Tarea No. 1

First Reading - Interfaces and Abstract Classes

Grupo 3

Integrantes:

Viviana Poveda Orozco

Diego Paesani Pena

Kiara Rojas Mata

Universidad de Costa Rica

Sede del Atlántico

Recinto de Paraíso

Curso: Algoritmos y estructura de datos

Curso lectivo 2023

1. Resumen Chapter 2: Object-Oriented Design

2.1 **Object-Oriented Design Goals**

Los objetos son como su nombre lo indica la base en la programación orientada a objetos, estos mismo requieren “instancias” de la clase a la que pertenecen, estas instancias son la representación de los objetos creados en la clase instanciada.

2.1.1

Toda implementación de software debe cumplir con algunas características, robustez para que el programa pueda detectar errores en la entrada de datos sin caerse, adaptabilidad que permita evolucionar a diferentes plataformas o tecnología sin un gran cambio, un ejemplo de esto sería que una aplicación debe ser capaz si es necesario de pasar a formato web, reutilización que va de la mano con adaptabilidad pues ese código debe poder servir para diferentes plataformas ahorrando costos en nuevo software.

2.1.2 **Object-Oriented Design Principles**

Algunas herramientas o técnicas que facilitan poder cumplir con las características anteriores son las siguientes:

• Abstracción: es una táctica que busca las partes más fundamentales de un programa y usa diferentes herramientas como un ADT que como se dice en el libro Data Structures and Algorithms in Java “Un ADT es un matemático modelo de una estructura de datos que especifica el tipo de datos almacenados, las operaciones soportadas en ellos y los tipos de parámetros de las operaciones”

• Encapsulación: la encapsulación ayuda a que los componentes de un código no revelen información, además de que le da libertad al usuario de implementar código sin miedo de que otros programadores escriben código que depende de decisiones internas.

• Modularidad: “El modularidad se refiere a un principio organizador en el que los diferentes componentes de un sistema de software se dividen en unidades funcionales separadas.” Data Structures and Algorithms in Java pagina 62.

2.1.3 **Design Patterns**

Para la programación orientada a objetos se requiere más que conocer las metodologías o estrategias, también se requiere de un correcto uso de las mismas, para ello se han creado “Patrones de Diseño” que son soluciones a problemas típicos en programación, como una plantilla que nos guía a resolver el problema en cuestión.

2.2 **Inheritance**

Una forma efectiva de organizar componentes estructurales es hacerlo en forma de jerarquía o de herencia, mientras más alto en la jerarquía menos detalles o datos requiere ese componente. Esto significa que funciones más generales quedan más alto en la jerarquía y ayuda a futura reutilización

2.3. Interfaces and Abstract Classes

Dos objetos interactúan si “conocen” los métodos que admite cada objeto. Para que se de ese conocimiento las clases deben especificar la interfaz que sus objetos presentan a otros objetos. En el enfoque ADT-based, una interfaz que define un ADT se especifica como una definición de tipo y una colección de métodos para ese tipo, con argumentos para cada método. Esta especificación la emplea el compilador que requiere que los tipos de parámetros que se pasan a los métodos se ajusten al tipo especificado en la interfaz. Este requisito se llama strong typing. Aunque definir interfaces y hacer que esas definiciones se apliquen mediante strong typing es una carga para el programador, esta carga se compensa con hacer cumplir el principio de encapsulación y con detectar errores de programación que podrían pasar desapercibidos.

2.3.1 Interfaces in Java

La interfaz es el elemento estructural principal en Java que impone una API. Una interfaz es una colección de declaraciones de métodos sin datos ni cuerpos. Es decir, sus métodos están vacíos; solo son firmas de métodos. Las interfaces no tienen constructores y no se pueden instanciar directamente.

Para que una clase implemente una interfaz, todos sus métodos deben estar declarados en la interfaz. Así las interfaces imponen los requisitos para que una clase de implementación tenga métodos con ciertas firmas específicas.

