Universidad Simón Bolivar
Lenguajes de Programacion
CI-3641

Jesus Gutierrez
20-10332
Prof. Ricardo Monascal
Sep - Dic 2024

Java - Tarea 1

1. Diga qué tipo de alcances y asociaciones posee, argumentando las ventajas y desventajas de la decisión tomada por los diseñadores del lenguaje, en el contexto de sus usuarios objetivos.

   Java es un lenguaje de proposito general el cual posee alcance estatico y posee ambas asociaciones (profunda y superficial).

   - Ventajas:

   - Portabilidad: Java es .^escribe una vez, ejecuta en cualquier lugar"(WORA). Gracias a la JVM (Máquina Virtual de Java), el código compilado puede ejecutarse en cualquier sistema que tenga una JVM instalada sin necesidad de recompilación.
   - Orientación a objetos: Java es un lenguaje puramente orientado a objetos, lo que facilita la organización del código, la reutilización y la creación de aplicaciones más modulares.
   - Seguridad: Java tiene un fuerte enfoque en la seguridad, con características como la verificación de tipos en tiempo de compilación y ejecución, la gestión de excepciones y un modelo de seguridad basado en permisos.
   - Gran comunidad: Java cuenta con una comunidad de desarrolladores muy grande y activa, lo que significa que hay una gran cantidad de recursos, bibliotecas y herramientas disponibles.
   - Robustez: Java incluye características como la gestión automática de memoria (garbage collection) y la prevención de punteros nulos, lo que ayuda a reducir los errores de programación.
   - Escalabilidad: Java es adecuado para desarrollar aplicaciones de gran escala y de alto rendimiento.
   - Multihilo: Java soporta la programación multihilo de forma nativa, lo que permite ejecutar múltiples tareas de forma concurrente.

   - Desventajas:

   - Consumo de Memoria: Java puede consumir más memoria en comparación con lenguajes como C o C++.
   - Verbosidad: En comparación con algunos lenguajes más modernos, Java puede ser más verboso, lo que puede llevar a un código más largo y menos conciso.
   - Rendimiento: Aunque Java ha mejorado significativamente en términos de rendimiento, en algunas situaciones puede ser más lento que lenguajes compilados directamente a código máquina, como C o C++.

- Inicio más lento: Las aplicaciones Java suelen tener un tiempo de inicio más largo debido a la necesidad de cargar la JVM.
- Tamaño de las aplicaciones: Las aplicaciones Java tienden a ser más grandes que las aplicaciones equivalentes en otros lenguajes, debido a la JVM y a las bibliotecas estándar.
- Curva de aprendizaje: Aunque Java es relativamente fácil de aprender, dominar todas sus características puede requerir un tiempo considerable.

II. Diga qué tipo de módulos ofrece (de tenerlos) y las diferentes formas de importar y exportar nombres.

Un módulo de Java es un conjunto de paquetes y tipos de Java específicos, incluidas clases, interfaces y mucho más, empaquetados con archivos de datos y recursos estáticos. Los módulos son la principal novedad de Java 9. Un módulo agrupa código y recursos como los JARs tradicionales, pero añade además un descriptor que restringe el acceso a sus paquetes, y describe sus dependencias. Los distintos tipos de modulo que ofrece Java son:

- Módulos del sistema: Son los módulos que forman parte del JRE (Java Runtime Environment) y son esenciales para la ejecución de cualquier aplicación Java. Ejemplos:

```
java.base, java.sql, java.desktop
```

- Módulos automáticos (automatic modules): Estos son módulos que se crean a partir de JARs que no contienen un descriptor de módulo **module-info.java**, pero se utilizan en el sistema de módulos.
- Módulos explícitos (named modules): Estos módulos tienen un descriptor de módulo **module-info.java** y son el tipo más común de módulos definidos por el usuario..

- Exportacion de nombres

exports: Esta directiva se utiliza en el archivo module-info.java de un módulo para especificar qué paquetes o tipos se pueden acceder desde otros módulos.

```
module miModulo {
    exports mi.paquete.publico;
}
```

En este ejemplo, el paquete **mi.paquete.publico** está expuesto para que otros módulos puedan importarlo.

Se puede exportar un paquete solo a módulos específicos usando **to**, es decir hacemos una exportacion condicionalmente. Por ejemplo:

```
module my.module {
    exports com.example.mypackage to another.module;
}
```

opens: Para permitir la reflexión en tiempo de ejecución, se pueden abrir paquetes usando opens. Por ejemplo:

```
module my.module {
    opens com.example.mypackage;
}
```

- Importacion de nombres:

requires: Esta directiva se utiliza para declarar las dependencias de un módulo. Al declarar una dependencia, se obtiene acceso a los paquetes exportados por ese módulo.

```
module miModulo {
    requires java.sql;
}
```

En este ejemplo, el módulo **miModulo** requiere el módulo **java.sql**, lo que significa que puede utilizar las clases y interfaces definidas en ese módulo.

uses: Esta directiva se utiliza para indicar que un módulo utiliza un servicio definido por otro módulo.

```
module miModulo {
    uses java.util.spi.CurrencyNameProvider;
}
```

En este ejemplo, el módulo **miModulo** utiliza el servicio **CurrencyNameProvider**, que es proporcionado por el módulo **java.base.**

III. Diga si el lenguaje ofrece la posibilidad de crear aliases, sobrecarga y polimorfismo. En caso afirmativo, dé algunos ejemplos.

