COMP249 Tutorial 10: Review Tutorial

Ella Noyes

August 12, 2024

Before we start with some review notes, I'd like to go over some code from tutorial 8 (the lambdas and generics practice problem), and last tutorial (the Music Queue example).

1 Relationships in UML Class Diagrams

UML class diagrams specify different types of relationships between classes; the nature of those relationships is shown using different connectors.

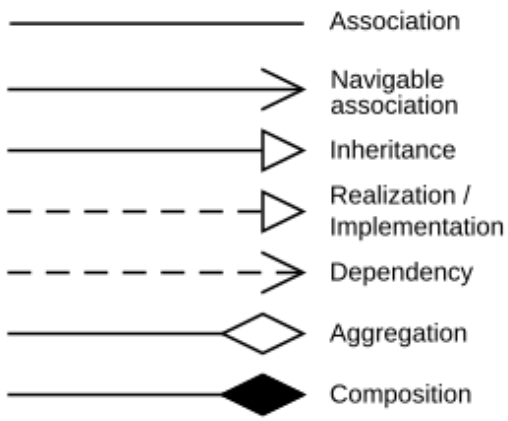

Figure 1: https://en.wikipedia.org/wiki/Class_diagram

In brief,

- **Dependency:** Changes in class A's definition will affect class B.
- **Association:** "Uses-a", or "Has-a". Class A uses an instance of class B, or class A has an instance of class B.
- **Inheritance:** "Is-a". An object of class B is an object of class A, but class B extends class A with its own fields and/or methods, or overrides existinf methods in class A.
- **Realization/Implementation:** Class B implements abstract methods in class A. Class A can therefore be a abstract class (where class B uses the **extends** keyword) or an interface (where class B uses the **implements** keyword).
- "**Aggregation:** [...] the child can exist independently of the parent. Example: Class (parent) and Student (child). Delete the Class and the Students still exist." https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/
- "**Composition:** [...] the child cannot exist independent of the parent. Example: House (parent) and Room (child). Rooms don't exist separate to a House." https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/

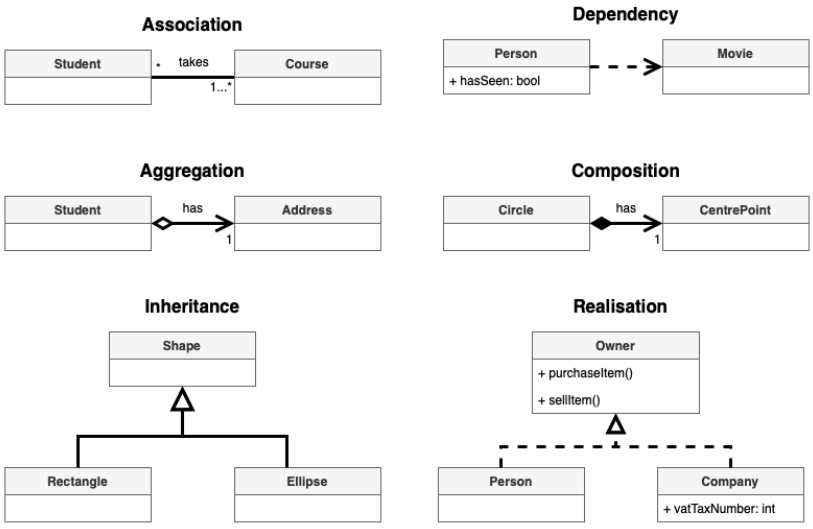

Figure 2: https://www.drawio.com/blog/uml-class-diagrams

2 Inner Classes

Inner classes are classes that are defined (and therefore contained) within a class. There are four types of inner classes. I'll show you some code demonstrating each of these in a moment.

1. **Static Inner Classes:** Since these are static, they may be instantiated without instantiating the outer class. And since they are static, they may not access non-static variables and methods from their outer class.
2. **Non-Static Inner Classes:** These may access the outer class' non-static variables and methods.
3. **Local Inner Classes:** Defined within a block (e.g., a method, constructor, or an initialization block.) It is local to that block; it can only be instantiated and used within the block where it is defined.
4. **Anonymous Inner Classes:** A type of local inner class that does not have a name and is used to instantiate an object of a class or interface with an immediate implementation.

See code examples of all of these in the tutorial 4.

3 Polymorphism

Polymorphism means having "many different forms" and describes a method that takes on different forms/functionality.

