Programming in Java

8. Exception Handling

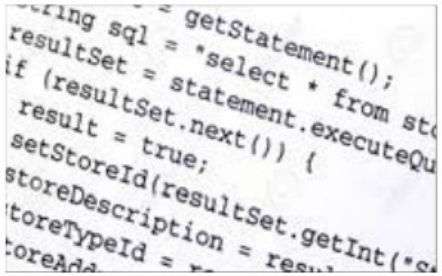
Terminology

- *Defect*: A generic term for any of the following
- *Failure*: When a software system or component does not perform its required functions according to the stated specifications or fails testing
- *Fault*: A fault is an incorrect step, construct, process or data usage in the code that causes failures
  - We can eliminate faults by fixing the code that results in a failure
  - Faults always result in the same failures until the code is fixed
- *Exception*: A fault in the environment the software runs in
  - Exceptions occur even when there is no fault in the code
  - Exceptions do not occur every time the code runs – depends on where it is running
  - All we can do is respond in some way when an exception occurs
    - For example, out of memory error, stack overflow, missing input file
- *Error*: A error is a human action that resulted in a fault

Terminology

Defects

causes



causes

Error **Fault** **Failure**

Handling Exceptions

- When an exception occurs, a special kind of object called an *Exception* is created
  - Contains a bundle of information about what kind of exception occurred and where it occurred
  - The object is the *thrown* or passed to an exception handler called *catch block*
  - The catch block decides how to respond to an exception
- The main goal of exception handling
  - Ensure the system is not left in an unstable state
  - Most of the time, the catch block will execute a clean shutdown of the application
    - Closing file, releasing resources, etc
  - Sometimes interactive applications will allow the user to correct the problem
    - Entering a different file name for a missing file
- Exceptions should *not* be thrown for predicable logic failures
  - For example, a bank account object should never throw a NSF exception
  - The NSF condition should be handled in the program logic

Try blocks

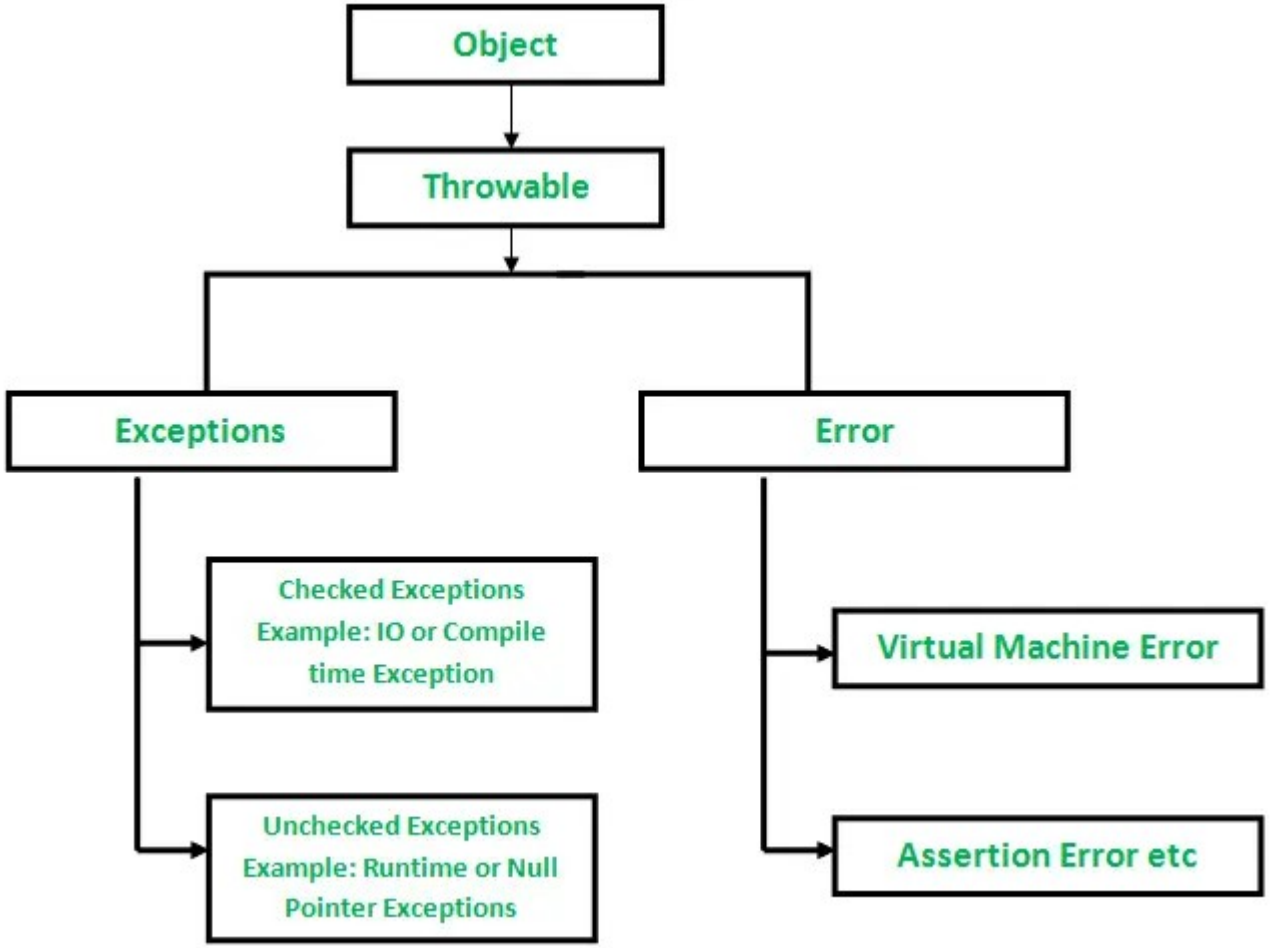
- A try block defines a block of code where an exception can occur
  - If it does, the try block stops
  - The catch blocks are examined to see which one can handle the thrown exception
  - That hander's code is executed
  - Execution continues at the first statement after the last catch block

```
public class TryCatchExample {
    public static void main(String[] args) {
        try {
            // Divide by zero
            int result = 10 / 0;

            // Null pointer access
            String text = null;
            System.out.println(text.length());

        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero: " + e.getMessage());
        } catch (NullPointerException e) {
            System.out.println("Null reference: " + e.getMessage());
        }

        System.out.println("Program continues after the try-catch block.");
    }
}
```

