Instituto Politécnico Nacional

Escuela Superior de Cómputo

Sistemas Distribuidos

Tarea 1: Fundamentos de Java

7CM1

Alumno: Franco Olvera Demian Oder
Boleta: 2021630278

Profesor: Ukranio Coronilla Contreras

Fecha: 18 de febrero de 2024

Contenido

Preámbulo

Java es un lenguaje de programación del paradigma Orientado a Objetos de uso popular que fue creado desde 1990 y ha permanecido como una opción viable al momento de desarrollar proyectos informáticos. El pertenecer al POO le permite a Java tener una opción viable para reutilizar código y es conocido por ser de propósito general, lo que significa que es capaz de ejecutarse en cualquier máquina sin la necesidad de realizar modificaciones importantes dentro del código fuente.

A continuación, se presenta una serie de apuntes que explican términos importantes con respecto a este lenguaje de programación.

JDK, JRE & JVM

El desarrollo en Java parte del uso de múltiples herramientas clave, como es el caso del *Java Development Kit* (JDK), *Java Runtime Environment* (JRE) y la *Java Virtual Machine* (JVM). Se explican a continuación:

JDK

El *Java Development Kit* es un software que incluye al intérprete de Java, clases y otras herramientas de desarrollo como un compilador, depurador, desensamblador, generador de archivos y documentación, entre otras cosas. Permite generar aplicaciones que de ejecución única (sin modificación o segunda compilación del código). Parte importante del JDK son también los paquetes y herramientas de Java.

JRE

Se lo entiende como una capa de software ejecutado sobre el Sistema de Operativo para proporcionar las bibliotecas, clases y demás recursos para que un programa Java se pueda ejecutar. El JRE realiza una combinación entre el código Java que es creado gracias a las herramientas proporcionadas por el JDK para ejecutarlo en una instancia de la JVM.

JVM

La *Java Virtual Machine* es un entorno de ejecución que puede colocarse dentro de cualquier Sistema Operativo. Tiene como función ejecutar las instrucciones generadas por un compilador de Java; tiene un intérprete de *Bytecode* y un entorno de tiempo de ejecución para ejecutar archivos de clase Java en cualquier plataforma. Esto es lo que permite que Java sea un lenguaje de propósito general.

Datos primitivos en Java

Dentro de Java, el lenguaje como tal es considerado de tipado estático, por lo que cada variable es declarada junto con el tipo de dato al que pertenece. Existen ocho tipos de datos primitivos, los cuales son descritos a continuación junto con sus rangos y la memoria que utilizan:

- byte: utiliza 8 bits en memoria y su rango es -128 a 127.
- short: utiliza 16 bits en memoria y su rango es -32,768 a 32,767.
- int: utiliza 32 bits en memoria y su rango es -2^{31} a 2^{31}.
- long: utiliza 64 bits en memoria y su rango es -2^{63} a 2^{63}.
- float: utiliza 32 bits en memoria y su rango es -2^{31} a 2^{31}.
- double: utiliza 64 bits en memoria y su rango es -2^{63} a 2^{63}.
- boolean: utiliza sólo un bit de información.
- char: utiliza 16 bits en memoria y su rango es -32,768 a 32,767.

Casting

Hay ciertas situaciones donde es necesario convertir un dato a otro tipo de dato de manera cruda y directa. A esta acción se le conoce como Casting o conversión y es muy utilizada entre datos primitivos. Suele hacerse directamente cuando los dos tipos de datos guardan una relación concreta.

Puede realizarse de forma explícita o implícita. Cuando se realiza un Cast implícito se iguala el valor de una variable con el de otra de distinto tipo siempre y cuando el tipo que va a almacenarla posea un rango más grande, mientras que los Cast explícitos declaran el tipo de variable específico al que se quiere convertir y puede almacenar datos grandes en primitivos con rangos más pequeños.