Otra característica de las clases e interfaces en Java, es que una clase puede implementar múltiples interfaces. Esto permite una gran flexibilidad al definir clases que deben ajustarse a varias API.

2.3.2 Multiple Inheritance for Interfaces

La herencia múltiple es la capacidad de extenderse desde más de un tipo. En Java, se permite la herencia múltiple para las interfaces, pero no para las clases. Eso porque las interfaces no definen campos ni cuerpos de métodos, en cambio, las clases sí. Además, si Java permitiera la herencia múltiple para las clases, podría haber una confusión si una clase intentara extenderse desde dos clases con el mismo nombre o métodos con las mismas firmas.

Un uso de la herencia múltiple de interfaces es aproximarse a una técnica llamada mixin. Y es que, a diferencia de Java, algunos lenguajes orientados a objetos permiten la herencia múltiple de interfaces y de clases concretas En esos lenguajes, se definen clases, llamadas clases mixin, que no son creadas como objetos independientes, sino que están proporcionan funcionalidad adicional a las clases existentes. Sin embargo, tal herencia no está permitida en Java, por lo que se debe aproximarla con interfaces. Entonces se puede usar la herencia múltiple de interfaces como para "mezclar" los métodos de dos o más interfaces no relacionadas para hacer una interfaz que combine sus funciones.

2.3.2 Abstract Classes

En Java, las clases abstractas se encuentran entre las clases tradicionales y las interfaces. Esto porque como una interfaz, una clase abstracta puede definir firmas para métodos sin tener que implementar sus cuerpos; esos métodos se llaman métodos abstractos. Sin embargo, una clase abstracta también puede definir uno o más campos y métodos con implementación (llamados métodos concretos). Además, estas clases puede extender otra clase y ser extendida por más subclases.

Como las interfaces, una clase abstracta no se puede instanciar, no se puede crear ningún objeto a partir de una clase abstracta. Una subclase de una clase abstracta debe implementar los métodos abstractos de su superclase o debe permanecer abstracta. Para diferenciarlas de las clases abstractas, a las clases no abstractas se les llama clases concretas.

Está claro que las clases abstractas son más poderosas que las interfaces, porque pueden facilitar alguna funcionalidad concreta. Aun así, el uso de clases abstractas en Java se limita a la herencia simple, por lo que una clase puede tener como máximo una superclase, ya sea concreta o abstracta.

Se debe aprovechar las clases abstractas, ya que admiten una mayor reutilización del código. Porque la similitud entre una familia de clases se puede colocar dentro de una clase abstracta, que sirve como superclase para múltiples clases concretas. Así, las subclases concretas solo deben implementar la funcionalidad adicional que las diferencia entre sí.

2.4 Exceptions

Las excepciones son eventos inesperados que suceden durante la ejecución de un programa. Pueden producirse por un recurso no disponible, una entrada inesperada de un usuario o simplemente un error lógico del programador. En Java, las excepciones son objetos que genera o el código cuando se encuentra en una situación inesperada o la máquina virtual de Java. Además, una excepción puede ser capturada por un bloque de código que "maneja" el problema de manera adecuada. Cuando no se detecta, una excepción la máquina virtual puede dejar de ejecutar el programa y enviar un mensaje a la consola.

2.4.1 Catching Exceptions

Cuando ocurre una excepción y no se maneja, el sistema de tiempo de ejecución finaliza el programa después de imprimir un mensaje junto con un seguimiento de la pila de tiempo de ejecución. Ese seguimiento muestra las llamadas de métodos anidados que estaban activas en el momento en que ocurrió la excepción.

Sin embargo, antes de que finalice un programa, cada método en el seguimiento de la pila puede detectar la excepción. Cada método puede capturar la excepción o permitir que pase al método que lo llamó.

La metodología general para el manejo de excepciones es una construcción try-catch en la que se ejecuta un fragmento protegido de código que podría generar una excepción. Si arroja una excepción, esta se captura haciendo que el flujo de control salte a un bloque de captura

predefinido que contiene el código para analizar la excepción y aplicar una adecuada solución. Si no se produce ninguna excepción, se ignoran todos los bloques catch.

