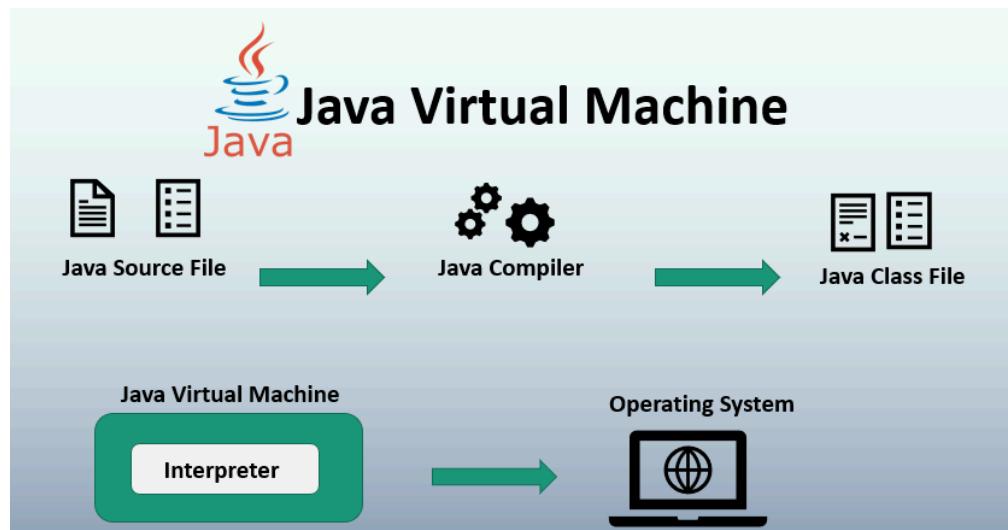
- **JRE**

Por sus siglas “Java Runtime Environment”. Es un entorno de ejecución para aplicaciones Java que permite a los usuarios ejecutar aplicaciones Java en sus sistemas sin necesidad de instalar el JDK. Incluye la JVM, bibliotecas de clases y otros archivos que son necesarios para ejecutar aplicaciones Java.



- **JVM**

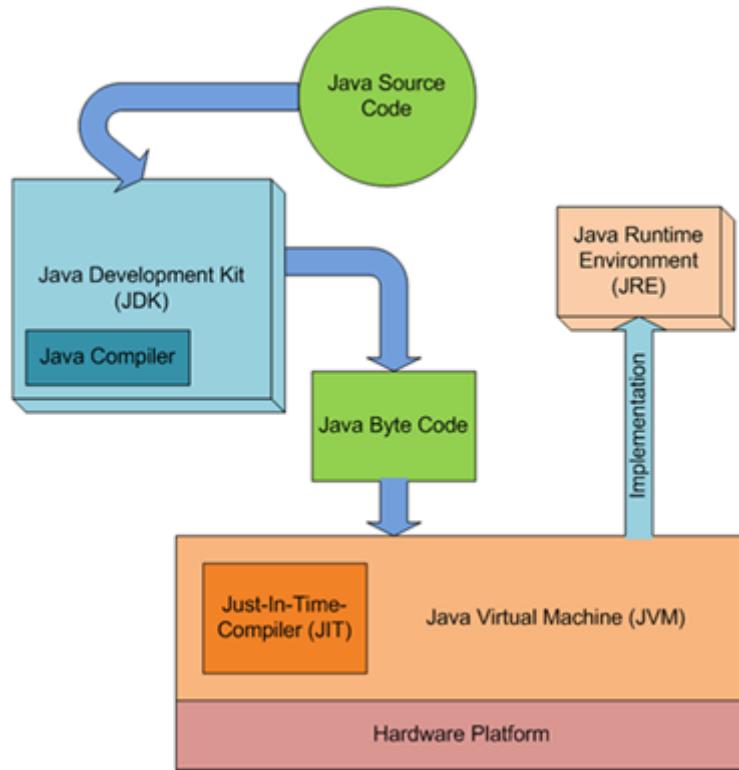
Por sus siglas “Java Virtual Machine”. Es un componente de software que ejecuta el código de byte de Java. Interpreta el bytecode y lo traduce en código de máquina nativo que puede ser ejecutado por el sistema operativo. La JVM es un componente clave para la portabilidad de Java, ya que permite que el mismo código de byte se ejecute en diferentes sistemas operativos y arquitecturas de hardware.



- **Cómo se relacionan JDK vs JRE vs JVM**

En términos de relación y diferencia, la JVM es la capa que se sitúa en medio del JRE y el código fuente de Java. El JRE proporciona la plataforma de ejecución para las aplicaciones Java, que se ejecutan en la JVM. Por otro lado, el JDK contiene el JRE y las herramientas de desarrollo de Java, incluyendo el compilador de Java. Es decir, JDK es un conjunto completo de herramientas de desarrollo de Java, mientras que JRE solo proporciona la plataforma de ejecución para las aplicaciones Java.

En resumen, la JVM es la pieza clave que permite la portabilidad de Java, el JRE es el entorno de ejecución para las aplicaciones Java, y el JDK es el conjunto completo de herramientas de desarrollo de Java, incluyendo el compilador.



- **Es compilado y luego interpretado**

En el contexto de Java, el proceso de compilación y ejecución implica dos fases distintas:

Compilación: El código fuente escrito en lenguaje Java es compilado utilizando el compilador de Java (javac) para generar código de byte de Java (archivo .class). Este archivo contiene instrucciones en un lenguaje de bajo nivel llamado bytecode que es independiente de la plataforma y que puede ser interpretado por la JVM.

Interpretación: Una vez que el archivo .class es generado, la JVM lo interpreta en tiempo de ejecución y lo traduce a código de máquina nativo de la plataforma específica en la que se está ejecutando. La JVM también proporciona otras características como la gestión de memoria y el recolector de basura.

En resumen, Java es un lenguaje compilado e interpretado porque se compila el código fuente a bytecode y luego se interpreta el bytecode en tiempo de ejecución por la JVM. Esto permite que el código Java sea altamente portátil y pueda ejecutarse en diferentes plataformas sin necesidad de recompilar el código fuente.

- **Bytecode**

Un bytecode es un código de bajo nivel generado por el compilador de un lenguaje de programación, que está diseñado para ser ejecutado por una

máquina virtual específica en lugar de ser ejecutado directamente por el procesador de la computadora.

En el caso de Java, el bytecode se genera a partir del código fuente escrito en lenguaje Java utilizando el compilador de Java (javac). El bytecode de Java es un conjunto de instrucciones en un formato binario especial que se puede interpretar por la máquina virtual de Java (JVM).

La ventaja de usar bytecode es que es un lenguaje de bajo nivel que es independiente de la plataforma y, por lo tanto, se puede utilizar para crear aplicaciones que se pueden ejecutar en diferentes plataformas sin la necesidad de recompilar el código fuente. La JVM se encarga de interpretar el bytecode y ejecutar las instrucciones correspondientes en la plataforma específica en la que se está ejecutando la aplicación.

- **Ediciones de Java**

- **Java ME**

Por sus siglas “Java Micro Edition”. Es una edición de Java diseñada para dispositivos móviles y dispositivos de bajo consumo, como teléfonos móviles, PDAs(*) y televisores. Java ME incluye un subconjunto de la API de Java SE y una serie de perfiles que se adaptan a diferentes tipos de dispositivos.

(*) PDA se refiere a "Personal Digital Assistant" en inglés, que traducido al español significa "Asistente Digital Personal". Un PDA es un dispositivo electrónico de mano que combina funcionalidades de computadora, teléfono, agenda electrónica y otros servicios útiles. Los PDAs fueron populares en la década de 1990 y principios de la década de 2000, antes de la proliferación de los smartphones que incorporaban funcionalidades similares.

NOTAS: Aunque Java ME fue una vez una opción importante para el desarrollo de aplicaciones móviles en dispositivos con recursos limitados, su uso ha disminuido considerablemente en favor de otras tecnologías más modernas y especializadas para plataformas móviles.



- **Java SE**

Por sus siglas “Java Standard Edition”. Es la edición “estándar” de Java, que proporciona la API de Java básica y el entorno de ejecución para aplicaciones de escritorio y servidores. Java SE incluye las bibliotecas y herramientas de desarrollo necesarias para construir y ejecutar aplicaciones Java en una amplia variedad de plataformas.

- **Java EE**

Por sus siglas “Java Enterprise Edition”. Es una edición de Java diseñada para construir aplicaciones empresariales escalables y robustas. Java EE incluye un conjunto de APIs y especificaciones para aplicaciones web, aplicaciones empresariales y servicios web, así como un servidor de aplicaciones Java EE que proporciona el entorno de ejecución para las aplicaciones Java EE.

NOTAS: Cada edición de Java está diseñada para satisfacer diferentes necesidades y casos de uso. Java ME se enfoca en dispositivos móviles y de bajo consumo, Java SE es para aplicaciones de escritorio y servidores, y Java EE es para aplicaciones empresariales escalables.

- **Versiones de Java**

- **¿Qué cambios ocurrieron con el traspaso de Java a Oracle con respecto a Java?**

Cambio en la Licencia de Distribución de Oracle JDK: Antes de la adquisición, la implementación de referencia de Java, el JDK (Java

Development Kit), estaba disponible con una licencia de código abierto conocida como la Licencia Pública General de GNU (GPL). Con Oracle, se introdujo una nueva licencia comercial, y Oracle JDK comenzó a requerir una suscripción de soporte para uso comercial en entornos de producción.

OpenJDK: Oracle inició un enfoque más fuerte en OpenJDK, el proyecto de código abierto que implementa las especificaciones de Java. La implementación de referencia de Java, en lugar de ser exclusivamente Oracle JDK, ahora es cada vez más OpenJDK.

Ciclo de Liberación más Rápido: Oracle introdujo un nuevo modelo de ciclo de liberación para Java, con versiones programadas cada seis meses. Esto permitió la entrega más rápida de nuevas características y mejoras, pero también requirió que los desarrolladores se mantuvieran al día con las actualizaciones más frecuentes.

Modelo de Soporte LTS: Oracle ofrece versiones LTS de Java con soporte a largo plazo para aquellos que requieren estabilidad a largo plazo. Otras versiones, que se lanzan cada seis meses, son de corto plazo y reciben soporte solo hasta el próximo lanzamiento.

- **Java 1.8 o 8 - Características**
 - **Expresiones Lambda**

Las expresiones lambda son una característica introducida en Java en la versión 8 como parte de las mejoras del lenguaje. Permiten expresar instancias de interfaces funcionales (interfaces con un solo método abstracto) de una manera más concisa y, a menudo, más legible. Las expresiones lambda se utilizan principalmente para definir implementaciones de interfaces funcionales de forma más compacta.