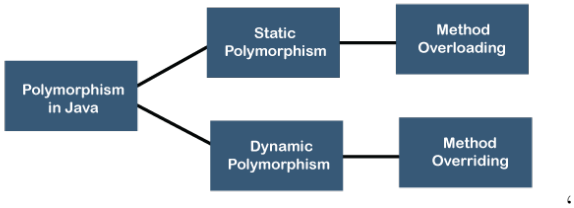

Figure 3: Diagram is from: https://www.javatpoint.com/dynamic-polymorphism-in-java

- Dynamic Polymorphism is also called run-time polymorphism
- Static Polymorphism is also called compile-time polymorphism

4 Abstract Classes and Interfaces

A reminder from tutorial 2:
Abstract classes and interfaces are both mechanisms in Java for defining methods that can be implemented by subclasses or implementing classes, but they serve different purposes and have distinct characteristics.

- Abstract classes are used when you want to provide a common base with shared code and allow subclasses to extend it while optionally overriding methods. They can contain both abstract methods (without implementation) and concrete methods (with implementation).
- Interfaces, on the other hand, are used to define a contract for what methods a class should implement without providing any implementation details. A class can implement multiple interfaces, enabling a form of multiple inheritance.

Overall, abstract classes are best for creating a foundational class with shared behavior, while interfaces are ideal for specifying capabilities that can be shared across different class hierarchies.

Valeriia has more detailed slides on this in her review powerpoint on her GitHub.

5 Exception Handling

An exception is an event that interrupts a program's flow of execution.

In a try-catch statement, a catch block can catch an exception as an object of its superclass (e.g., you can catch a FileNotFoundException as IOException, or even as an Exception). But we typically try to catch the most specific type of exception that we expect will occur so that we can handle it appropriately.

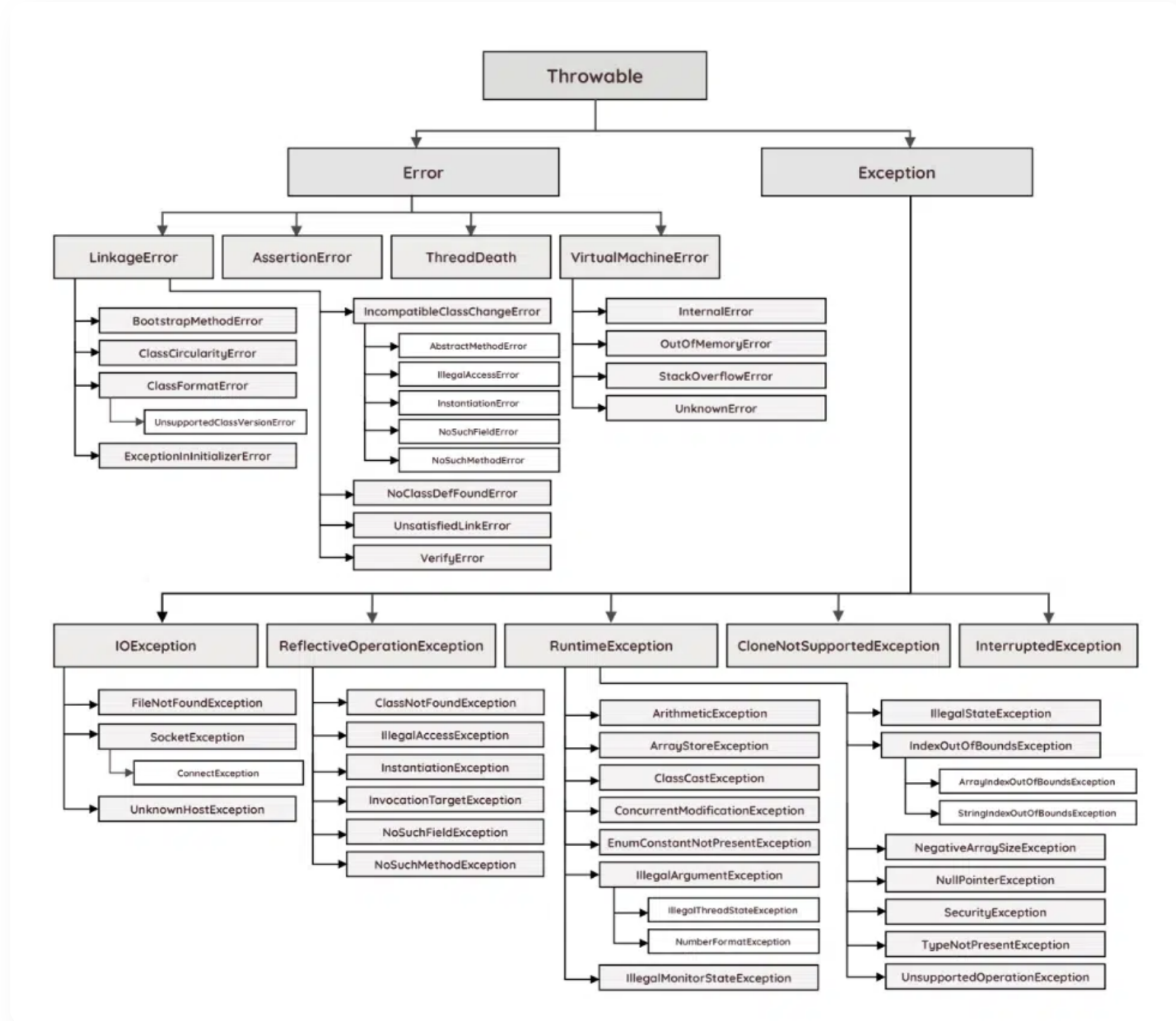

Figure 4: Diagram is from: https://rollbar.com/blog/java-exceptions-hierarchy-explained/