Hay varias reacciones cuando se detecta una excepción. La más común es imprimir un mensaje de error y terminar el programa. También hay casos en los que la mejor manera de manejar una excepción es atraparla e ignorarla silenciosamente. Otra forma es crear y lanzar otra excepción, que especifique la condición excepcional con mayor precisión.

2.4.2 Throwing Exceptions

Las excepciones se hacen cuando una pieza de código encuentra algún tipo de problema durante la ejecución y lanza un objeto de excepción. Esto se hace usando throw seguida de una instancia del tipo de excepción que se lanzará. A menudo es conveniente instanciar un objeto de excepción en el momento en que se debe lanzar la excepción. Así que, generalmente una declaración de lanzamiento se escribe como: throw new exceptionType(parameters).

The throws clause

Cuando se declara un método, es posible declarar la posibilidad de que se produzca un tipo de excepción durante una llamada a ese método. No importa si la excepción proviene de una instrucción throw en el cuerpo de ese método o si se propaga hacia arriba desde una llamada de método secundario.

La sintaxis para declarar posibles excepciones en la firma de un método se basa en throws. Por ejemplo:

public static int parseInt(String s) throws NumberFormatException;

Aunque se use una cláusula throws en la firma de un método se tiene que documentar todas las posibles excepciones usando la etiqueta @throws dentro de un comentario javadoc. El tipo y las razones de cualquier posible excepción deben declararse en la documentación de un método.

2.4.3 Java´s Exception Hierarchy

Java posee una jerarquía de herencia de todos los objetos que se consideran Throwable. La jerarquía se divide en dos subclases: error y excepción. Los errores solo los genera la máquina virtual de Java y designan las situaciones más graves que es poco probable que se puedan recuperar. Por el contrario, las excepciones designan situaciones en las que un programa en ejecución podría recuperarse.

Checked and Unchecked Exceptions

Java posee mayor refinamiento al declarar la clase RuntimeException como una subclase importante de Exception. Todos los subtipos de esa clase en Java se tratan como unchecked exceptions (excepciones no verificadas), y cualquier tipo de excepción que no sea parte de RuntimeException es una checked exception (excepción verificada).

La intención es que las excepciones de tiempo de ejecución se deban a errores en la lógica de programación. Aunque esos errores ocurrirán como parte del proceso de desarrollo de software, se supone que deberían resolverse antes de que el software alcance la calidad de producción. Gracias a la eficiencia no se verifica explícitamente cada uno de estos errores en el tiempo de ejecución y, entonces estos se designan como excepciones "no verificadas".

Por el contrario, se producen otras excepciones por condiciones que no se pueden detectar hasta que se ejecuta un programa. Por lo general, se designan como excepciones "verificadas".

La designación entre excepciones verificadas y no verificadas es importante en la sintaxis del lenguaje. En particular, todas las excepciones verificadas que puedan propagarse hacia arriba desde un método deben declararse explícitamente en su firma.

Una consecuencia es que, si un método llama a un segundo método declarando excepciones verificadas, entonces la llamada a ese segundo método debe protegerse dentro de una declaración try-catch, o bien el método que llama debe declarar las excepciones verificadas en su firma.

2.5 Casting and Generics

2.5.1 Casting

Widening Conversions

Métodos para convertir tipos de objetos ocurre cuando:

- Ty U son tipos de clase y U es una super clase de T
- T y U son tipos de interfaz y U es una super interfaz de T
- T es una clase que implementa la interfaz U

Se realizan para almacenar el resultado de una expresión en una variable, nos permite directamente asignar el resultado de T a una variable de tipo U.

Narrowing Conversions

Métodos para convertir tipos de objetos ocurre cuando:

- T y S son tipos de clase y S es una subclase de T.
- T y S son tipos de interfaz y S es una subinterfaz de T.
- T es una interfaz implementada por la clase S.