Las expresiones lambda son especialmente útiles cuando se trabaja con interfaces funcionales, como en el caso de Java 8, que introdujo muchas interfaces funcionales en el paquete java.util.function. Estas interfaces funcionales se utilizan comúnmente en contextos como streams, programación funcional y manipulación de colecciones. Las expresiones lambda facilitan la implementación rápida y concisa de funciones para operar en estos contextos.

```
// Antes de las expresiones Lambda (Java 7 y anteriores)
Runnable runnableAntes = new Runnable() {
    @Override
    public void run() {
        System.out.println("Hola desde la implementación de Runnable");
    }
};
```

```
// Con expresiones Lambda (Java 8 y posteriores)
Runnable runnableLambda = () -> {
    System.out.println("Hola desde la expresión lambda de Runnable");
};
```

- **Consumer, BiConsumer**

Consumer: Representa una operación que toma un solo argumento y no devuelve ningún resultado. Puede ser utilizada cuando se necesita realizar una acción sobre un valor.

```
Consumer<String> printString = s -> System.out.println(s);
printString.accept("Hola, Mundo!");
```

BiConsumer: Representa una operación que toma dos argumentos y no devuelve ningún resultado. Puede ser utilizada en contextos donde se necesita realizar una acción sobre dos valores.

```
BiConsumer<String, Integer> printKeyValue = (key, value) -> System.out.println(key + ": " + value);
printKeyValue.accept("Edad", 25);
```

- **Supplier**

Representa un proveedor de resultados. No toma ningún argumento, pero produce un resultado.

```
Supplier<Double> randomValue = () -> Math.random();
System.out.println("Número aleatorio: " + randomValue.get());
```

- **Function, BiFunction**

Representan funciones que toman uno o dos argumentos, respectivamente, y producen un resultado.

```
Function<Integer, String> intToString = n -> String.valueOf(n);
BiFunction<Integer, Integer, Integer> sum = (a, b) -> a + b;
```

- **Unary & Binary**

Son subtipos de Function y BiFunction, respectivamente, donde los argumentos y el resultado tienen el mismo tipo.

```
UnaryOperator<Integer> square = n -> n * n;
BinaryOperator<Integer> multiply = (a, b) -> a * b;
```

- **Predicate, Bipredicate**

Representan funciones que toman uno o dos argumentos, respectivamente, y devuelven un valor booleano.

```
Predicate<String> isLong = s -> s.length() > 5;
BiPredicate<Integer, Integer> isSumEven = (a, b) -> (a + b) % 2 == 0;
```

- **Interfaces Funcionales**

Las interfaces funcionales son interfaces de Java que tienen un único método abstracto (sin contar los métodos heredados de Object). Estas interfaces fueron introducidas en Java 8 como parte de la mejora del lenguaje para admitir programación funcional. La anotación `@FunctionalInterface` se utiliza para indicar que una interfaz es funcional.

Las interfaces funcionales son esenciales para aprovechar las expresiones lambda y los métodos de referencia, que son características clave de la programación funcional en Java. Permiten tratar las funciones como objetos de primera clase, lo que facilita el trabajo con operaciones de alto orden, como pasar funciones como argumentos a otros métodos.

- **@FunctionalInterface**

La anotación `@FunctionalInterface` es opcional, pero su uso es recomendado. No afecta la funcionalidad de la interfaz, pero proporciona claridad y documentación en el código, indicando que la interfaz está diseñada para ser funcional y, por lo tanto, es adecuada para ser utilizada con expresiones lambda.

```
@FunctionalInterface
public interface MiInterfazFuncional {
    void miMetodoAbstracto();

    // Otros métodos pueden ser estáticos o por defecto (default)
    default void otroMetodo() {
        System.out.println("Este es un método por defecto.");
    }

    static void metodoEstatico() {
        System.out.println("Este es un método estático.");
    }
}
```

```
}
```

En este caso, MiInterfazFuncional es una interfaz funcional porque tiene un solo método abstracto (miMetodoAbstracto). La anotación @FunctionalInterface es opcional pero proporciona claridad sobre la intención de la interfaz.

Puedes utilizar esta interfaz funcional con expresiones lambda o métodos de referencia de la siguiente manera:

```
MiInterfazFuncional miFuncion = () -> System.out.println("Implementación de la interfaz funcional");
miFuncion.miMetodoAbstracto();
```

■ Referencia a métodos

La referencia a métodos en Java es una característica que permite simplificar la creación de expresiones lambda que llaman a un único método. Proporciona una forma más concisa de expresar una lambda cuando simplemente se está reenviando una llamada a un método existente. La referencia a métodos es especialmente útil cuando el cuerpo de la lambda consiste en una sola llamada a un método, ya que permite reducir la redundancia en el código.

- **Referencia a métodos estáticos**

Se refiere a un método estático

```
// Expresión Lambda
Function<String, Integer> toInt = s -> Integer.parseInt(s);

// Referencia a método estático
Function<String, Integer> toIntRef = Integer::parseInt;
```

- **Referencia a un método de instancia de un objeto particular**

Se refiere a un método de una instancia específica.

```
// Expresión Lambda
BiFunction<String, Integer, String> substring = (s, start) -> s.substring(start);

// Referencia a método de instancia
BiFunction<String, Integer, String> substringRef = String::substring;
```

- **Referencia a un método de instancia de un tipo particular**

Se refiere a un método de instancia de un tipo en particular.

```
// Expresión Lambda
Predicate<String> isEmpty = s -> s.isEmpty();

// Referencia a método de instancia
Predicate<String> isEmptyRef = String::isEmpty;
```

- **Referencia a un constructor**

Se refiere a un constructor.

```
// Expresión Lambda
Supplier<List<String>> listSupplier = () -> new ArrayList<>();

// Referencia a constructor
Supplier<List<String>> listSupplierRef = ArrayList::new;
```

- **Default Method en Interfaces y Static Method en Interfaces**

En Java, a partir de la versión 8, se introdujeron dos nuevas características en las interfaces: los "Default Methods" (Métodos por Defecto) y los "Static Methods" (Métodos Estáticos). Estas adiciones permitieron la introducción de nuevas funcionalidades en las interfaces sin romper la compatibilidad hacia atrás.

- Default Methods (Métodos por Defecto):**

- Definición:

Un "Default Method" es un método definido dentro de una interfaz que proporciona una implementación predeterminada. Puede ser heredado por clases que implementan la interfaz, pero estas clases pueden optar por sobreescribirlo.

- Uso:

Permite agregar nuevos métodos a interfaces sin afectar las clases existentes que las implementan.
Se utiliza la palabra clave **default** seguida de la implementación del método.

```
public interface Saludable {
    void saludar();
```

```

    default void despedir() {
        System.out.println("Hasta luego");
    }

public class Persona implements Saludable {
    @Override
    public void saludar() {
        System.out.println("Hola");
    }
}

// Uso
Persona persona = new Persona();
persona.saludar(); // Imprime "Hola"
persona.despedir(); // Imprime "Hasta Luego"

```

Static Methods (Métodos Estáticos):

Definición:

Un "Static Method" en una interfaz es un método que se puede llamar en la propia interfaz, no en instancias de clases que la implementan.

Uso:

Proporciona utilidades relacionadas con la interfaz que no dependen de instancias específicas.

```

public interface Calculadora {
    int sumar(int a, int b);

    static int restar(int a, int b) {
        return a - b;
    }
}

// Uso
int resultadoSuma = Calculadora.sumar(5, 3);
int resultadoResta = Calculadora.restar(5, 3);

```

NOTAS: Una de las principales diferencias de una con la otra es la herencia. Puedes heredar de múltiples interfaces que tengan métodos por defecto con implementaciones, lo que no es posible con métodos estáticos. Si hay posibilidad de conflicto con los métodos por defecto en interfaces superiores, puede ser preferible utilizar

métodos estáticos para evitar la herencia no deseada de implementaciones por defecto.

■ **Java.Time**

java.time es un paquete en Java que proporciona clases para trabajar con fechas y horas. Fue introducido en Java 8 para abordar las limitaciones y problemas de diseño del antiguo paquete java.util.Date y java.util.Calendar. El nuevo paquete java.time es parte del paquete java.time y ofrece una API más moderna, robusta y fácil de usar para el manejo de fechas y horas.

- **LocalDate**

Representa una fecha sin información sobre la hora.

2024-01-11

- **LocalTime**

Representa un tiempo sin información sobre la fecha.

10:30:45.123456789

- **LocalDateTime**

Combina fecha y hora, sin información sobre la zona horaria.

2024-01-11T10:30:45.123456789

- **ZonedDateTime**

Similar a LocalDateTime, pero incluye información sobre la zona horaria.

2024-01-11T05:30:45.123456789-05:00[America/New_York]

- **Period and Duration**

Representan duraciones y períodos, respectivamente.

```
// Fecha y hora de inicio
LocalDateTime inicio = LocalDateTime.of(2022, 1, 1, 10, 30);
System.out.println("Inicio: " + inicio);

// Fecha y hora de finalización
LocalDateTime fin = LocalDateTime.of(2024, 1, 11, 15, 45);
System.out.println("Fin: " + fin);
```

```

// Calcular la diferencia en términos de años, meses y días usando Period
Period periodo = Period.between(LocalDate.from(inicio), LocalDate.from(fin));
System.out.println("Diferencia en términos de años, meses y días: " + periodo.getYears() + " años, " + periodo.getMonths() + " meses, "
+ periodo.getDays() + " días");

// Calcular la diferencia en términos de horas, minutos y segundos usando Duration
Duration duracion = Duration.between(inicio, fin);
System.out.println("Diferencia en términos de horas, minutos y segundos: " + duracion.toHours() + " horas, " + duracion.toMinutesPart()
+ " minutos, " + duracion.toSecondsPart() + " segundos");

```

```

Inicio: 2022-01-01T10:30
Fin: 2024-01-11T15:45
Diferencia en términos de años, meses y días: 2 años, 10 meses, 10 días
Diferencia en términos de horas, minutos y segundos: 5546 horas, 15 minutos, 0 segundos

```

■ **ForEach() en Iterable**

El método forEach en Java se utiliza para iterar sobre cada elemento de una colección (que implementa la interfaz Iterable) y aplicar una acción a cada elemento individualmente.

```

iterable.forEach(elemento -> {
    // Acción a realizar en cada elemento
});

```

■ **Streams**

Los streams en Java son una secuencia de elementos que soportan operaciones de procesamiento de datos en una forma declarativa y funcional. Proporcionan una manera de operar y manipular colecciones de datos de una manera similar a la programación funcional. Las operaciones de los streams son divididas en dos categorías: operaciones intermedias y operaciones terminales.

- **filter(Predicate<T> predicate)**

Selecciona los elementos de un stream que cumplan con un predicado dado.

Retorna un nuevo stream que contiene sólo los elementos que satisfacen el predicado.

```

// Creamos una Lista de números
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

```

```
// filter: selecciona Los números pares
List<Integer> pares = numeros.stream()
    .filter(numero -> numero % 2 == 0)
    .collect(Collectors.toList());
System.out.println("Números pares: " + pares);
// Resultado esperado: [2, 4, 6, 8, 10]
```

- **map(Function<T, R> mapper)**

Transforma cada elemento de un stream aplicando una función a cada uno.

Retorna un nuevo stream que contiene los resultados de aplicar la función a cada elemento del stream original.

```
// map: transforma Los números a sus cuadrados
List<Integer> cuadrados = numeros.stream()
    .map(numero -> numero * numero)
    .collect(Collectors.toList());
System.out.println("Cuadrados de los números: " + cuadrados);
// Resultado esperado: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- **sorted(Comparator<T> comparator)**

Ordena los elementos de un stream utilizando un comparador dado.

Retorna un nuevo stream que contiene los elementos ordenados según el comparador especificado

```
// sorted: ordena Los números en orden descendente
List<Integer> descendente = numeros.stream()
    .sorted((a, b) -> b.compareTo(a))
    .collect(Collectors.toList());
System.out.println("Números ordenados en orden descendente: " + descendente);
// Resultado esperado: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

- **distinct()**

Elimina los elementos duplicados de un stream.

Retorna un nuevo stream que contiene sólo elementos únicos.

```
// distinct: elimina Los números duplicados
```

```
List<Integer> sinDuplicados = Arrays.asList(1, 2, 2, 3, 3, 3, 4, 4, 5, 5)
    .stream()
    .distinct()
    .collect(Collectors.toList());
System.out.println("Números sin duplicados: " + sinDuplicados);
// Resultado esperado: [1, 2, 3, 4, 5]
```

- **collect(Collector<T, A, R> collector)**

Recolesta los elementos de un stream en una colección o en otro contenedor mutable.