Clases y objetos

Las clases y objetos son quizá el concepto más importante y general de Java. Las clases son la base de la POO. Una clase es una plantilla para la creación de objetos; dentro de una clase se colocan los datos y el código que se va a utilizar para operar dichos datos; las clases periten construir objetos, que vendrían a ser simplemente instancias de esa misma clase con datos específicos propios del objeto.

Al definir una clase nueva en Java, se declara su forma y naturaleza exacta. Esto quiere decir que se especifican los atributos que van a utilizar en instancia y los métodos que van a operar dentro de dichas clases.

En resumen, una clase es un prototipo o plantilla que define a las variables y a los métodos de todos los objetos comunes que van a representar a una entidad en particular. Los objetos, por otro lado, es una instancia o espécimen de una clase en particular. Los objetos suelen modelar o representar objetos del mundo real que suelen encontrarse en la vida cotidiana. A continuación, se presenta un ejemplo:

```java
// Se declara una clase pública y se nombra
public class ClasePrueba
{
    // Se declaran los atributos que tendrá la clase
    private String atributo1;
    private long atributo2;
    private fouble atributo3;

    // Se declara un constructor para la clase
    public ClasePrueba(){}

    // Se declaran los métodos que tendrá la clase
    // Método vacío
    public void metodo1()
    {
        System.out.println("Metodo vacio.");
    }
    // Método con retorno
    public String metodo2()
    {
        return "Metodo 2.";     // Regresamos la String "Metodo 2."
    }
}
```

Clases Wrapper

Como ya se sabe, Java posee tipos de datos primitivos. Tomando en consideración las características POO inherentes del lenguaje, se define a los datos primitivos como aquellos que no tienen métodos, mientras que los no primitivos son todos aquellos que tienen métodos (objetos).

Para todos los datos primitivos existen unas clases conocidas como *Wrapper* o de envoltorio, las cuales tienen una serie de mecanismos que envuelven datos primitivos para tratarlos como objetos. Cada dato primitivo tiene asociado un *Wrapper*, los cuales se mencionan a continuación:

- *Byte.*
- *Short.*
- *Integer.*
- *Long.*
- *Float.*
- *Double.*
- *Boolean.*
- *Character.*

Todos los *Wrapper* provienen de la clase *Object*. La forma en que se utilizan es muy sencilla ya que lo único que se tiene que hacer es tratar a los datos a los que se les quiere hacer *Wrapper* como la creación de cualquier otro objeto. A continuación, un ejemplo con la clase *Integer*:

```
1.  // Se declara una clase pública y se nombra
2.  public class ClasePrueba
3.  {
4.      // Se declara una función main
5.      public static void main(String args[]){
6.          // Declaramos un entero con su clase Wrapper
7.          Integer i = new Integer(5);
8.          int x = i.intValue();
9.      }
10. }
```

Cada objeto de tipo *Wrapper* tiene su conjunto de métodos particulares.

Almacenar en memoria variable local y de tipo objeto

Hay que explicar primero la diferencia entre una variable local y otra de tipo objeto.
Las variables locales se declaran dentro de un método o un constructor y que sólo pueden accederse bajo ese contexto, por lo que sólo existen durante la ejecución de dicho método o bloque. Por otro lado, las variables de tipo objeto almacenan referencias y suelen estar a nivel de clase o método, por lo que su vida útil realmente no está determinada.

- Las variables locales se almacenan en el *Stack* de la memoria, por lo que se eliminan por completo cuando el bloque de código donde se encuentran termina su ejecución.
- Las variables de objeto almacenan una referencia a dicho objeto en el *Heap* de la memoria. El objeto se crea en el *Heap* y la variable sólo contiene una referencia. Cuando una variable de objeto deja de tener referencias hacia ella, puede eliminarse por el *Garbage Collector*.