You can have multiple catch blocks

```
try {
        FileOutputStream fos = new FileOutputStream("src/Students.txt");

        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(student);

        FileInputStream fis = new FileInputStream("src/Students.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);

        Student deserializedStudent = (Student)ois.readObject();

        deserializedStudent.sayHello();

        oos.close();
        ois.close();
    } catch(IOException e) {
        System.out.println("Encountered an IO exception");
    } catch(ClassNotFoundException e) {
        System.out.println("Could not find class");
    }
```

Or you can catch different exceptions types in one catch block, separating the exception types using |

```
try {
        FileOutputStream fos = new FileOutputStream("src/Students.txt");

        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(student);

        FileInputStream fis = new FileInputStream("src/Students.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);

        Student deserializedStudent = (Student)ois.readObject();

        deserializedStudent.sayHello();

        oos.close();
        ois.close();
    } catch(IOException | ClassNotFoundException e) {
        System.out.println("Encountered an exception");
    }
```

If a method explicitly throws an exception, or calls a method that throws an exception, then it can be declared using the **throws** keyword. A method that throws an exception must be called within a try-catch block.
withdraw() explicitly throws a user-defined exception.

```
public void withdraw(double amount) throws InsufficientFundsException {
    if(amount <= balance) {
        balance -= amount;
    }else {
        double needs = amount - balance;
        throw new InsufficientFundsException(needs);
    }
}

public static void main(String [] args) {
    // Some code

    try {
        c.withdraw(600.00);
    } catch (InsufficientFundsException e) {
        // Handle exception
    }
}
```

readFirstLineOfFile() calls a method that throws a FileNotFound exception.

```
public static void readFirstLineOfFile(File file) throws FileNotFoundException {
    Scanner myScanner = new Scanner(file);
    if(myScanner.hasNext()) {
        System.out.println(myScanner.nextLine());
    }
    myScanner.close();
}

public static void main(String[] args) {
    File myFile = new File("src/myFile.txt");

    try {
        readFirstLineOfFile(myFile);
    } catch(FileNotFoundException fileNotFoundException) {
        fileNotFoundException.printStackTrace();
    }
}
```

6 File I/O

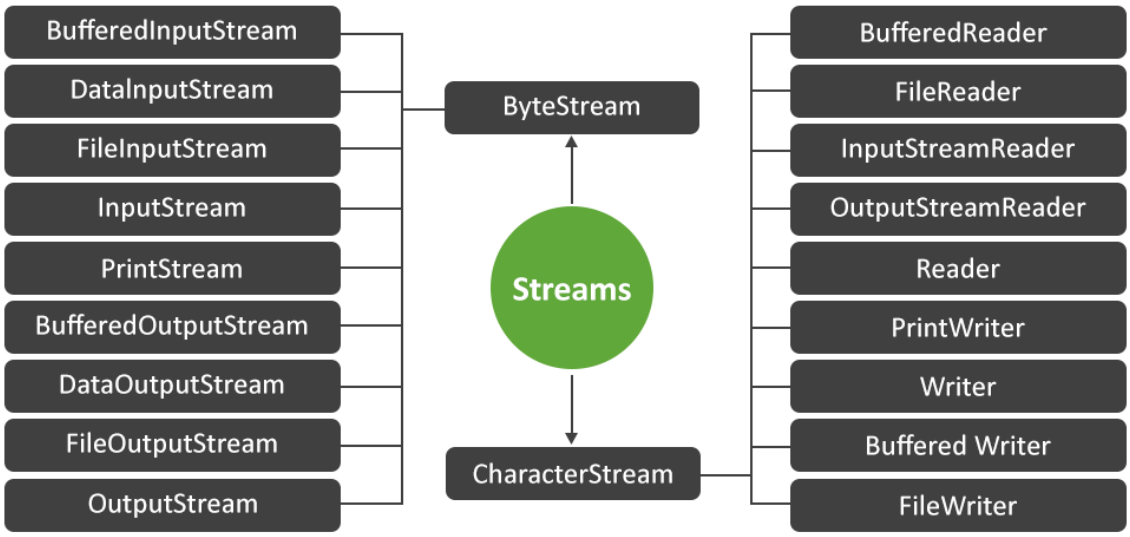

Figure 5: Diagram is from: https://www.geeksforgeeks.org/java-io-tutorial/

This diagram shows some of Java's I/O byte stream classes (a.k.a binary I/O or stream I/O) and I/O character stream classes (a.k.a text I/O).