Su validez debe ser probada por el entorno de tiempo de ejecución de java durante la ejecución del programa.

Casting Exceptions

Java proporciona un operador instaceof que prueba si la variable se refiere a un objeto que pertenece a un tipo en particular. Devuelve verdadero si satisface el tipo de referencia y falso de ser lo contrario, permitiendo así evitar un ClassCastException.

2.5.2 Generics

Java incluye soporte para escribir clases y métodos generics que a menudo evita la necesidad de conversiones explicitas. Los generics nos permite definir una clase en términos de conjunto de parámetros de tipo formal que luego se pueden usar como tipo declarado dentro de la definición de clase.

Si necesitamos almacenar un par de datos podríamos diseñar una clase personalizada para ese propósito, pero luego queremos almacenar otros datos pero de diferente tipo que los iniciales, generics nos permite escribir una sola clases que pueda representar todos esos pares.

Using Java's Generics Framework

Se utiliza cualquier identificador valido, una letra entre paréntesis angulares es suficiente <E>

Generics and Arrays

Java permite que una matriz definida con un tipo parametrizado se inicialice con una nueva matriz no paramétrica y luego convertirla al tipo parametrizado (el compilador emitirá una advertencia ya que no es completamente seguro).

- Declarar una matriz que almacena instancias de la clase generic con parámetros de tipo reales fuera de la clase generic. Se deberá declarar con un tipo parametrizado, pero instanciarse sin parametrizar y luego convertir de nuevo al tipo parametrizado.
- Declarar una matriz que almacena objetos que pertenecen a uno de los tipos de parámetros formales. Queremos almacenar un numero fijo de entradas genéricas en una matriz, si el class usa <T> como un tipo parametrizado, puede declarar una matriz de tipo T[ ], pero no puede instanciar directamente una matriz de este tipo. Lo mejor seria crear una instancia de una matriz de tipo Object[] y luego hacer una conversión para escribir T[].

Generic Methods

Permite definir versiones genéricas de métodos individuales para esto se incluye un genérico de tipo formal entre los modificadores del método.

2.6 Nested Classes

Permite anidar una definición de la clase dentro de la otra definición, el uso principal de las clases anidadas es cuando se define una clase que esta fuertemente relacionada o ligada a otra clase, ayudando a aumentar la encapsulación y reducir conflictos de nombres.

Son esenciales al implementar estructuras de datos ya que se puede representar una pequeña porción de una estructura.

Una clase anidada no estática se conoce como clase interna de Java, esta clase solo se puede crear en un método no estático de la clase externa.

2. Glosario

Burden: Es una carga pesada que se lleva y es difícil de soportar. Es algo difícil o desagradable de lo que tienes que lidiar o preocuparte. También se refiere a la responsabilidad de pagar el dinero que debe, o a la gran cantidad de dinero que debe. (Cambridge, s.f.).

Shipment: Se refiere a una gran cantidad de bienes enviados a un lugar, o al acto de enviarlos. (Cambridge, s.f.).

Inheritance: Es una característica física o mental heredada de los padres a través de sus genes, o el proceso en el cual eso sucede. También se usa para el dinero, tierra o posesiones recibidas de alguien después de que la persona haya muerto. (Cambridge, s.f.).

Hierarchy: Es un sistema en el que las personas o las cosas se organizan en varios niveles o rangos de acuerdo con su importancia o con la autoridad que tienen. Y en donde, las personas en los niveles superiores lo controlan. (Cambridge, s.f.).

Nest: Practica de incorporar llamadas a funciones o procedimientos dentro de otras. (Cambridge, s.f.).

3. Programa funcional en java

Adjunto en mediación virtual.

Referencias:

Goodrich, M., & Tamassia, R. (2014). Data structures and algorithms in java(6th ed.). John Wiley Sons, Inc.

Cambridge University Press. (s.f.). *Cambridge Dictionary*. https://dictionary.cambridge.org/es/diccionario/ingles/