Ahora, el Garbage Collector es una parte del JRE que gestiona la memoria liberándola de objetos que dejan de ser referenciados para evitar fugas o desbordamientos de memoria. Al eliminar dichos objetos, da lugar a que objetos nuevos ocupen su lugar. De esta forma, esta herramienta permite que no sea necesario gestionar la memoria directamente.

Arreglos

No se trata realmente de un concepto exclusivo de Java, pero los arreglos son una estructura de datos que se utiliza para almacenar datos del mismo tipo; dichos elementos se encentran almacenados en ubicaciones de memoria juntos unos de otros. Pese a que no son exclusivos de Java, Java trata a los arreglos como objetos, por lo que los es adecuado decir que los arreglos tienen métodos.
Sus funciones son muy útiles: permiten almacenar múltiples valores del mismo tipo dentro de una única variable, así como permitir el acceso a cada uno de dichos valores gracias al hecho de que cada uno de ellos posee un índice numérico. Cabe aclarar que los arreglos son unidimensionales.

Arreglo String[] args

Es normal que todas las aplicaciones Java tengan un método principal o main y cada uno de ellos suele tener declarado dicho arreglo dentro de los argumentos que pueden recibir.
Dicho arreglo es especial dado que recibe comandos directamente desde la línea de comandos (que podría ser el CMD de Windows o la terminal de Linux); de esta manera, es posible enviar información inmediata al programa gracias a los argumentos que se colocan dentro de la línea de comandos. Un caso de uso muy simple puede verse a continuación:

```java
// Se declara una clase pública y se nombra
public class ClasePrueba
{
    // Se declara una función main
    public static void main(String args[]){
        // Tomamos los argumentos desde consola para realizar una suma
        int suma = 0;
        for (String arg : args) {
            int num = Integer.parseInt(arg);
            suma = suma + num;
        }
        System.out.println("El resultado es: " + suma);
    }
}
```

Packages

Los paquetes o dentro de Java es el medio que ofrece Java para ordenar los componentes de una aplicación. Permiten colocar en su interior cualquier cosa, como vendría a ser el caso de clases, archivos y demás código. Sirven principalmente para otorgarle un orden particular a alguna aplicación más compleja que incorpore muchos elementos para su funcionamiento. Permite ordenar o incluso categorizar de forma lógica los componentes de una aplicación.
Los paquetes dentro de Java, se generan y se importan; a continuación, se muestra un ejemplo práctico de creación e importación de un paquete dentro de Java:
Se crea un paquete con la clase Suma, para realizar dicha operación a través de un método que toma dos elementos y regresa un int con el resultado de la suma entre ambos:

```java
// Nombramos el paquete
package adicion;

// Generamos la clase Suma
```

```java
public class Suma {
    // Establecemos el método que va a sumar los dos números enteros
    public static int sumar(int a, int b) {
        return a + b;
    }
}
```

Tras ello, dicho paquete es importado en otro archivo que contiene una función main que importará el paquete y utilizará la clase para ejecutar una suma simple:

```java
// Importamos el paquete
import adicion.Suma;

// Se declara una clase pública y se nombra
public class ClasePrueba
{
    // Se declara una función main
    public static void main(String args[]){
        int resultado = Suma.sumar(5, 3);
        System.out.println("La suma es: " + resultado);
    }
}
```

Clase String

Dentro de lenguajes un poco más simples como es el caso de C, en Java existen también las String con la diferencia de que no se trata de Arrays como en otros casos, sino que son retratadas como clases dentro de este lenguaje de programación.

En Java existe la clase String, la cual representa una cadena de caracteres alfanuméricos cuya composición es invariable una vez creada. Existe una amplia variedad de métodos para la Clase String y entre los más importantes están:

1. length(): que devuelve el tamaño de la cadena en un int.
2. concat(String): que permite concatenar una String nueva.
3. equals(Object): que permite comparar cadenas entre sí.
4. equalsIgnoreCase(Object): compara cadenas entre sí sin importar las mayúsculas o minúsculas.
5. substring(int, int): devuelve una subcadena a partir de los índices especificados en sus parámetros.
6. indexOf(int/char): regresa el índice de la primera ocurrencia del carácter que se coloca.
7. startsWith(String): verifica que la String inicie con un prefijo en específico.
8. endsWith(String): verifica que la String finalice con un sufijo en específico.
9. toUpperCase(): pasa la String a mayúsculas.
10. toLowerCase(): pasa la String a minúsculas.
11. trim(): elimina espacios en blanco al inicio y al final de la String.