Retorna el resultado de la recolección, que puede ser una lista, un set, un mapa, etc.

- **reduce(T identity, BinaryOperator<T> accumulator)**

Combina los elementos de un stream utilizando un operador de reducción binaria.

Retorna el resultado de la reducción.

```
// reduce: suma todos los números
int suma = numeros.stream()
    .reduce(0, (a, b) -> a + b);
System.out.println("Suma de todos los números: " + suma);
// Resultado esperado: 55
```

- **anyMatch(Predicate<T> predicate)**

Comprueba si al menos un elemento de un stream cumple con un predicado dado.

Retorna true si al menos un elemento cumple con el predicado, de lo contrario false.

```
// anyMatch: verifica si hay algún número mayor que 5
boolean algunoMayorQue5 = numeros.stream()
    .anyMatch(numero -> numero > 5);
System.out.println("¿Hay algún número mayor que 5?: " + algunoMayorQue5);
// Resultado esperado: true
```

- **allMatch(Predicate<T> predicate)**

Comprueba si todos los elementos de un stream cumplen con un predicado dado.

Retorna true si todos los elementos cumplen con el predicado, de lo contrario false.

```
// allMatch: verifica si todos Los números son menores que 20
boolean todosMenoresQue20 = numeros.stream()
    .allMatch(numero -> numero < 20);
System.out.println("¿Son todos los números menores que 20?: " + todosMenoresQue20);
// Resultado esperado: true
```

- **noneMatch(Predicate<T> predicate)**

Comprueba si ningún elemento de un stream cumple con un predicado dado.

Retorna true si ninguno de los elementos cumple con el predicado, de lo contrario false.

```
// noneMatch: verifica si ninguno de Los números es igual a 100
boolean ningunoIgual100 = numeros.stream()
    .noneMatch(numero -> numero == 100);
System.out.println("¿Ninguno de los números es igual a 100?: " + ningunoIgual100);
// Resultado esperado: true
```

- **CompletableFuture**

CompletableFuture es una clase en Java que representa un futuro completado (o posiblemente aún no completado) y proporciona una forma de trabajar con resultados que pueden estar disponibles en el futuro. Esta clase forma parte del paquete java.util.concurrent y se introdujo en Java 8 como parte del conjunto de características para mejorar la programación concurrente y asíncrona en Java.

- **runAsync(Runnable runnable)**

Este método inicia una tarea asíncrona que realiza un trabajo sin producir un resultado. Toma un objeto Runnable que representa la tarea a realizar.

Este método devuelve un CompletableFuture<Void>, ya que la tarea no produce ningún resultado.

```
CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
    // Simular una tarea que toma tiempo
    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
});
```

```

    }
    System.out.println("Tarea asíncrona completada.");
});

// Esperar a que la tarea asíncrona se complete (esto es solo para fines de demostración)
future.join();

```

- **supplyAsync(Supplier<U> supplier)**

Este método inicia una tarea asíncrona que produce un resultado. Toma un objeto Supplier<U> que representa la tarea a realizar y que produce un resultado de tipo U.

Este método devuelve un CompletableFuture<U>, que contendrá el resultado producido por la tarea.

```

CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {
    // Simular una tarea que produce un resultado
    return ThreadLocalRandom.current().nextInt(1, 100);
});

// Adjuntar un callback para imprimir el resultado cuando esté disponible
future.thenAccept(result -> System.out.println("Resultado: " + result));

// Esperar a que la tarea asíncrona se complete (esto es solo para fines de demostración)
future.join();

```

- **handle(BiFunction<? super T, Throwable, ? extends U> handler)**

Este método permite manejar el resultado o una excepción lanzada por una tarea asíncrona. Toma una función BiFunction que recibe el resultado (si la tarea fue exitosa) o la excepción (si ocurrió alguna) y devuelve un nuevo valor.

La función debe manejar tanto el resultado como la excepción y devolver un nuevo valor.

Este método devuelve un nuevo CompletableFuture<U> que contendrá el resultado manejado.

```

CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {
    // Simular una tarea que lanza una excepción
    throw new RuntimeException("Ocurrió un error");
});

```

```

// Manejar la excepción y proporcionar un valor alternativo
CompletableFuture<Integer> handledFuture = future.handle((result, ex) -> {
    if (ex != null) {
        System.out.println("Se produjo una excepción: " + ex.getMessage());
        return -1; // Valor alternativo en caso de error
    } else {
        return result; // Devolver el resultado normal
    }
});

// Imprimir el resultado o el valor alternativo
handledFuture.thenAccept(result -> System.out.println("Resultado o valor alternativo: " + result));

// Esperar a que la tarea asíncrona se complete (esto es solo para fines de demostración)
handledFuture.join();

```

- **exceptionally(Function<Throwable, ? extends T> exceptionHandler)**

Este método se utiliza para manejar una excepción lanzada por una tarea asíncrona y proporcionar un valor alternativo en caso de fallo.

Toma una función Function que recibe la excepción y devuelve un valor alternativo.

Este método devuelve un nuevo CompletableFuture<T> que contendrá el valor alternativo en caso de fallo.

```

CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {
    // Simular una tarea que lanza una excepción
    throw new RuntimeException("Ocurrió un error");
});

// Manejar la excepción y proporcionar un valor alternativo
CompletableFuture<Integer> exceptionallyFuture = future.exceptionally(ex -> {
    System.out.println("Se produjo una excepción: " + ex.getMessage());
    return -1; // Valor alternativo en caso de error
});

// Imprimir el resultado o el valor alternativo
exceptionallyFuture.thenAccept(result -> System.out.println("Resultado o valor alternativo: " + result));

// Esperar a que la tarea asíncrona se complete (esto es solo para fines de demostración)
exceptionallyFuture.join();

```

NOTA: En resumen, handle es más genérico ya que puede manejar tanto resultados normales como excepciones, mientras que exceptionally se centra exclusivamente en manejar excepciones y proporcionar valores alternativos en caso de errores. La elección entre ellos depende de los requisitos específicos de manejo de errores en una tarea asíncrona dada.

■ Optionals

Los Optionals en Java 8 son una característica muy útil que proporciona una forma más segura y más clara de trabajar con valores que pueden ser nulos. Un Optional puede contener un valor o puede ser vacío si el valor es nulo. Los optionals nacieron como una solución a los [NullPointerException](#).

```
import java.util.Optional;

public class Persona {
    private String nombre;

    public Persona(String nombre) {
        this.nombre = nombre;
    }

    public Optional<String> getNombre() {
        return Optional.ofNullable(nombre);
    }
}

public class Main {
    public static void main(String[] args) {
        Persona persona1 = new Persona("Alice");
        Persona persona2 = new Persona(null);

        // Obtener el nombre de la primera persona de forma segura
        Optional<String> nombre1 = persona1.getNombre();
        nombre1.ifPresentOrElse(
            nombre -> System.out.println("Nombre persona 1: " + nombre), // Si está presente, imprime el nombre
            () -> System.out.println("Nombre persona 1 no disponible") // Si no está presente, imprime un mensaje
        );

        // Obtener el nombre de la segunda persona de forma segura
        Optional<String> nombre2 = persona2.getNombre();
        nombre2.ifPresentOrElse(
            nombre -> System.out.println("Nombre persona 2: " + nombre), // Si está presente, imprime el nombre
            () -> System.out.println("Nombre persona 2 no disponible") // Si no está presente, imprime un mensaje
    }
}
```

```
        );
    }

// Nombre persona 1: Alice
// Nombre persona 2 no disponible
```

of(T value):

- Este método crea un Optional que contiene un valor especificado.
- Si el valor pasado como argumento es nulo, este método arrojará una excepción NullPointerException.
- Por lo general, se utiliza cuando se sabe que el valor no es nulo y se desea envolverlo en un Optional.

```
Optional<String> optional = Optional.of("Hola");
```

ofNullable(T value):

- Este método crea un Optional que contiene un valor especificado, permitiendo que el valor sea nulo.
- Si el valor pasado como argumento es nulo, este método devolverá un Optional vacío.
- Es útil cuando no se está seguro de si el valor es nulo y se desea manejar ambos casos de manera segura.

```
String valor = null;
Optional<String> optional = Optional.ofNullable(valor);
```

○ **Java 1.11 o 11**
■ **Modularización**

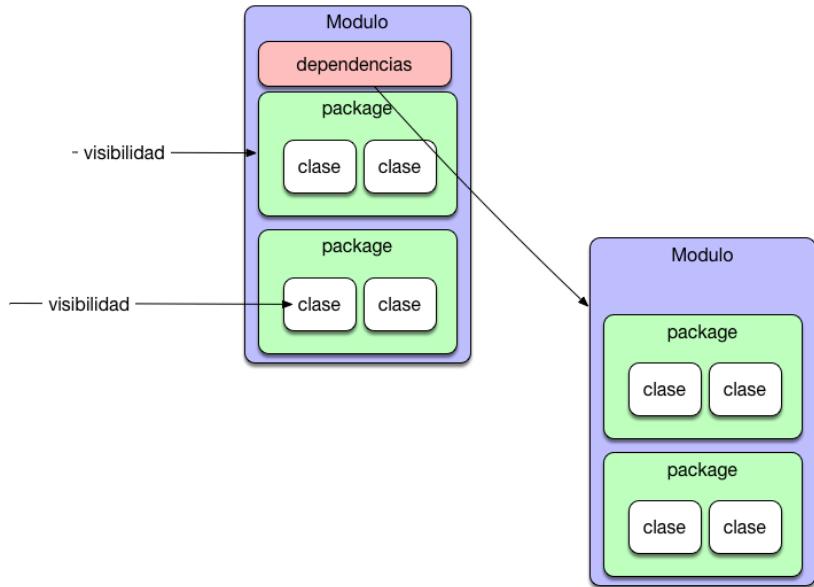
Con Java 9, se introdujeron importantes cambios en el sistema de módulos para permitir la modularización del código. Estos cambios continuaron y se mejoraron en versiones posteriores, incluida Java 11

Módulos:

- Un módulo en Java es un conjunto de paquetes relacionados que se pueden agrupar juntos y se pueden encapsular para controlar el acceso a ellos.
- Un módulo se define utilizando un descriptor de módulo (module-info.java) que especifica su nombre y las directivas de módulo.

module-info.java:

- El descriptor de módulo (module-info.java) se utiliza para declarar un módulo y especificar sus dependencias y exportaciones.
- Este archivo se coloca en el directorio raíz del módulo y contiene la declaración del módulo y las directivas de módulo.



- **open module**

La directiva open module se utiliza para permitir que otros módulos accedan a los miembros (clases, interfaces, etc.) de un módulo mediante la reflexión, incluso si esos miembros son private.

Esto es útil para ciertos casos de uso, como la serialización y la deserialización de objetos, donde se requiere acceso a campos privados.

- **exports**

La directiva exports se utiliza para exportar paquetes desde un módulo, lo que permite que otros módulos accedan a los tipos públicos en esos paquetes.

Los paquetes exportados son accesibles para otros módulos, pero los tipos no exportados dentro de esos paquetes siguen siendo inaccesibles.

- **requires transitive**

La directiva require transitive se utiliza para requerir que los módulos que dependen de un módulo actual también

dependan automáticamente de los módulos transitivamente requeridos por el módulo actual.

