Week 5 – Topics

- Object-Oriented Programming (OOP) Concepts
- Classes
- The Pillars of Object-Oriented Programming
- Exceptions
- Interfaces
- OOP Menu App

Object-Oriented Programming (OOP) Concepts

What is Object-Oriented Programming (OOP)

Object Oriented Programming (OOP) is a programming paradigm. Java is an Object Oriented Language, which means that it supports OOP, providing features that support and implement the Four Pillars of OOP.

In OOP, objects are the key construct in our programming, and these objects contain properties and functionality.

Instead of writing code all in one place, we structure our program into objects. Each object will contain relevant properties and functionality, which is code that is structured into methods that can be used over and over.

In OOP, programs are written by creating objects that contain both data (properties) and methods (functionality), in contrast with procedural programming where data exists, and procedures and methods perform operations on the data.

It takes time to start to think of data as being stored in an object which also contains all of the methods written for that object. Another way to explain this is to think of an object, and imagine that everything that you need (data or methods) is accessible anytime you reference that object. If you have access to the object, you also have access to any available property and functionality that belong to that object.

Classes vs Objects

Classes are the mechanism by which we structure our code in OOP.

A Class is a blueprint, in which you create your properties and functionality.
An Object is the actual product developed from the blueprint.
An Object is an instance of a Class!

So, a class is a template for objects, and an object is an instance of a class.

Example:

Class: Animal

Properties:	Functionality:
• name	
• weight	describe()
• type	
• location	

Object: dog

- A dog will be instantiated as an instance of the class, Animal.

```
Animal dog = new Animal();
```

Using / Accessing Objects

Getters / Setters are used for accessing and assigning values to Object properties (data)

We are going to create a class called *Animal.* This class will have two constructors, getters, setters, and a *describe()* method.

https://github.com/ckiefriter1/Java-Code-Examples/tree/main/src/classes

https://github.com/ckiefriter1/Java-Code-Examples/blob/main/src/classes/Animal.java

https://github.com/ckiefriter1/Java-Code-Examples/blob/main/src/classes/AnimalMain.java

https://github.com/ckiefriter1/Java-Code-Examples/blob/main/src/classes/GetterSetterExample.java

The Pillars of Object-Oriented Programming

There are four pillars of OOP which are:

1. Encapsulation -- Data Security -- binding related data and methods together into objects -- data hiding (hide the unnecessary).
2. Inheritance -- Code reusability -- acquiring the existing functionality of a parent class, with the ability to add additional functionality and features into the child class -- an object can inherit some properties and methods from another object.
3. Polymorphism -- Many Forms -- A single object can have multiple behaviors, or respond in different ways to the same function.
4. Abstraction -- Present a simplified view -- Hide the complexity from the user (show only what is necessary).

Each pillar is essential, and important, but these pillars are also dependent upon each other.

- Without encapsulation, abstraction and inheritance are not possible.
- Additionally, polymorphism does not exist without inheritance.

Each of these pillars is essential to the paradigm, and together define object-oriented programming. OOP as a whole functions through objects the contain properties and methods, and the relationships of these objects with other objects.

The Pillars of Object-Oriented Programming

What Is an Object?
An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life. This lesson explains how state and behavior are represented within an object, introduces the concept of data encapsulation, and explains the benefits of designing your software in this manner.

What Is a Class?
A class is a blueprint or prototype from which objects are created. This section defines a class that models the state and behavior of a real-world object. It intentionally focuses on the basics, showing how even a simple class can cleanly model state and behavior.

What Is Inheritance?
Inheritance provides a powerful and natural mechanism for organizing and structuring your software.

What Is an Interface?
An interface is a contract between a class and the outside world. When a class implements an interface, it promises to provide the behavior published by that interface.

What Is a Package?
A package is a namespace for organizing classes and interfaces in a logical manner

https://docs.oracle.com/javase/tutorial/java/concepts/index.html

Exceptions

Exceptions

What is an Exception?

An exception is defined an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

In Java, there are two types of Exceptions:

- Checked Exceptions -- these are checked by the compiler at compiler time
  - *Example: syntax errors -- a missing semi-colon or closing curly brace, incorrect datatype assignment, returning the wrong data type from a method, etc.*

- Unchecked Exceptions -- these are not caught by the compiler, included here are runtime exceptions and errors.
  - Runtime Exception Example: Trying to access a Null Address, Out-of-Bounds Indices,
    Class: RuntimeException, NullPointerException, ArithmeticException, etc.
  - Error Exception Example: Serious issues that cause an application to abort, or to stop running, including Memory or Stack Overflow Errors,
    Class: OutOfMemory, StackOverflow, VirtualMachineError, etc.

https://github.com/ckiefriter1/Java-Code-Examples/tree/main/src/exceptions

https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

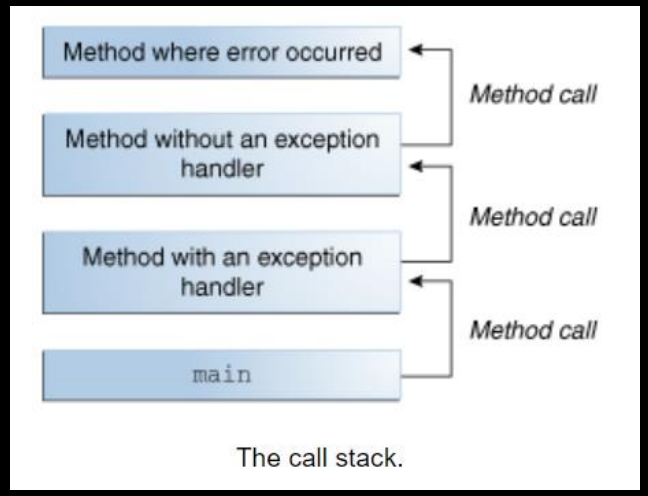
Exceptions

- An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

- When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred.