Palabra reservada this

this es una palabra reservada que hace referencia al objeto actual de la clase que se está trabajado. Permite utilizar métodos y atributos de la misma clase desde alguno de sus métodos. Permite el acceso al objeto completo, por lo que también se utiliza para usar

constructores de clase o incluso pasar el objeto completo a otra clase u objeto completamente diferentes.
Se utiliza para evitar la redundancia que implica llamar al objeto dentro del mismo método o bloque de código donde se supone que pertenece, por lo que suele funcionar para incrementar la claridad del código.

Herencia

Es un concepto exclusivo de la POO, por lo que se encuentra presente en Java. Es un concepto que apoya ampliamente en la simplificación de las aplicaciones y la reutilización del código.
La herencia se da cuando se genera una clase a partir de una clase existente. A la clase nueva se le considera como la clase Hija y a la clase existente se le llama clase Padre. De este modo, se evita duplicar el código y éste puede expandirse de la forma más orgánica.
Las clases hijas heredan los atributos y métodos de las clases padre y luego pueden tomar dichos elementos para modificarlos según las necesitades que se presente. Para utilizar este concepto en las diferentes clases, se tiene que utilizar la palabra reservada *extends*.
A continuación, se muestra un ejemplo:
Primero, generamos una clase genérica con un método que tome dos valores enteros con un método:

```
1.  // Creamos la clase genérica
2.  public class OperacionMatematica {
3.      // Escribimos el método genérico
4.      public int calcular(int n1, int n2) {
5.          return 0;
6.      }
7.  }
```

Ahora, generamos una clase hija que va a heredar el método de la clase genérica y modificarlo para realizar una operación matemática con ambos enteros:

```
1.  // Creamos la clase hija
2.  public class Suma extends OperacionMatematica {
3.      // Reescribimos el método para realizar una suma
4.      @Override
5.      public int calcular(int n1, int n2) {
6.          return n1 + n2;
7.      }
8.  }
```

Polimorfismo

El polimorfismo representa otro de los elementos importantes presentes en la POO. En el contexto de Java, el polimorfismo es la capacidad que tienen ciertos objetos de una clase en ofrecer una respuesta diferente e independiente en función de los parámetros que se utilizan para invocarlos. Suele ser conocido también como sobrecarga de parámetros.
Un ejemplo de ellos puede verse a continuación:
Si retomamos el código anterior, también es posible tomar la clase genérica para que otras clases tomen su método y hagan más que sumar dos números. Veamos:

Para la clase Suma:

```java
// Creamos la clase hija
public class Suma extends OperacionMatematica {
    // Reescribimos el método para realizar una suma
    @Override
    public int calcular(int n1, int n2) {
        return n1 + n2;
    }

    // Se sobrecarga el método para sumar más de dos números
    public int calcular(int n1, int n2, int n3) {
        return n1 + n2 + n3;
    }
}
```

Para la nueva clase Resta:

```java
// Creamos una nueva clase hija
public class Resta extends OperacionMatematica {
    // Reescribimos el método para realizar una resta
    @Override
    public int calcular(int n1, int n2) {
        return n1 - n2;
    }

    // Se sobrecarga el método para restar más de dos números
    public int calcular(int n1, int n2, int n3) {
        return n1 - n2 - n3;
    }
}
```

@Override

Representa una técnica que permite crear una versión de un método que ya se ha definido en una superclase (o clase padre). De este modo, puede cambiarse de una forma directa algún método heredado para modificar o agregar funciones adicionales.
Cuando una clase hija define un método con el mismo nombre, parámetros y retorno, entonces se puede decir que anula el comportamiento del método original, por lo que se vuelve necesario utilizar la anotación @Override de tal forma que la implementación de ese método para dicha clase sea diferente.
toString() es un método que devuelve la representación en String de algún objeto en particular; normalmente, lo convierte en una cadena legible; por otro lado, equals(Object) verifica si dos objetos son iguales.
Ambas clases extienden de la clase Object.