Esto simplifica la dependencia de los módulos que requieren el módulo actual, ya que no necesitan declarar explícitamente las dependencias transitivas.

```
// module-info.java

module miModulo {
    // Exporta el paquete "com.example.miproyecto" para que otros módulos puedan acceder a sus tipos públicos
    exports com.example.miproyecto;

    // Abre el paquete "com.example.miproyecto" para permitir la reflexión en sus miembros privados
    open com.example.miproyecto;

    // Requiere que otros módulos dependan automáticamente de los módulos transitivamente requeridos
    requires transitive otroModulo;
}
```

■ Inferencia de Tipos

La inferencia de tipos en Java es una característica introducida en Java 7 que permite al compilador deducir el tipo de una variable en función del contexto en el que se utiliza. Esto ayuda a reducir la verbosidad del código y a hacerlo más legible.

Antes de la introducción de la inferencia de tipos, cuando uno declaraba una variable, necesitaba especificar su tipo explícitamente. Por ejemplo:

```
String mensaje = "Hola mundo";
List<String> lista = new ArrayList<String>();
```

Con la inferencia de tipos, puedes omitir la repetición del tipo al lado derecho de la asignación, siempre y cuando el compilador pueda determinar el tipo de manera segura en función del contexto. Utilizamos el operador de diamante <> para indicar que queremos que el compilador infiera el tipo. Por ejemplo:

```
String mensaje = "Hola mundo";
List<String> lista = new ArrayList<>();
```

El concepto de var en Java se refiere a la característica de inferencia de tipos introducida en Java 10. La palabra clave var se utiliza para

declarar variables locales con tipos deducidos por el compilador en función del contexto en el que se utiliza.

Cuando se utiliza var para declarar una variable, el tipo de la variable se infiere automáticamente por el compilador en tiempo de compilación. Esto significa que no es necesario especificar explícitamente el tipo de la variable al declararla; en su lugar, el compilador deducirá el tipo basándose en la expresión que se utiliza para inicializar la variable.

Por ejemplo, considera el siguiente código:

```
var mensaje = "Hola mundo";
```

■ Nuevos métodos de Collections

Estas son algunas de las formas de crear colecciones inmutables y convertir colecciones mutables en inmutables en Java.

Mutable: Una colección mutable es aquella cuyo contenido puede ser modificado después de que se crea. Esto significa que se pueden agregar, eliminar o modificar elementos en la colección después de su creación.

Inmutable: Una colección inmutable es aquella cuyo contenido no puede ser modificado después de que se crea. Una vez que se crea una colección inmutable, su contenido no puede ser alterado de ninguna manera. No se pueden agregar, eliminar ni modificar elementos en la colección.

- **List.of & Set.of & Stream.of & Map.of**

Estos métodos estáticos se introdujeron en Java 9 y permiten crear colecciones inmutables de forma compacta y segura.

[List.of](#) y [Set.of](#) crean listas y conjuntos inmutables, respectivamente, a partir de los elementos proporcionados como argumentos.

[Stream.of](#) crea un Stream a partir de los elementos proporcionados como argumentos.

[Map.of](#) crea un mapa inmutable a partir de pares clave-valor proporcionados como argumentos.

```
List<String> lista = List.of("a", "b", "c");
Set<Integer> conjunto = Set.of(1, 2, 3);
```

```
Stream<String> stream = Stream.of("x", "y", "z");
Map<String, Integer> mapa = Map.of("uno", 1, "dos", 2, "tres", 3);
```

- **List.copyOf & Set.copyOf & Map.copyOf**

Estos métodos se introdujeron en Java 10 y permiten crear colecciones inmutables a partir de colecciones existentes.

[List.copyOf](#) y [Set.copyOf](#) crean copias inmutables de listas y conjuntos existentes, respectivamente.

[Map.copyOf](#) crea una copia inmutable de un mapa existente.

```
List<String> lista = Arrays.asList("a", "b", "c");
List<String> listaInmutable = List.copyOf(lista);

Set<Integer> conjunto = new HashSet<>(Arrays.asList(1, 2, 3));
Set<Integer> conjuntoInmutable = Set.copyOf(conjunto);

Map<String, Integer> mapa = new HashMap<>();
mapa.put("uno", 1);
mapa.put("dos", 2);
Map<String, Integer> mapaInmutable = Map.copyOf(mapa);
```

- **toUnmodifiableList(), toUnmodifiableSet() y toUnmodifiableMap()**

Estos métodos son proporcionados por la interfaz Collectors en Java para convertir colecciones mutables en inmutables.

[toUnmodifiableList\(\)](#) y [toUnmodifiableSet\(\)](#) convierten un Stream en una lista o un conjunto inmutable, respectivamente.

[toUnmodifiableMap\(\)](#) convierte un Stream de pares clave-valor en un mapa inmutable.

```
List<String> lista = Arrays.asList("a", "b", "c");
List<String> listaInmutable = lista.stream().collect(Collectors.toUnmodifiableList());

Set<Integer> conjunto = new HashSet<>(Arrays.asList(1, 2, 3));
Set<Integer> conjuntoInmutable = conjunto.stream().collect(Collectors.toUnmodifiableSet());

Map<String, Integer> mapa = new HashMap<>();
mapa.put("uno", 1);
mapa.put("dos", 2);
Map<String, Integer> mapaInmutable = mapa.entrySet().stream()
```

```
.collect(Collectors.toUnmodifiableMap(Map.Entry::getKey, Map.Entry::getValue));
```

- **Nuevos métodos de Streams**

- `takeWhile()`, `dropWhile()`, `iterate()` y `ofNullable()`

[takeWhile y dropWhile](#):

Estos métodos permiten tomar o descartar elementos de un Stream mientras se cumple una condición dada.

[takeWhile](#) toma elementos del principio del Stream hasta que la condición especificada ya no se cumple.

[dropWhile](#) descarta elementos del principio del Stream hasta que la condición especificada ya no se cumple.

```
// takeWhile
Stream<Integer> numeros = Stream.of(1, 2, 3, 4, 5, 6);
numeros.takeWhile(n -> n < 4).forEach(System.out::println); // Imprime 1, 2, 3

// dropWhile
Stream<Integer> numeros2 = Stream.of(1, 2, 3, 4, 5, 6);
numeros2.dropWhile(n -> n < 4).forEach(System.out::println); // Imprime 4, 5, 6
```

[ofNullable](#):

Este método estático en la interfaz Stream permite crear un Stream que puede contener cero o un solo elemento no nulo.

Es útil para crear flujos a partir de valores que pueden ser nulos, sin incluir los valores nulos en el flujo resultante.

```
// ofNullable
Stream<String> stream = Stream.ofNullable("foo");
stream.forEach(System.out::println); // Imprime "foo"

Stream<String> stream2 = Stream.ofNullable(null);
stream2.forEach(System.out::println); // No imprime nada (Stream vacío)
```

[iterate con predicado de término](#):

El método iterate en la interfaz Stream se mejoró para admitir un predicado de término que especifica cuándo detener la iteración.

Anteriormente, iterate generaba un Stream infinito, pero con el predicado de término, ahora se puede especificar una condición para terminar la iteración.

```
// iterate con predicado de término
Stream.iterate(0, n -> n < 10, n -> n + 2).forEach(System.out::println); // Imprime 0, 2, 4, 6, 8
```

toList_toSet_toMap en Collectors:

Se agregaron métodos convenientes a la interfaz Collectors para crear colecciones directamente desde un Stream.

toList crea una lista a partir de un Stream.

toSet crea un conjunto a partir de un Stream.

toMap crea un mapa a partir de un Stream, permitiendo especificar cómo manejar duplicados y cómo mapear claves y valores.

```
// toList
List<Integer> lista = Stream.of(1, 2, 3).collect(Collectors.toList());
System.out.println(lista); // Imprime [1, 2, 3]

// toSet
Set<Integer> conjunto = Stream.of(1, 2, 2, 3, 3).collect(Collectors.toSet());
System.out.println(conjunto); // Imprime [1, 2, 3]

// toMap
Stream<String> stream = Stream.of("uno", "dos", "tres");
Map<String, Integer> mapa = stream.collect(Collectors.toMap(s -> s, String::length));
System.out.println(mapa); // Imprime {uno=3, dos=3, tres=4}
```

■ Nuevos métodos de Optional

- ifPresentOrElse(), or() y stream()

ifPresentOrElse:

Este método ejecuta una acción si el Optional contiene un valor y ejecuta una acción alternativa si está vacío.

Ayuda a evitar la necesidad de usar isPresent() seguido de orElse() para ejecutar acciones diferentes según si el Optional está presente o no.

or:

Este método devuelve el valor actual del Optional si está presente, o el valor de otro Optional si está vacío.

Es útil para proporcionar un valor de respaldo o predeterminado si el Optional original está vacío.

stream:

Este método devuelve un Stream que contiene cero o un solo elemento si el Optional está vacío o presente, respectivamente.

Permite integrar fácilmente Optional con flujos de datos y operaciones de Streams.

```
// ifPresentOrElse
Optional<String> optional1 = Optional.of("Hola");
optional1.ifPresentOrElse(
    valor -> System.out.println("El valor es: " + valor), // Acción si presente
    () -> System.out.println("El valor no está presente") // Acción si vacío
);

Optional<String> optional2 = Optional.empty();
optional2.ifPresentOrElse(
    valor -> System.out.println("El valor es: " + valor), // Acción si presente
    () -> System.out.println("El valor no está presente") // Acción si vacío
);

// or
Optional<String> optional3 = Optional.empty();
Optional<String> backup = Optional.of("Respuesta de respaldo");
System.out.println(optional3.or(() -> backup)); // Imprime "Respuesta de respaldo"

// stream
Optional<String> optional4 = Optional.of("Uno");
Stream<String> stream1 = optional4.stream();
stream1.forEach(System.out::println); // Imprime "Uno"

Optional<String> optional5 = Optional.empty();
Stream<String> stream2 = optional5.stream();
stream2.forEach(System.out::println); // No imprime nada (Stream vacío)
```

■ **Métodos privados en Interfaces**

En Java 9 se introdujo la posibilidad de definir métodos privados en interfaces. Antes de Java 9, las interfaces solo podían contener métodos públicos abstractos, lo que limitaba la capacidad de reutilizar

lógica entre métodos dentro de la interfaz sin necesidad de implementarla en todas las clases que implementan la interfaz. La introducción de métodos privados en interfaces permite definir métodos auxiliares que son específicos de una interfaz sin exponerlos fuera de la interfaz misma.

```
public interface OperacionesMatematicas {  
    // Método público abstracto (por defecto)  
    int sumar(int a, int b);  
  
    // Método privado para verificar si un número es positivo  
    private boolean esPositivo(int num) {  
        return num > 0;  
    }  
  
    // Método público que utiliza el método privado  
    default boolean esSumaPositiva(int a, int b) {  
        return esPositivo(sumar(a, b));  
    }  
}
```

■ Jshell

JShell es una herramienta de línea de comandos introducida en Java SE 9. Es un entorno de desarrollo interactivo (REPL, por sus siglas en inglés: Read-Evaluate-Print Loop) que permite a los desarrolladores escribir, probar y depurar fragmentos de código Java de manera rápida y sencilla sin necesidad de crear un proyecto completo o un archivo de clase.

Con JShell, los usuarios pueden ingresar expresiones, declaraciones y bloques de código Java de forma interactiva y obtener resultados inmediatos. Esto es especialmente útil para experimentar con nuevas ideas, realizar pruebas rápidas, explorar APIs y aprender Java de manera interactiva.

Algunas características clave de JShell incluyen la capacidad de autocompletar, historial de comandos, retroceso y adelanto, así como la capacidad de definir, editar y redefinir variables y métodos sobre la marcha.