- **Alias:** En Java, un alias es una referencia a un objeto existente. Es decir, en lugar de crear un nuevo objeto, creamos una nueva variable que apunta al mismo objeto en memoria. Esto es especialmente útil cuando se trabaja con objetos grandes o cuando se desea modificar un objeto desde diferentes puntos de un programa. Los tipos primitivos en Java (int, double, etc.) no son objetos, por lo que no se pueden crear alias de ellos. Ejemplo:

```
String cadenaOriginal = "Hola mundo";
String aliasCadena = cadenaOriginal;
aliasCadena = aliasCadena + "!";
System.out.println(cadenaOriginal);
```

  En este ejemplo, **aliasCadena** es un alias de **cadenaOriginal**. Cualquier cambio realizado en **aliasCadena** también afectará a **cadenaOriginal** porque ambos apuntan al mismo objeto en memoria. En dicho ejemplo se imprime **Hola mundo!**

- **Sobrecarga:** La sobrecarga de métodos permite definir varios métodos con el mismo nombre, pero con diferentes parámetros. Esto permite que un método realice diferentes acciones dependiendo de los argumentos que se le pasen. Cabe destacar que Java no permite la sobrecarga de operadores. Por ejemplo:

```
public class OverloadingExample {
    public void print(int i) {
        System.out.println("Integer: " + i);
    }

    public void print(double d) {
        System.out.println("Double: " + d);
    }

    public void print(String s) {
        System.out.println("String: " + s);
    }
}
```

  En este caso se puede se tienen tres metodos **print**, uno para imprimir enteros, otro para imprimir punto flotante y uno para imprimir strings. El compilador selecciona el método adecuado en función de los tipos de los argumentos que se le pasen.

- **Polimorfismo:** El polimorfismo permite que una referencia de clase padre apunte a objetos de clase hija. Esto es útil para dinámicamente decidir qué método se ejecutará en tiempo de ejecución. Ejemplo:

```
class Animal {
    public void hacerSonido() {
        System.out.println("El animal hace un sonido");
    }
}
```

```
class Perro extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("El perro ladra");
    }
}

class Gato extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("El gato maulla");
    }
}
```

En este ejemplo, la clase **Animal** puede hacer referencia a objetos de tipo **Perro** o **Gato**. Cuando se llama al método **hacerSonido()**, se ejecutará la implementación correspondiente a la clase del objeto real.

IV. Diga qué herramientas ofrece a potenciales desarrolladores, como: compiladores, intérpretes, debuggers, profilers, frameworks, etc.

- Compiladores:
  - **javac:** El compilador estándar de Java que convierte código fuente Java (.java) en bytecode (.class).
- Interpretes:
  - **java:** La máquina virtual de Java (JVM) ejecuta bytecode Java.
- Debuggers:
  - **jdb:** El debugger de línea de comandos de Java.
- Profilers:
  - **jvisualvm:** Una herramienta visual para monitoreo, depuración y perfilado de aplicaciones Java.
  - **jconsole:** Una consola de monitoreo para la plataforma Java que utiliza JMX (Java Management Extensions).
  - **YourKit:** ayuda a monitorear el rendimiento y el consumo de recursos de aplicaciones Java.
- Frameworks de aplicaciones web:
  - **Spring:** Un marco para aplicaciones empresariales que facilita la creación de aplicaciones de alto rendimiento y escalables.
  - **Hibernate:** Un framework de mapeo objeto-relacional (ORM) para facilitar la persistencia de datos en bases de datos.
  - **Apache Struts:** Un marco para desarrollar aplicaciones web basadas en el patrón MVC (Modelo-Vista-Controlador).
- Frameworks para pruebas:
  - **JUnit:** El framework de pruebas unitarias más utilizado en Java.
  - **TestNG:** Un framework de pruebas flexible con características avanzadas.
- Entornos de Desarrollo Integrados (IDEs)
  - **Eclipse:** Un IDE gratuito y de código abierto muy popular, altamente personalizable y con una gran cantidad de plugins.
  - **IntelliJ IDEA:** Un IDE comercial muy potente y productivo, especialmente diseñado para desarrollo Java.
  - **NetBeans:** Otro IDE gratuito y de código abierto, con una interfaz intuitiva y soporte para muchos lenguajes.
- Herramientas de Gestión de Proyectos
  - **Maven:** Una herramienta de gestión de proyectos y construcción que automatiza la compilación, pruebas y empaquetado de aplicaciones Java.
  - **Gradle:** Una herramienta de construcción flexible y basada en Groovy, que ofrece una alternativa a Maven.