Sobrecarga de métodos

Este es un concepto clásico de Java. La sobrecarga de métodos se refiere a que un método dentro de una clase tiene la capacidad de recibir diferentes tipos de argumentos o parámetros. Esto ocurre cuando una clase ya tiene un método definido con una serie de parámetros ya también establecidos y es vuelto a declarar bajo el mismo nombre, pero con diferentes parámetros.

En ejemplos previos, ya se vio algo de este estilo, pero va a hacerse uno nuevo para establecer el concepto:

```
1.  // Creamos una nueva clase hija para el ejemplo de sobrecarga
2.  public class Producto extends OperacionMatematica {
3.      // Reescribimos el método para realizar una resta
4.      @Override
5.      public int calcular(int n1, int n2) {
6.          return n1 * n2;
7.      }
8.
9.      // Se sobrecarga el método para multiplicar más de dos números (sobrecarga)
10.     public int calcular(int n1, int n2, int n3) {
11.         return n1 * n2 * n3;
12.     }
13. }
```

Manejo de excepciones

Se define como Excepción a un evento que interrumpe la secuencia normal de operaciones dentro de un programa o aplicación; en otras palabras, son errores que pueden ocurrir y que impiden que las operaciones se desarrollen de una manera normal. Para manejar estos errores, se utilizan las excepciones. El manejo de ellas por parte del usuario se realiza de manera preventiva, para evitar que sea el programa mismo quien detenga todos los procesos abruptamente; no obstante, es responsabilidad del programador hallarlas y gestionarlas.
Concluyendo con el ejemplo de las operaciones, se creará una clase que divida dos números y lance una excepción cuando el usuario intente dividir entre 0:
Creamos la clase de División con su excepción:

```
1.  // Creamos una nueva clase para dividir
2.  public class Division extends OperacionMatematica {
3.      // Reescribimos el método para realizar una resta
4.      @Override
5.      public int calcular(int n1, int n2) {
6.          if(n2 == 0) {
7.              throw new ArithmeticException("No se puede dividir entre cero.");
8.          }
9.          return n1 * n2;
10.     }
11. }
```

Implementamos una función main para que el usuario pueda ver la excepción:

```
1.  // Generamos una función main
2.  public class Main(String args[]) {
3.      // Escribimos la función main
4.      public static void main{
5.          int a = 4;
6.          int b = 0;
7.          int resultado = 0;
8.
9.          // Generamos un bloque try-catch para usar el método
10.         try {
11.             resultado = Division.calcular(a, b);
12.             System.out.println(resultado);
13.         }
14.         catch (ArithmeticException e) {
```

```
15.             System.out.println("Error: " + e.getMessage());
16.         }
17.     }
18. }
```

Uso de Static

La palabra Static se utiliza para dos situaciones particulares. En primera instancia, cuando están colocadas junto a un atributo de una clase. Lo que esto hará es convertir dicha variable en una variable de clase, lo que significa que dicha variable será compartida entre todas las instancias de la misma; es decir, sólo existirá una copia de dicha variable estática al inicializarse una única vez para mantener su valor. Sirve para ahorrar memoria.
Esa palabra reservada puede usarse también en conjunto con los métodos para tener el mismo efecto. Los métodos que tienen la palabra reservada Static se convierten automáticamente en métodos de clase, por lo que son propios de la clase más que de la instancia como tal.