En resumen, JShell proporciona un entorno de programación interactivo y dinámico que complementa el proceso de desarrollo de software en Java, haciendo que la exploración y experimentación con código Java sea más accesible y eficiente.

```
jshell> /help intro
intro
The jshell tool allows you to execute Java code, getting immediate results.
You can enter a Java definition (variable, method, class, etc), like: int x = 8
or a Java expression, like: x + x
or a Java statement or import.
These little chunks of Java code are called 'snippets'.
There are also jshell commands that allow you to understand and
control what you are doing, like: /list
For a list of commands: /help
jshell>
```

■ Nuevo Garbage Collector

En Java 11, se introdujo un nuevo recolector de basura (garbage collector) llamado "Epsilon". Sin embargo, es importante señalar que Epsilon no es un recolector de basura convencional en el sentido de que no realiza la recolección de basura propiamente dicha. En cambio, Epsilon es un recolector de basura pasivo, diseñado para ser utilizado en situaciones donde no es necesaria la gestión automática de la memoria, es decir, en escenarios donde se desee deshabilitar completamente la recolección de basura.

El propósito principal de Epsilon es proporcionar una opción para casos de uso específicos en los que la recolección de basura puede ser innecesaria o incluso contraproducente. Por ejemplo, en aplicaciones que requieren tiempos de ejecución extremadamente predecibles o que gestionan su propia asignación y liberación de memoria de manera eficiente, como ciertas aplicaciones de alto rendimiento o sistemas de tiempo real.

Epsilon está diseñado para ser un recolector de basura extremadamente ligero y de bajo costo en términos de uso de CPU y memoria. Al no realizar ninguna actividad de recolección de basura, evita cualquier posible interrupción en la ejecución de la aplicación debido a la recolección de basura.

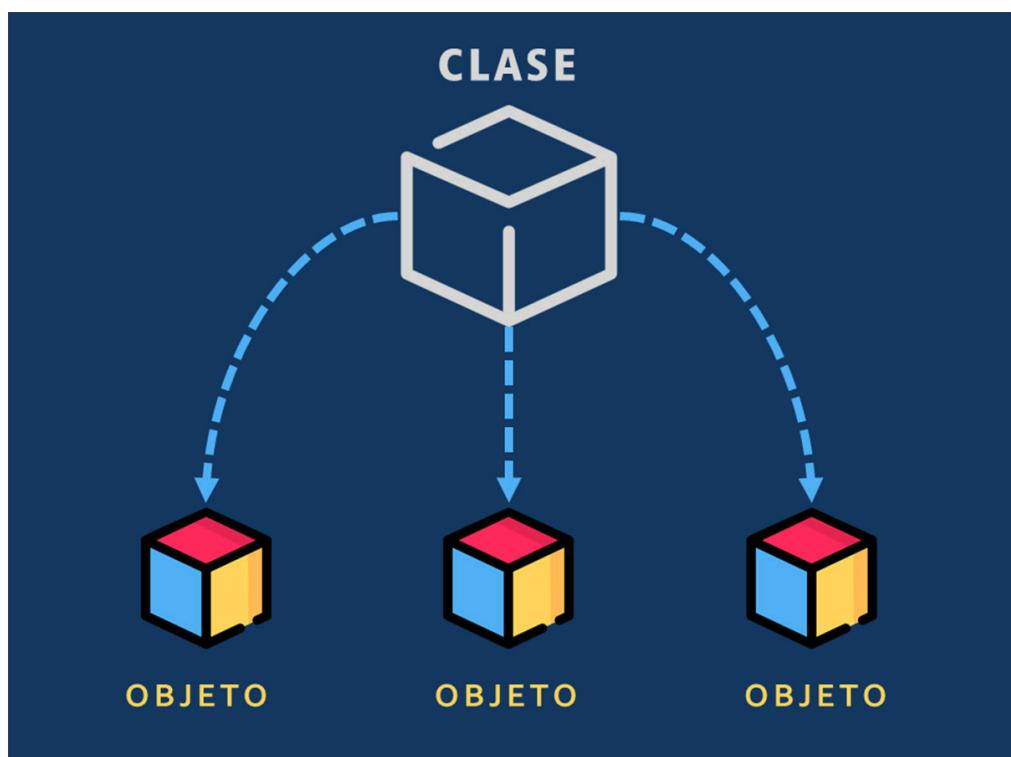
Es importante tener en cuenta que Epsilon no es adecuado para la mayoría de las aplicaciones Java convencionales y su uso está dirigido a casos muy específicos donde la desactivación completa de la recolección de basura es una opción viable y deseada.



- **Java en general**

- **Qué es POO**

La Programación Orientada a Objetos (POO) es un paradigma de programación que se basa en el concepto de "objetos". En la POO, un objeto es una entidad que combina datos (también conocidos como atributos o propiedades) y funciones (también conocidas como métodos o comportamientos) que operan en esos datos. Los objetos interactúan entre sí a través de mensajes, lo que permite que los programas se organicen de manera modular y se abstraigan los detalles de implementación.



- **Objeto**

Un objeto es una instancia concreta de una clase en la programación orientada a objetos. Representa una entidad del mundo real y puede tener atributos (datos) y métodos (comportamiento) asociados que operan en esos datos.

Por ejemplo, si tienes una clase llamada "Coche", un objeto de esa clase podría representar un coche específico, con sus propias características y comportamientos, como la marca, el modelo, el color y la capacidad para acelerar y frenar.

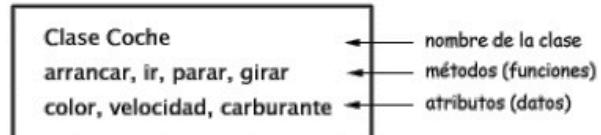
- **Clase**

Una clase es un plano o una plantilla que define la estructura y el comportamiento de los objetos. Define los atributos y los métodos que los objetos creados a partir de esa clase tendrán.

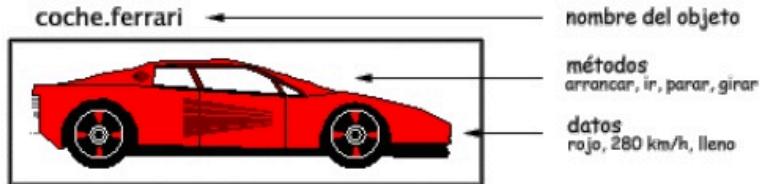
Siguiendo el ejemplo anterior, la clase "Coche" podría definir los atributos como "marca", "modelo" y "color", y los métodos como "acelerar" y "frenar". Los objetos de la clase "Coche" tomarían su forma y funcionalidad a partir de esta plantilla.

EJEMPLO DE CLASES Y OBJETOS

Clase:
Coche



♦ **Objeto:** *Ferrari*

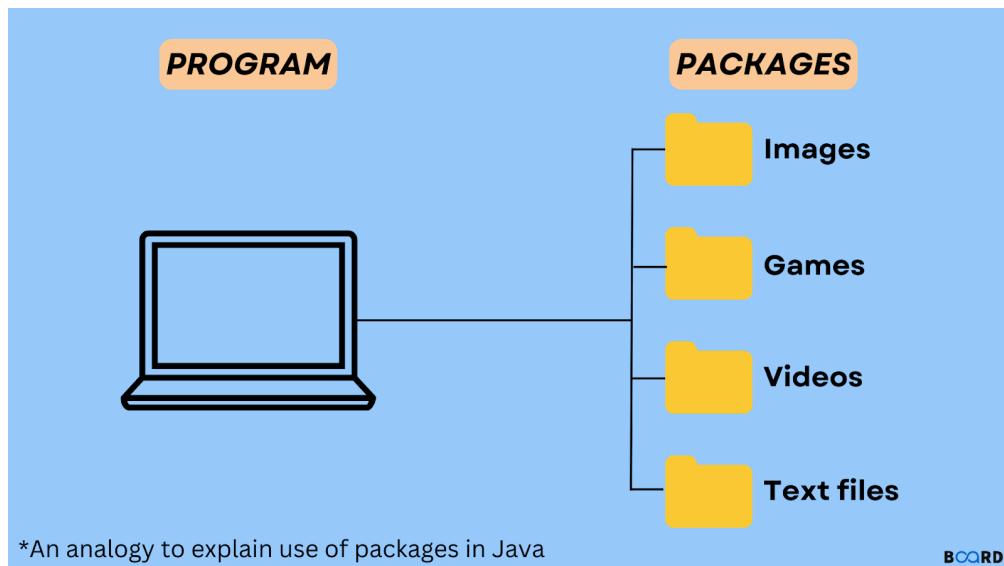


- **Paquetes**

Un paquete es una forma de organizar clases y otros elementos en un programa Java (o en otros lenguajes de programación que admitan paquetes

o módulos similares). Los paquetes proporcionan una estructura jerárquica para organizar y agrupar clases relacionadas.

Los paquetes son útiles para evitar conflictos de nombres y para mantener un código bien organizado y modular. Por ejemplo, un programa Java puede tener un paquete llamado "com.miempresa.vehiculos" que contenga clases relacionadas con vehículos, como "Coche", "Camión" y "Motocicleta".



NOTAS:

Un objeto es una instancia específica de una clase, con sus propios datos y comportamientos.

Una clase es una plantilla que define la estructura y el comportamiento de los objetos.

Un paquete es una forma de organizar y agrupar clases relacionadas en un programa.

- **Modificadores de acceso**

En programación, un modificador de acceso es una palabra clave que se utiliza para especificar el nivel de visibilidad o acceso que tienen los miembros (atributos o métodos) de una clase o una interfaz en un programa. Los modificadores de acceso controlan quién puede acceder a los miembros y en qué contexto se pueden utilizar.

- **private**

Los miembros marcados como private son accesibles sólo desde dentro de la misma clase. No se pueden acceder desde clases derivadas, clases en el mismo paquete ni desde clases externas.

Este es el nivel de visibilidad más restrictivo y se utiliza para ocultar los detalles internos de una clase.

■ **protected**

Los miembros marcados como protected son accesibles desde dentro de la clase misma, en clases derivadas (herencia) y en clases dentro del mismo paquete, pero no desde clases fuera del paquete.

Se utiliza para permitir que las clases derivadas accedan a ciertos miembros y, al mismo tiempo, limitar el acceso desde clases externas.

■ **public**

Los miembros marcados como public son accesibles desde cualquier lugar, ya sea dentro de la clase misma, en clases derivadas (herencia), en otras clases dentro del mismo paquete o en clases fuera del paquete.

Este es el nivel de visibilidad más permisivo y se utiliza para exponer una interfaz o funcionalidad que debe ser accesible desde cualquier parte del programa.

■ **default o sin modificador**

Los miembros sin un modificador explícito (también conocido como "default") son accesibles sólo desde dentro de la misma clase y desde clases dentro del mismo paquete.

Este nivel de visibilidad restringe el acceso a miembros a clases que están en el mismo paquete.

Modificador	Clase	Package	Subclase	Otros
public	✓	✓	✓	✓
protected	✓	✓	✓	•
default	✓	✓	•	•
private	✓	•	•	•

○ **Heap y Stack**

El **"heap"** es una región de memoria utilizada para almacenar objetos y datos de forma dinámica durante la ejecución de un programa. En otras palabras,

es donde se asignan y almacenan los objetos que se crean en tiempo de ejecución, como objetos de clases y arreglos.