Handling Exceptions

- Exceptions halt any processing at the place that it occurs
  - This enables the JVM to undo anything that was done that could leave the system in an unstable state – allocating objects on the stack for example
  - This is called *unwinding the stack*
  - Keeping track of what was done is very expensive
  - We only do it over blocks of code we enclose in a *try block*
- Java has a built in hierarchy of exceptions
  - *Errors:* These usually are catastrophic failures in the JVM – we can't do anything about them
  - *Checked Exceptions:* Failing to provide exception handlers for these is a compile time error
  - *Unchecked Exceptions* Do not need to have exception handlers.
- Runtime exceptions
  - Subset of unchecked exceptions
  - Refer to common run time error that cannot be checked at compile time
  - Requiring these to be checked would be way to complicated and computationally expensive

Exception Classes



The throws Statement

- Methods may be called inside a try block
  - Exceptions thrown by the called method are still in the scope of the try block
  - The Java compiler can't figure this out from looking at the code
  - It just assumes you forgot a try block
  - Adding the *throws <exception>* to the method signature tells the compiler the exception is caught by the calling method

```java
public class ThrowsExample {

    // Method declares it may throw IOException
    public static void readInput() throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter your name: ");
        String name = reader.readLine();  // May throw IOException
        System.out.println("Hello, " + name);
    }

    public static void main(String[] args) {
        try {
            readInput();  // Caller handles the exception
        } catch (IOException e) {
            System.out.println("An error occurred while reading input: " + e.getMessage()
        }
    }
}
```

More Exceptions

- It is a best practice to define your own exceptions
  - These identify the specific issues that might arise in your own code
  - For example, and *AccountNotFound* exception when data is missing from a database that should be there
  - We create custom exceptions by extending either *Exception* or *RuntimeException*
  - Designing an effective exception hierarchy is part of good program design
- A try block can have multiple catch blocks
  - Each one handles a different kind of exception
  - Ordering is important since a catch block for an exception type will also match all the exceptions derived from it
  - The *finally* block contains clean up code that would be executed whether an exception is thrown or not – it allows us not not repeat code in different places
  - Nowadays, it is preferred to use the *try-with-resoruces* the does the same thing whenever possible

Custom Exceptions

- It is a good practice to create specialized custom exceptions for critical or important potential errors
  - In the example, a custom exception is created for an empty string
  - Notice that we have a constructor for Exception that takes a string argument for the message

```java
// Custom exception class
class EmptyStringException extends Exception {
    public EmptyStringException(String message) {
        super(message);
    }
}
```

```java
public class CustomExceptionDemo {

    // Method that throws an exception for empty strings
    public static void checkName(String name) throws EmptyStringException {
        if (name.length() == 0) {
            throw new EmptyStringException("String is empty");
        } else {
            System.out.println("Valid name: " + name);
        }
    }

    public static void main(String[] args) {
        try {
            checkName("");  // This will throw the custom exception
        } catch (EmptyStringException e) {
            System.out.println("Caught exception: " + e.getMessage());
        }
    }
}
```

Finally Block

- Used to define code that always executes
  - Regardless of whether an exception was thrown or caught.
  - Typically used for clean-up actions like closing files, releasing resources, or resetting states.
  - Runs after the try and catch blocks, even if an exception is thrown or not.

```java
public class FinallyExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 0;  // Will throw ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Caught exception: " + e.getMessage());
        } finally {
            System.out.println("This runs no matter what.");
        }

        System.out.println("Program continues after try-catch-finally.");
    }
}
```

Try-with-resources

- Used to define code that always executes
  - Regardless of whether an exception was thrown or caught.
  - Typically used for clean-up actions like closing files, releasing resources, or resetting states.
  - Runs after the try and catch blocks, even if an exception is thrown or not.

```
public class FinallyExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 0;  // Will throw ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Caught exception: " + e.getMessage());
        } finally {
            System.out.println("This runs no matter what.");
        }

        System.out.println("Program continues after try-catch-finally.");
    }
}
```

Lab 8-1

Exceptions


Java™