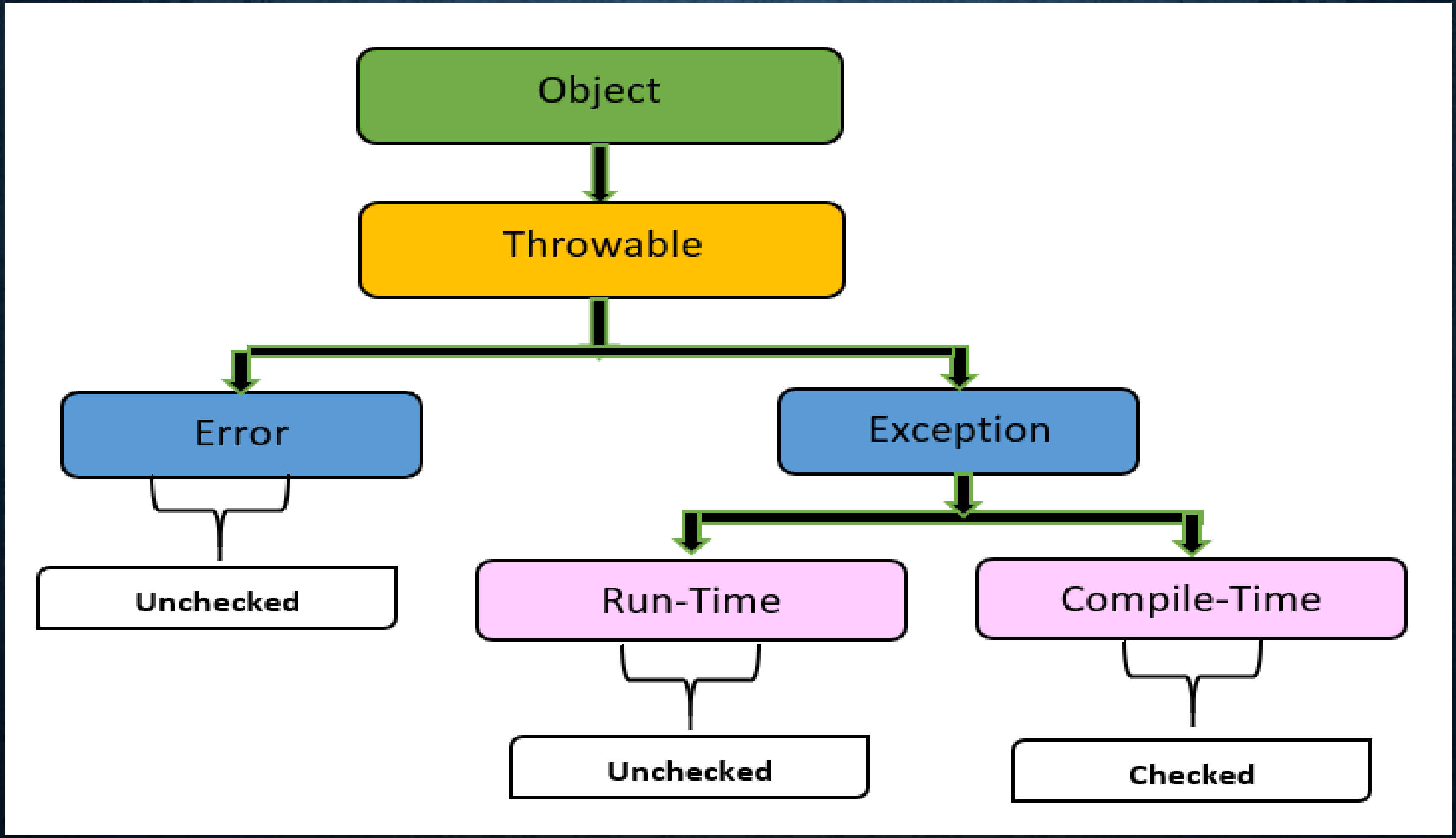
CORE JAVA

Day-7

EXCEPTION HANDLING

- Exception handling in Java is a powerful mechanism that manages runtime errors, allowing the program to run without interruption.
- Exception handling is important because it helps handle unexpected events that may occur during the execution of a program, ensuring that the code is robust and stable. The main components of exception handling in Java include
  - Exception: An exception is an event or condition that disrupts the normal flow of the program. It is an object which is thrown at runtime and is related to issues typically recoverable without necessarily halting the entire program.
  - Error: Unlike exceptions, errors indicate serious problems that a reasonable application should not try to catch. Errors are used by the Java runtime system to indicate errors having to do with the runtime environment itself (e.g., OutOfMemoryError).