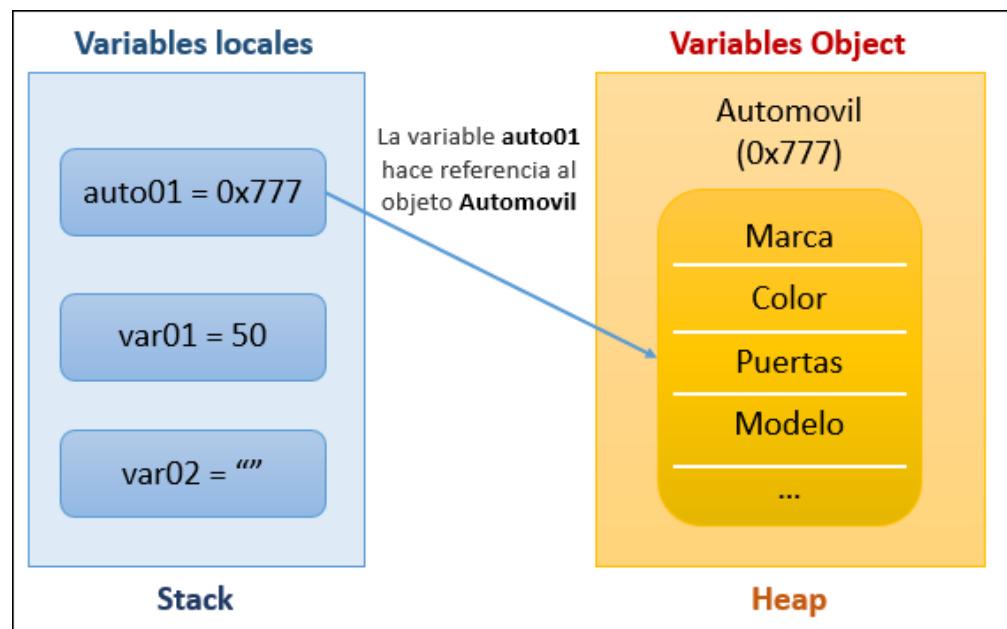
La memoria en el "**heap**" es gestionada por el sistema de gestión de memoria del lenguaje de programación o del sistema operativo. En lenguajes como Java, se encarga automáticamente de asignar y liberar memoria para los objetos cuando ya no son referenciados (lo que se conoce como recolección de basura).

Los objetos en el "**heap**" tienen una vida útil que puede ser más larga que el alcance de un solo método o función. Persisten en memoria mientras exista una referencia a ellos.

El "**stack**" (pila) es una región de memoria utilizada para almacenar datos de manera temporal durante la ejecución de un programa. Se utiliza principalmente para mantener información sobre la ejecución de funciones o métodos, como las variables locales, parámetros y registros de activación.

Cada vez que se llama a una función o método, se crea un nuevo "marco de pila" que contiene la información relevante para esa llamada, como las variables locales y los parámetros. Cuando la función retorna, su marco de pila se elimina.

El "**stack**" es una estructura de datos de tipo LIFO (último en entrar, primero en salir), lo que significa que la última función llamada es la primera en salir de la pila cuando se completa su ejecución.



NOTAS:

En resumen, el "heap" y el "stack" son dos áreas de la memoria con propósitos diferentes:

El "heap" se utiliza para almacenar objetos y datos dinámicos con una vida útil más larga y es gestionado automáticamente por el sistema.

El "stack" se utiliza para administrar la ejecución de funciones y métodos, y almacena datos de manera temporal y eficiente. Su estructura de datos es una pila que se crea y destruye de manera automática durante la ejecución del programa.

- **Garbage Collector**

El "Garbage Collector" (recolector de basura) es un componente fundamental de muchos lenguajes de programación modernos, incluidos Java, C#, y otros, que se encarga de gestionar la memoria y liberar recursos automáticamente al eliminar objetos que ya no son accesibles o referenciados por el programa. La función principal del Garbage Collector es evitar las fugas de memoria y garantizar que la memoria se utilice eficientemente.



- **Serial GC**

El Garbage Collector (GC) Serial es uno de los algoritmos de recolección de basura disponibles en Java. Este GC utiliza un solo hilo para realizar la recolección de basura, lo que significa que detiene todas las operaciones de la aplicación mientras recolecta basura. Es especialmente adecuado para aplicaciones de tamaño pequeño a mediano, o aplicaciones que se ejecutan en entornos con recursos limitados, ya que utiliza menos recursos de CPU y memoria en comparación con otros GC más complejos.

El proceso de recolección de basura en el GC Serial sigue un enfoque secuencial, donde se realiza una parada completa de la aplicación para recolectar y liberar la memoria no utilizada. Esto puede causar

pausas prolongadas en la aplicación, lo que puede ser un inconveniente en aplicaciones de gran escala que requieren una alta capacidad de respuesta.

Aunque el GC Serial puede ser útil en ciertos escenarios, como aplicaciones de escritorio simples o en dispositivos con recursos limitados, generalmente se prefiere en aplicaciones de producción el uso de GC más avanzados y eficientes, como el Parallel GC o el Garbage-First (G1) GC, que ofrecen una mejor gestión de la memoria y tiempos de pausa más cortos en la recolección de basura.

■ **Parallel GC**

El Parallel Garbage Collector (GC) es un algoritmo de recolección de basura disponible en Java que se enfoca en aprovechar múltiples núcleos de CPU para realizar la recolección de basura de manera más eficiente. A diferencia del Garbage Collector Serial, que utiliza un solo hilo, el Parallel GC utiliza varios hilos para realizar la recolección de basura de manera paralela, lo que puede reducir significativamente el tiempo total de recolección de basura.

El proceso de recolección de basura en el Parallel GC sigue un enfoque paralelo, donde se divide el trabajo de recolección de basura en varias tareas más pequeñas y se ejecuta en múltiples hilos de CPU de forma simultánea. Esto permite aprovechar al máximo los recursos de hardware disponibles y acelerar la recolección de basura, lo que resulta en tiempos de pausa más cortos para la aplicación.

El Parallel GC es especialmente adecuado para aplicaciones que requieren un alto rendimiento y una capacidad de respuesta rápida, ya que minimiza el tiempo de inactividad de la aplicación debido a la recolección de basura. Sin embargo, puede consumir más recursos de CPU y memoria en comparación con el GC Serial, por lo que es importante evaluar y ajustar la configuración del GC según las necesidades específicas de la aplicación y el entorno de ejecución.

■ **Concurrent Mark-Sweep GC**

El Concurrent Mark-Sweep (CMS) Garbage Collector es un algoritmo de recolección de basura disponible en Java que se enfoca en reducir los tiempos de pausa de la aplicación al realizar la recolección de basura de manera concurrente, lo que significa que intenta minimizar el tiempo en el que la aplicación se detiene debido a la recolección de basura. A diferencia de otros algoritmos que pueden detener por completo la aplicación durante la recolección de basura, el CMS GC intenta realizar gran parte de la recolección de basura mientras la aplicación sigue en ejecución.

El proceso de recolección de basura en el CMS GC consta de dos fases principales: la fase de marcado (mark) y la fase de barrido (sweep). Durante la fase de marcado, se identifican y marcan los objetos que están en uso y los que pueden ser liberados. Esta fase se realiza de manera concurrente con la ejecución de la aplicación, lo que minimiza las interrupciones. Luego, durante la fase de barrido, se eliminan los objetos marcados como basura de manera rápida.

Aunque el CMS GC puede reducir los tiempos de pausa de la aplicación, tiene algunas limitaciones. Por ejemplo, puede generar fragmentación de la memoria y puede no ser tan eficiente en sistemas con una alta tasa de creación y eliminación de objetos. Además, la recolección de basura concurrente puede consumir recursos adicionales de CPU y memoria, lo que puede afectar el rendimiento general de la aplicación.

En resumen, el CMS GC es una opción adecuada para aplicaciones que requieren tiempos de pausa mínimos y una alta capacidad de respuesta, pero es importante considerar sus limitaciones y realizar pruebas exhaustivas para determinar si es la opción óptima para una aplicación específica.

■ G1 GC

El Garbage-First (G1) Garbage Collector es un algoritmo de recolección de basura disponible en Java que está diseñado para proporcionar un equilibrio entre tiempos de pausa predecibles y una alta eficiencia de recolección de basura. A diferencia de otros algoritmos, como el Concurrent Mark-Sweep (CMS) GC, que se enfocan en minimizar los tiempos de pausa, el G1 GC utiliza una estrategia de recolección de basura adaptativa que prioriza la recolección de basura en áreas de la memoria donde hay una alta concentración de objetos no utilizados.

El proceso de recolección de basura en el G1 GC se divide en varias etapas. En primer lugar, el G1 GC divide el montón de memoria en regiones más pequeñas y recopila datos sobre el uso de cada región. Luego, utiliza esta información para determinar qué regiones deben ser recolectadas de basura en cada ciclo de recolección.

El G1 GC utiliza una técnica llamada recolección de basura por regiones, donde recolecta basura de un subconjunto de regiones en cada ciclo de recolección. Esto permite que el G1 GC tenga tiempos de pausa predecibles, ya que puede limitar la cantidad de memoria que necesita ser revisada en cada ciclo de recolección.

Además de proporcionar tiempos de pausa predecibles, el G1 GC también está diseñado para adaptarse dinámicamente a las

características de la aplicación y al comportamiento de la carga de trabajo. Por ejemplo, puede ajustar dinámicamente la cantidad de regiones dedicadas a la recolección de basura en función de la cantidad de objetos no utilizados en el montón de memoria.

En resumen, el G1 GC es una opción adecuada para aplicaciones que requieren tiempos de pausa predecibles y una alta eficiencia de recolección de basura. Sin embargo, puede consumir más recursos de CPU y memoria en comparación con otros algoritmos de recolección de basura, por lo que es importante evaluar y ajustar la configuración del G1 GC según las necesidades específicas de la aplicación y el entorno de ejecución.

- **Pilares de POO**

- **Polimorfismo**

El polimorfismo permite que objetos de diferentes clases respondan a mensajes de la misma manera. Esto se logra mediante el uso de interfaces o clases base comunes y la implementación de métodos con el mismo nombre en diferentes clases.

- **Herencia**

La herencia permite que una clase (llamada subclase o clase derivada) herede propiedades y métodos de otra clase (llamada superclase o clase base). Esto permite la reutilización de código y la creación de relaciones jerárquicas entre clases.

- **Abstracción**

La abstracción es el proceso de identificar las características esenciales de un objeto y omitir los detalles innecesarios. En programación, una clase representa una abstracción de un objeto o entidad del mundo real, y contiene propiedades y métodos que definen su comportamiento.