7 Generics

Generics make code more generic (i.e., generalized or inclusive, as opposed to type-specific).

7.1 Syntax

A generic class is declared in Java using the following syntax:

```
public class ClassName<T> {
    ...
}
```

The angled brackets are the essential component in this. T is the "type parameter", it will be used to refer to objects of that generic type elsewhere in the class. You might, for example, have a data member of type T with a setter and a getter.

```
public class ClassName<T> {
    private T myObject;

    ...


    public T getMyData() {
        return myObject;
    }

    public void setMyData(T object) {
        this.myObject = objectl
    }
}
```

You can also write generic methods:

```
public <T> returnType methodName(){
    ...
}
```

You can restrict the types that generic methods/classes may use by restricting the type parameter using the **extends** keyword.

```
public <T extends MyClass> returnType methodName(){
    ...
}
```

T may only be be MyClass or one of its subclasses.

Tutorial 8 included some programming examples of generics.

8 Recursion

Recursion occurs when a method calls itself. It can be used to break a problem down into smaller parts. Recursion is referred to as a "Divide and Conquer" approach to problem-solving. A recursive method has two properties:

- **A base case:** The case that breaks out of the recursive method calls (i.e., a scenario that contains a return statement).
- **A recursive step:** A rule that leads successive method calls towards the base case.

9 Lambdas

A lambda expression is a shorthand notation for creating an instance of a functional interface by providing an implementation of its single abstract method in a concise form.
The syntax looks like this:

```
parameter -> expression
(parameter1, parameter2) -> expression
(parameter1, parameter2) -> { code block }
```

A statement lambda (one with a code block in curly braces) requires an explicit return statement. An expression lambda is just one line of code; everything to the right of the arrow is the return statement.

10 Practice

10.1 UML Diagrams

10.1.1 Drawing Diagrams

This one is from this Github site.

"Draw a UML class diagram of this code. Show relationships between classes and multiplicity of association."

```
class Person {
    private String name;
    public String getName() { return name; }
}

public class Student extends Person implements Comparable<Student> {
    private Long id;
    private List<Enrollment> courseList;
    public Student(String name, Long id) { /* code omitted. */ }

    public int compareTo(Student other) { /* code omitted */ }

    // an unmodifiable (immutable) view of the student's course list
    public List<Enrollment> getCourseList() {
        return Collections.unmodifiableList( courseList );
    }
}

public class Enrollment {
    ...
}
```

10.1.2 Interpreting UML Diagrams

uml-diagrams.org has many great examples of UML class diagrams that contain several types of relationships. It could be good practice to look through these and try to make sense of them. Here's an example

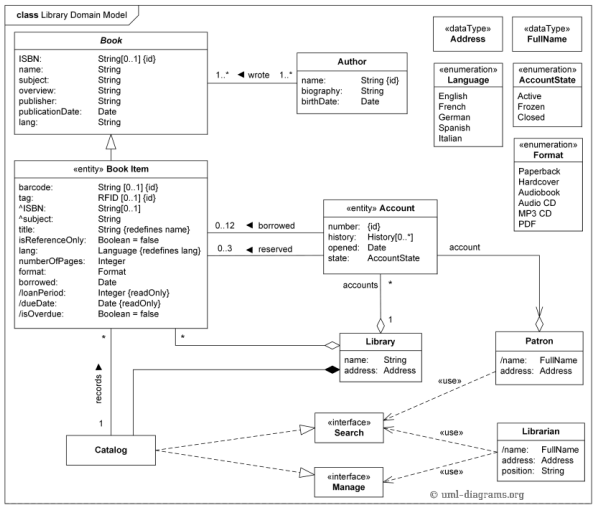

Figure 6: https://www.uml-diagrams.org/library-domain-uml-class-diagram-example.html

10.2 File I/O and Recursion

This is just like the prime number parsing problem from the recursion tutorial.
Create a Java program that performs the following tasks:

1. **Generate Random Integers and Write to a File:**
   - Write a Java program that generates **200 random integers** between **1** and **15** (inclusive).
   - Save these integers into a file named `numbers.dat`, with each integer on a new line.
2. **Implement Factorial Computation:**
   - Write a recursive method named `factorial(int number)` that computes the factorial of a given integer. The factorial of a non-negative integer n is the product of all positive integers less than or equal to n. For example, $5! = 120$.
3. **Read from File and Compute Factorials:**
   - Read the integers from `numbers.dat`.
   - Compute the factorial for each integer using the recursive method.
   - Print each computed factorial value to the console.

10.3 Lambdas

Find several programming problems on lambdas with here

10.4 Inheritance