- Creating an exception object and handing it to the runtime system is called throwing an exception.

- After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the call stack (see the next figure).



The call stack.

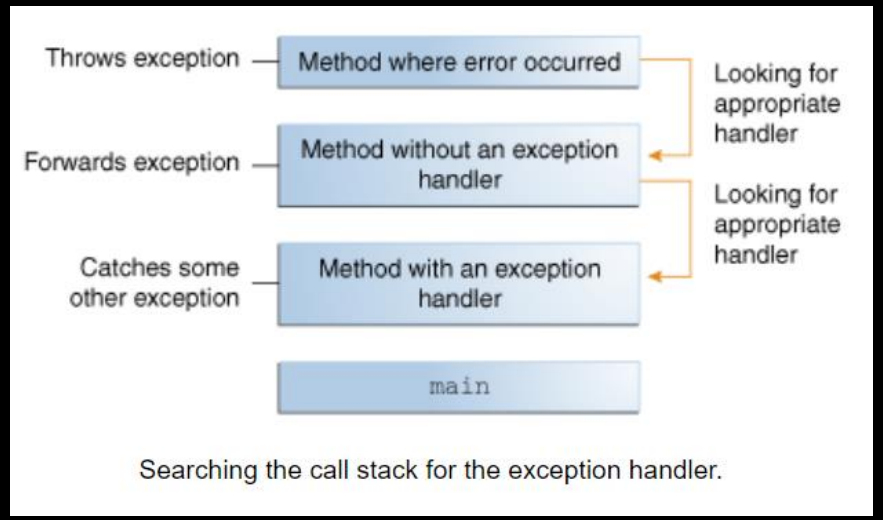
https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

Exceptions

- The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an exception handler.

- The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.

- The exception handler chosen is said to catch the exception. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the next figure, the runtime system (and, consequently, the program) terminates.



Searching the call stack for the exception handler.

https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

Catching Exceptions – try → catch → finally blocks

- You associate exception handlers with a try block by providing one or more catch blocks directly after the try block. No code can be between the end of the try block and the beginning of the first catch block.

```
try {
   // Execute some code here
} catch (ExceptionType name) {

} catch (ExceptionType name) {

} finally {
   // Code that always runs when the try block runs even when an Exception occurs
}
```

- Each catch block is an exception handler that handles the type of exception indicated by its argument. The argument type, ExceptionType, declares the type of exception that the handler can handle and must be the name of a class that inherits from the Throwable class. The handler can refer to the exception with name.

- The catch block contains code that is executed if and when the exception handler is invoked. The runtime system invokes the exception handler when the handler is the first one in the call stack whose ExceptionType matches the type of the exception thrown. The system considers it a match if the thrown object can legally be assigned to the exception handler's argument.

- The following are two exception handlers for the writeList method:

```
try {
   // Do something
} catch (IndexOutOfBoundsException e) {
   System.err.println("IndexOutOfBoundsException: " + e.getMessage());
} catch (IOException e)
   System.err.println("Caught IOException: " + e.getMessage());
```

Catching Exceptions – try → catch → finally blocks

- Exception handlers can do more than just print error messages or halt the program. They can do error recovery, prompt the user to make a decision, or propagate the error up to a higher-level handler using chained exceptions, as described in the Chained Exceptions section.

- Catching More Than One Type of Exception with One Exception Handler
  - In Java SE 7 and later, a single catch block can handle more than one type of exception. This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.

  - In the catch clause, specify the types of exceptions that block can handle, and separate each exception type with a vertical bar (|):

```
catch (IOException|SQLException ex) {
  logger.log(ex);
  throw ex;
  }
```

https://github.com/ckiefriter1/Java-Code-Examples/tree/main/src/exceptions

Catching Exceptions – try → catch → finally blocks

- You can use a finally block to ensure that a resource is closed regardless of whether the try statement completes normally or abruptly. The following example uses a finally block instead of a try-with-resources statement:

```
static String readFirstLineFromFileWithFinallyBlock(String path) throws IOException {
        FileReader fr = new FileReader(path);
        BufferedReader br = new BufferedReader(fr);
        try {
          return br.readLine();
        } finally {
          br.close();
          fr.close();
        }
}
```

Interfaces

Interfaces

What Is an Interface?
An interface is a contract between a class and the outside world. When a class implements an interface, it promises to provide the behavior published by that interface

An interface is a completely "abstract class" that is used to group related methods with empty bodies:

To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class with the implements keyword (instead of extends). The body of the interface method is provided by the "implemented" class

Classes can implement multiple interfaces

https://github.com/ckiefriter1/Java-Code-Examples/tree/main/src/interfaces

OOP Menu Example

Menu-Driven Applications

TBD

Appendix