- **Encapsulamiento**

La encapsulación es el principio de ocultar los detalles internos de un objeto y proporcionar una interfaz pública para interactuar con él. En POO, las propiedades de un objeto (a menudo llamadas atributos) se definen como privadas y se acceden a través de métodos públicos (a menudo llamados "getters" y "setters") para mantener la integridad de los datos y controlar el acceso.



o Clases Abstractas

En la programación orientada a objetos (POO), una clase abstracta es una clase que no puede ser instanciada directamente, es decir, no se pueden crear objetos directamente a partir de una clase abstracta. En cambio, una clase abstracta se utiliza como una plantilla o base para crear subclases (clases hijas) que heredan su estructura y pueden implementar sus propios comportamientos específicos.

```
// Clase abstracta Figura
abstract class Figura {
    // Método abstracto para calcular el área
    public abstract double calcularArea();
}

// Clase concreta que extiende la clase abstracta
class Circulo extends Figura {
    private double radio;

    public Circulo(double radio) {
        this.radio = radio;
    }
}
```

```

@Override
public double calcularArea() {
    // Área del círculo: π * radio^2
    return Math.PI * radio * radio;
}

// Clase concreta que extiende la clase abstracta
class Cuadrado extends Figura {
    private double lado;

    public Cuadrado(double lado) {
        this.lado = lado;
    }

    @Override
    public double calcularArea() {
        // Área del cuadrado: Lado^2
        return lado * lado;
    }
}

public class Main {
    public static void main(String[] args) {
        Circulo circulo = new Circulo(5.0);
        System.out.println("Área del círculo: " + circulo.calcularArea());

        Cuadrado cuadrado = new Cuadrado(4.0);
        System.out.println("Área del cuadrado: " + cuadrado.calcularArea());
    }
}

```

- **Clases Genéricas**

Las clases genéricas, a menudo denominadas simplemente "genéricos", son una característica de muchos lenguajes de programación, incluidos Java, C#, C++, y otros, que permiten la creación de clases y métodos que pueden trabajar con tipos de datos específicos que se determinan en tiempo de compilación. Esto permite escribir código más flexible y reutilizable al permitir que las clases y los métodos trabajen con diferentes tipos de datos de manera segura.

```

// Clase genérica Pila
class Pila<T> {
    private T[] elementos;

```

```

private int tope;

public Pila(int capacidad) {
    elementos = (T[]) new Object[capacidad];
    tope = -1;
}

public void apilar(T elemento) {
    elementos[++tope] = elemento;
}

public T desapilar() {
    if (tope == -1) {
        throw new IllegalStateException("La pila está vacía");
    }
    return elementos[tope--];
}

public class Main {
    public static void main(String[] args) {
        Pila<Integer> pilaEnteros = new Pila<>(5);
        pilaEnteros.apilar(1);
        pilaEnteros.apilar(2);
        pilaEnteros.apilar(3);

        System.out.println("Elementos desapilados:");
        System.out.println(pilaEnteros.desapilar());
        System.out.println(pilaEnteros.desapilar());
        System.out.println(pilaEnteros.desapilar());
    }
}

```

- Colecciones
 - Arrays

Los arrays son estructuras de datos que contienen elementos del mismo tipo, organizados en una secuencia indexada. Son de tamaño fijo y pueden contener elementos primitivos o referencias a objetos.

```

// Declaración e inicialización de un array de enteros
int[] numeros = new int[5];
numeros[0] = 10;
numeros[1] = 20;

```

■ Clase Wrapper

Las clases wrapper en Java son clases que encapsulan tipos de datos primitivos, permitiendo tratarlos como objetos. Esto facilita su uso en contextos donde se requieren objetos en lugar de tipos primitivos.

```
// Declaración e inicialización de un Integer (clase wrapper para int)
Integer numero = 10;
```

■ List

La interfaz List en Java es una colección ordenada de elementos que pueden contener duplicados. Proporciona métodos para agregar, eliminar, acceder y modificar elementos en función de su posición.

```
// Declaración e inicialización de una lista de cadenas
List<String> lista = new ArrayList<>();
lista.add("Uno");
lista.add("Dos");
```

■ Map

La interfaz Map en Java representa una colección de pares clave-valor, donde cada clave está asociada a un único valor. No puede contener claves duplicadas y cada clave puede estar asociada a un solo valor.

```
// Declaración e inicialización de un mapa de cadenas a enteros
Map<String, Integer> mapa = new HashMap<>();
mapa.put("Uno", 1);
mapa.put("Dos", 2);
```

■ Stack

La clase Stack en Java representa una pila, una estructura de datos de tipo LIFO (Last In, First Out), donde los elementos se agregan y eliminan desde la parte superior de la pila.

```
// Declaración e inicialización de una pila de enteros
Stack<Integer> pila = new Stack<>();
pila.push(10);
pila.push(20);
```

■ Queue

La interfaz Queue en Java representa una cola, una estructura de datos de tipo FIFO (First In, First Out), donde los elementos se agregan al final de la cola y se eliminan desde el principio.

```
// Declaración e inicialización de una cola de cadenas
Queue<String> cola = new LinkedList<>();
cola.offer("Uno");
cola.offer("Dos");
```

- **Manejo de Excepciones**

- **Try, Catch, Finally**

Los bloques try-catch se utilizan para manejar excepciones. En Java 8, puedes usar múltiples bloques catch en una sola estructura try para manejar diferentes tipos de excepciones de manera más concisa.

La cláusula finally se utiliza para ejecutar código que debe ejecutarse independientemente de si se produjo una excepción o no. Esto se utiliza para liberar recursos o realizar acciones de limpieza.

```
public static void main(String[] args) {
    FileReader reader = null;
    try {
        File file = new File("archivo.txt");
        reader = new FileReader(file);
        char[] buffer = new char[1024];
        int length;
        while ((length = reader.read(buffer)) != -1) {
            System.out.println(new String(buffer, 0, length));
        }
    } catch (IOException e) {
        System.err.println("Error de lectura: " + e.getMessage());
    } finally {
        try {
            if (reader != null) {
                reader.close();
            }
        } catch (IOException e) {
            System.err.println("Error al cerrar el lector: " + e.getMessage());
        }
    }
}
```

■ Throws

En Java, las excepciones se pueden dividir en dos tipos: excepciones verificadas (checked exceptions) y excepciones no verificadas (unchecked exceptions). Las excepciones verificadas deben ser declaradas en la firma del método o manejadas dentro del método utilizando bloques try-catch.

```
public static void main(String[] args) {
    try {
        int resultado = dividir(10, 0);
        System.out.println("Resultado de la división: " + resultado);
    } catch (ArithmeticeException e) {
        System.err.println("Error: " + e.getMessage());
    }
}

public static int dividir(int dividendo, int divisor) throws ArithmeticeException {
    if (divisor == 0) {
        throw new ArithmeticeException("Divisor no puede ser cero");
    }
    return dividendo / divisor;
}
```

■ Try-With-Resources

Java 7 introdujo el try-with-resources, una forma más segura y conveniente de manejar recursos que deben ser cerrados después de su uso, como flujos de archivos o conexiones de bases de datos. En Java 8, esta característica se mejoró aún más.

```
public static void main(String[] args) {
    try (BufferedReader reader = new BufferedReader(new FileReader("archivo.txt"))) {
        String linea;
        while ((linea = reader.readLine()) != null) {
            System.out.println(linea);
        }
    } catch (IOException e) {
        System.err.println("Error de lectura: " + e.getMessage());
    }
}
```

■ Crear una excepción

Java permite definir tus propias excepciones personalizadas extendiendo la clase `Exception` o `RuntimeException`. Esto es útil cuando deseas crear excepciones específicas para tu aplicación.

```

// Definición de una excepción personalizada
class MiExcepcion extends Exception {
    public MiExcepcion(String mensaje) {
        super(mensaje);
    }
}

public class Main {
    // Método que Lanza la excepción personalizada
    public static void lanzarExcepcion() throws MiExcepcion {
        throw new MiExcepcion("¡Esta es una excepción personalizada!");
    }

    public static void main(String[] args) {
        try {
            lanzarExcepcion();
        } catch (MiExcepcion e) {
            System.err.println("Se produjo una excepción: " + e.getMessage());
        }
    }
}

```

- o **Overloading, Overriding**

Sobrecarga de métodos (**method overloading**) implica definir varios métodos con el mismo nombre en la misma clase pero con diferentes listas de parámetros.

```

// Clase con métodos sobrecargados
class Calculadora {
    public int sumar(int a, int b) {
        return a + b;
    }

    public double sumar(double a, double b) {
        return a + b;
    }

    public int sumar(int a, int b, int c) {
        return a + b + c;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculadora calc = new Calculadora();
    }
}

```

```

        System.out.println("Suma de enteros: " + calc.sumar(2, 3));      // Salida: 5
        System.out.println("Suma de dobles: " + calc.sumar(2.5, 3.5));    // Salida: 6.0
        System.out.println("Suma de tres enteros: " + calc.sumar(2, 3, 4)); // Salida: 9
    }
}

```

Sobrescritura de métodos (**method overriding**) permite que una subclase proporcione su propia implementación de un método que ya está definido en su superclase, con la condición de que el método en la subclase tenga la misma firma que el método en la superclase.

```

// Clase base
class Animal {
    public void hacerSonido() {
        System.out.println("El animal hace algún sonido.");
    }
}

// Clase derivada que sobrescribe el método hacerSonido()
class Perro extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("El perro ladra.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Animal();
        animal.hacerSonido(); // Salida: El animal hace algún sonido.

        Perro perro = new Perro();
        perro.hacerSonido(); // Salida: El perro Ladra.
    }
}

```

- **Nested Class**

Una "nested class" (clase anidada) en Java es una clase definida dentro de otra clase. Esto significa que la clase anidada está completamente encapsulada dentro de la clase exterior y puede acceder a los miembros (campos y métodos) de la clase externa, incluso si son privados.

■ Static Nested Class

Una clase estática anidada se declara con la palabra clave static y se comporta de manera similar a una clase normal, excepto que está dentro de otra clase.

```
public class OuterClass {  
    static class NestedStaticClass {  
        // Contenido de la clase anidada  
    }  
}
```

■ Non-Static Nested Class / Inner Class

Una clase interna (inner class) es una clase no estática declarada dentro de otra clase. Tiene acceso a todos los miembros de la clase exterior, incluso a los miembros privados.

```
public class OuterClass {  
    class InnerClass {  
        // Contenido de la clase interna  
    }  
}
```

- **Anonymous Inner Class**

Una clase interna anónima es una clase sin nombre que se declara e instancia al mismo tiempo. Se utiliza para implementar interfaces o clases abstractas de manera concisa.

```
public class OuterClass {  
    InterfaceAnonima objetoAnonimo = new InterfaceAnonima() {  
        // Implementación de La interfaz o clase abstracta  
    };  
}
```

- **Local Inner Class**

Una clase interna local es una clase definida dentro de un método o un bloque de código. Solo se puede acceder a ella dentro del alcance en el que se declara.

```
public class OuterClass {  
    void metodoExterno() {  
        class LocalClass {  
            // Contenido de la clase Local  
        }  
    }  
}
```

```
    }
}
```

NOTAS:

En resumen, las clases anidadas en Java son útiles para organizar el código y modularizar funcionalidades relacionadas. Permiten una mejor encapsulación y pueden mejorar la legibilidad y la mantenibilidad del código al agrupar las clases que tienen una relación semántica.

- **Operador InstanceOf**

En Java, instanceof es un operador utilizado para verificar si un objeto es una instancia de una clase o una subclase de esa clase. Proporciona una forma de determinar si un objeto es compatible con un tipo específico antes de realizar una conversión o una operación en él.

Aquí hay algunos puntos importantes sobre instanceof:

Retorna un booleano: El operador instanceof retorna true si el objeto es una instancia de la clase especificada o de una subclase de esa clase. De lo contrario, retorna false.

Compatibilidad con subclases: Si el objeto es una instancia de una subclase de la clase especificada, también se considera compatible con esa clase.

Comprobación de tipos: Se utiliza comúnmente para comprobar si un objeto es de un tipo específico antes de realizar una conversión de tipo o una operación que podría generar una excepción de tipo ClassCastException.

Manejo de polimorfismo: Es útil en situaciones de polimorfismo, donde un objeto puede ser de diferentes tipos en tiempo de ejecución y se necesita determinar su tipo real antes de realizar ciertas operaciones.

```
Object objeto = new String("Hola");

if (objeto instanceof String) {
    // El objeto es una instancia de la clase String
    String cadena = (String) objeto;
    System.out.println("La cadena es: " + cadena);
} else {
    System.out.println("El objeto no es una cadena");
}
```

- Java Reactivo
 - RxJava

RxJava es una biblioteca para la programación reactiva en Java que implementa el patrón Observable/Observer. Permite trabajar con flujos de datos asíncronos y eventos de una manera declarativa y compuesta. Aquí hay algunos conceptos clave sobre RxJava:

Observable: Un Observable emite una secuencia de eventos o datos a lo largo del tiempo. Puede emitir cero o más elementos, y puede finalizar con éxito o con error. Los observables pueden ser creados a partir de una variedad de fuentes, como listas, arrays, eventos de UI, etc.

Observer: Un Observer es el consumidor de los datos emitidos por un Observable. Puede recibir los elementos emitidos, manejar los errores que puedan ocurrir y ser notificado cuando el Observable ha terminado de emitir elementos.