EXCEPTION CLASS HIERARCHY



TYPES OF EXCEPTIONS

- Checked Exceptions: These are exceptions that must be either caught or declared to be thrown. Checked exceptions are checked at compile-time. Example: IOException, SQLException.
- Unchecked Exceptions: These are not checked at compile-time, which means that the compiler does not require methods to catch or declare these exceptions. Unchecked exceptions are the class RuntimeException and its subclasses. Example: NullPointerException, ArithmeticException.
- Errors: These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are subclasses of Error. Example: StackOverflowError, OutOfMemoryError.

TYPES OF EXCEPTION

- **Checked Exception**
  - ClassNotFoundException
  - InterruptedException
  - IOException
  - InstantiationException
  - SQLException
  - FileNotFoundException
- **Unchecked Exception**
  - ArithmeticException
  - ClassCastException
  - NullPointerException
  - ArrayIndexOutOfBoundsException
  - ArrayStoreException
  - IllegalThreadStateException

COMPONENTS OF EXCEPTION HANDLING

- Try Block: Code that might throw an exception is placed within a try block. If an exception occurs within the try block, it is thrown.
- Catch Block: A method to catch and handle the exception thrown by the try block. Each try block can be followed by one or more catch blocks, each catching a different type of exception.
- Finally Block: A block that follows try or catch blocks. This block is executed whether an exception is thrown or not, and it is typically used for clean-up activities.
- Throw: Used to explicitly throw an exception. throw keyword is followed by an instance of Exception class or its subclasses.
- Throws: Used in the signature of a method to indicate that this method might throw one of the listed types of exceptions. The calling method is responsible for handling these exceptions.

THROW AND THROWS KEYWORD

```
public class Main {
    public static void main(String[] args) {
        try {
            checkAge(15); // Valid age
            checkAge(-5); // This will cause an
exception
        } catch (IllegalArgumentException e) {
            System.out.println("Exception
caught: " + e.getMessage());
        }
    }
```

```
public static void checkAge(int age) throws
IllegalArgumentException {
        if (age < 0) {
            throw new
IllegalArgumentException("Age cannot be
negative.");
        } else {
            System.out.println("Valid age: " +
age);
        }
    }
}
```

CUSTOM EXCEPTION

- In Java, custom exceptions are user-defined exceptions that you create by extending existing exception classes.
- Creating custom exceptions is a good practice when you want your application to handle specific conditions or errors that are not covered by Java's standard exceptions.

CUSTOM EXCEPTIONS

- Custom exceptions can be checked or unchecked exceptions:
- Checked exceptions are those which the compiler forces the calling code to catch or declare. They extend Exception.
- Unchecked exceptions are those which the compiler does not force the calling code to catch or declare. They extend RuntimeException.

WHY CUSTOM EXCEPTION?

- Custom exceptions can make your application's error handling more meaningful and user-friendly.
- By defining specific exceptions for various error conditions, you can provide more detailed error information, which can help with debugging and maintaining the application.
- They also help in differentiating between different types of errors that might occur, allowing finer-grained error handling in higher layers of the application.

WRAPPER CLASSES

- In Java, wrapper classes are a fundamental concept used to convert primitive data types into objects.
- This conversion is essential because primitives are not objects and hence lack the capabilities that objects have, such as being used in collections that store objects (like ArrayList) and having methods to perform operations.
- Each primitive data type in Java has a corresponding wrapper class in the java.lang package.

USAGE OF WRAPPER CLASSES

- Object Creation: You can create an object of a wrapper class with or without using the new keyword. For example:

```
Integer myInt = new Integer(100);
Integer anotherInt = Integer.valueOf(100);
```

- Auto-boxing: Since Java 5, auto-boxing allows automatic conversion of primitive types into their corresponding wrapper class objects:

```
Integer myInteger = 50;  // Compiler converts it to Integer.valueOf(50);
```

- Unboxing: The reverse of auto-boxing where the object of a wrapper class is automatically converted to its corresponding primitive type:

```
int myInt = myInteger;  // Compiler converts it to myInteger.intValue();
```