Operadores: RxJava proporciona una amplia gama de operadores para transformar, filtrar, combinar y manipular flujos de datos. Estos operadores permiten realizar transformaciones y acciones sobre los datos emitidos por los observables de una manera declarativa y funcional.

Schedulers: RxJava permite controlar en qué subprocesos se ejecutan las operaciones de los observables y los observadores utilizando programación reactiva. Los Schedulers proporcionan un mecanismo para especificar en qué subproceso se realizarán operaciones como la suscripción, la emisión de elementos y el manejo de eventos.

Gestión de la concurrencia: RxJava ofrece operadores y herramientas para gestionar la concurrencia en flujos de datos, como la combinación de múltiples flujos, el manejo de errores, la tolerancia a fallos y el control de la velocidad de producción de datos (backpressure).

Composición y reutilización de lógica: La programación reactiva con RxJava permite componer y reutilizar lógica de manera modular, lo que facilita la construcción de aplicaciones robustas y mantenibles.

```
// Crear un observable que emite los números del 1 al 5
Observable<Integer> observable = Observable.range(1, 5);

// Suscribirse al observable y definir un Observer para manejar los elementos emitidos
observable.subscribe(new Observer<Integer>() {
```

```

@Override
public void onSubscribe(Disposable d) {
    // Método llamado al establecer la suscripción
}

@Override
public void onNext(Integer value) {
    // Método llamado cuando se emite un elemento
    System.out.println("Elemento emitido: " + value);
}

@Override
public void onError(Throwable e) {
    // Método llamado en caso de error
    System.err.println("Error: " + e.getMessage());
}

@Override
public void onComplete() {
    // Método llamado cuando el Observable ha terminado de emitir elementos
    System.out.println("Observable completado");
}
});

```

- Project Reactor
- Akka Streams
- Testing Java Reactivo
- R2DBC
- Backpressure
- Future & Promises
- Callbacks
- Completable Future

Spring Framework

- Spring Initializer
- Swagger, OpenApi
- Spring vs Spring Boot
- Bean
 - Dependency Injection / DI / Inyección de dependencia
 - Tipos de DI

```
// Clase ServicioMensaje que depende de MensajeService
@Component
class ServicioMensaje {
    private final MensajeService servicio;

    // Inyección de dependencia a través del constructor
    @Autowired
    public ServicioMensaje(MensajeService servicio) {
        this.servicio = servicio;
    }

    public void enviarNotificacion(String mensaje) {
        servicio.enviarMensaje(mensaje);
    }
}
```

```
// Clase ServicioMensaje que depende de MensajeService
@Component
class ServicioMensaje {
    // Inyección de dependencia a través del campo
    @Autowired
    private MensajeService servicio;

    public void enviarNotificacion(String mensaje) {
        servicio.enviarMensaje(mensaje);
    }
}
```

```
// Clase ServicioMensaje que depende de MensajeService
@Component
class ServicioMensaje {
    private MensajeService servicio;

    // Setter para la inyección de dependencia
    @Autowired
    public void setServicio(MensajeService servicio) {
        this.servicio = servicio;
    }

    public void enviarNotificacion(String mensaje) {
        servicio.enviarMensaje(mensaje);
    }
}
```

- Ciclo de vida de un Bean / Life-cycle Bean
 - Aware Interfaces
 - BeanPostProcessor
 - postProcessBeforeInitialization
 - postProcessAfterInitialization
 - @PostConstruct
 - InitializingBean
 - @PreDestroy
 - Disposable Bean
 - Orden de Ejecución del ciclo de vida de un bean
 - JSR-250
- Ambito de un Bean / @Scope
 - Singleton
 - Prototype
 - Request
 - Session
 - Application
 - WebSocket
- Explicit Bean Declaration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- Declaración explícita de beans -->
    <bean id="emailService" class="com.example.EmailService"/>
    <bean id="smsService" class="com.example.SMSService"/>

</beans>
```

```
// Clase de configuración de Spring
@Configuration
public class AppConfig {

    // Método que define un bean de EmailService
    @Bean
    public EmailService emailService() {
        return new EmailService();
    }

    // Método que define un bean de ServicioMensaje y realiza la inyección de dependencias de EmailService
    @Bean
    public ServicioMensaje servicioMensaje(EmailService emailService) {
        ServicioMensaje servicioMensaje = new ServicioMensaje();
        servicioMensaje.setEmailService(emailService);
        return servicioMensaje;
    }
}
```

```
public ServicioMensaje servicioMensaje() {  
    return new ServicioMensaje(emailService());  
}
```

- Ambigüedad de un Bean / `@Qualifier`
- `@Primary`
- `@Autowired`
 - En Listas
- **Stereotypes / Estereotipos**
 - `@Component`
 - `@Service`
 - `@Controller`
 - `@Repository`
- **Perfiles**
 - Perfiles por Default
- Cómo funciona el `@SpringBootApplication`
- **Properties in Spring**
 - Carga de properties / `@PropertySource`
- **Spring Expression Language / SpEL**
- **AOP / Programación orientada a aspectos**
 - crosscutting
 - Aspect
 - Joinpoint
 - Advice
 - Before
 - After returning
 - After throwing
 - After finally
 - Around
 - Pointcut
 - Target object
 - Proxy
 - Weaving
- **HTTP**
 - Verbos HTTP / Metodos HTTP
 - GET
 - HEAD
 - POST
 - PUT
 - PATCH
 - DELETE
 - TRACE
 - OPTIONS
 - CONNECT
 - URI
 - Schema
 - Host
 - Path
 - Petición HTTP
 - Método HTTP
 - URI
 - Headers

- **Body**
- **Respuesta HTTP**
 - Versión
 - Status code
 - Headers
 - Body
- **Connectionless and Stateless**
- **REST**
 - Resource
- **Status HTTP**
 - 1xx
 - 2xx
 - 200 - OK
 - 201 - CREATED
 - 204 - NOT CONTENT
 - 3xx
 - 4xx
 - 400 - BAD REQUEST
 - 401 - UNAUTHORIZED
 - 404 - NOT FOUND
 - 405 - METHOD NOT ALLOWED
 - 409 - CONFLICT
 - 429 - TOO MANY REQUESTS
 - 5xx
 - 500 - INTERNAL SERVER ERROR
 - 501 - NOT IMPLEMENTED
 - 502 - BAD GATEWAY
 - 503 - SERVICE UNAVAILABLE
- **Versionamiento de API**
 - Por URL
 - Por Header
 - Por Dominio
 - Por Request param
- **Paginación, filtros y ordenamiento con Query Params**
- **REST**
 - **Spring MVC**
 - Model
 - View
 - Controller
 - **Anotaciones**
 - Controller
 - RestController
 - RequestMapping
 - GetMapping
 - PostMapping
 - PutMapping
 - DeleteMapping
 - PathMapping

- **PathVariable**
 - **RequestParam**
 - **ResponseStatus**
 - **Service**
- **Otras clases útiles**
 - **ResponseEntity**
 - **HttpStatus**
 - **ResponseStatusException**
- **Spring Actuator**
 - **Configuración**
 - **Métricas**
 - **Timed**
 - **Prometheus**
 - **Grafana**
- **Spring Cache**
 - **Configuración**
 - **Uso**
 - **CacheEvict**
 - **Caché en memoria**
 - **Redis**
 - **Configuración**
 - **Uso**
- **Spring Security**
 - **Resuelve**
 - **Impersonators / Imitadores**
 - **Ser Alguien**
 - **Tener Algo**
 - **Saber Algo**
 - **Upgraders / Mejoradores**
 - **Eavesdropper / Espía**
 - **Autenticación en memoria**
 - **Manejo de Usuarios**
 - **UserDetails**
 - **GrantedAuthority**
 - **UserDetailsService**
 - **UserDetailsManager**
 - **JdbcUserDetailsManager**
 - **Protección de Contraseñas**
 - **PasswordEncoder**
 - **Spring Security Crypto**
 - **BCrypt**
 - **Autenticación**
 - **AuthenticationProvider**
 - **SecurityContext**
 - **DelegatingSecurityContextRunnable**
 - **DelegatingSecurityContextCallable**
 - **DelegatingSecurityContextExecutor**
 - **Restricción de Acceso**

- Authorities
- Roles
- Filters
- Protección
 - CSRF
 - CORS
 - Tokens
 - JWT
- OAuth2
- Spring Security Reactive
- Spring Data
 - Spring Java Persistence Api / Spring JPA
 - Diferencias con EJB
 - Diferencias con Hibernate
 - JDBC
 - JNDI
 - ORM
 - Transactional

Otros temas de entrevista

- **Microservicios**
 - Definición
 - Ventajas y desventajas de aplicar microservicios
 - Teorema Gold Hammer
 - Configuración mediante repositorio remoto
 - Patrones de diseño de microservicios
 - Saga
 - CQRS (Command and Query Responsibility Segregation)
 - Circuit Breaker
 - Chassis
 - Api Gateway
 - Service Discovery
 - Service Registry
 - Descomposición de negocio en Microservicio
 - Capacidades de negocio
 - Subdominios
 - etc...
- Docker
 - Definición
 - Cómo se genera una imagen
 - Que es un container
- Kubernetes
 - Definición
 - Relación con Docker
 - Componentes principales
 - Control Plane
 - Cluster
 - Nodo
 - Pod
 - Service
 - Container
 - Job
- Redis
 - Definición
 - Estructura key-value
 - Redis vs Hazelcast
- Kafka
 - Definición
 - Kafka vs RabbitMQ
 - Comunicacion Asincrona vs Sincrona
 - Conceptos Principales
 - Zookeeper
 - Topic
 - Productor/Consumidor
 - Partición
 - Offset
- Patrones de diseño de Software
 - Patrones creacionales

- Singleton
 - Factory
 - Abstract Factory
- Patrones estructurales
 - Adapter
 - Decorator
 - Proxy
- Patrones de comportamiento
 - Observer
 - Strategy
 - Command
- Principios SOLID
 - Acoplamiento
 - Cohesión
 - Principio de responsabilidad unica
 - Principio de abierto y cerrado
 - Principio de sustitución de Liskov
 - Principio de segregación de interfaces
 - Principio de inversión de dependencia
- Estilos arquitectónicos
 - Monolítico
 - Cliente Servidor
 - Peer-to-peer (P2P)
 - Arquitectura en Capas
 - Microkernel
 - Service-Oriented-Architecture (SOA)
 - Microservicios
 - Event Driven Architecture (EDA)
 - Representational State Transfer
 - Hexagonal Architecture
 - Clean Architecture
- Patrones arquitectónicos
 - Data Transfer Object (DTO)
 - Data Access Object
 - Polling
 - Webhook
 - Load Balance
 - Service Registry
 - Service Discovery
 - Api Gateway
 - Access Token
 - Single Sign On (inicio de sesión único)
 - Store and forward
 - Circuit Breaker
 - Log aggregation
- Teorías
 - Don't Repeat Yourself (DRY)
 - Separation of Concerns (SOC)

- **Inversion of control (IoC)**
- **Code Smell - Código espagueti**
- **Load Balance**
- **Fault Tolerance**
- **Escalabilidad**
- **Config Server**
- **Distributed Tracing**
- **Orquestación / Coreografía**
- **Semántica y taxonomía de apis**
- **Backpressure**
- **Cómo medir el nivel de madurez de una api rest**
- **Patrón Strangler**
- **Patrón sub/pub**
- **Enfoque y Metodologías de desarrollo**
 - **TDD - Test Driven Development**
 - **DDD - Domain Driven Design**
 - **BDD - Behavior Driven Development**
- **Devops**
 - **Definición**
 - **Jenkins**