Mastech InfoTrellis Java Course Contents:

Day 1 - Encapsulation, Abstraction, Inheritance, and Polymorphism

---

Day 2 - If Statements, If-Else Ladders, Ternary Operator, For/While/Do While Loops, Switch Case Statements, BreakStatements, and Continue Statements

---

Day 3 - Constructors, Exception Handling, and Recursion

Day 4 - Interfaces, and Strings

Day 5 - Data Structures, Get, and Set

Day 6 - File Input, File Output, and Properties Files

Day 7 - Java Database Connectivity (JDBC)

Day 1 - Encapsulation, Abstraction, Inheritance, and Polymorphism

Encapsulation

- **Encapsulation -** Building fields (variables) and methods (functions) into a class to protect data from being modified by other classes.

Abstraction

- **Abstraction -** The proces of hiding certain details and showing only essential information to the user.
- Three Main Visibities of Classes:
    1. **Public -** Can be accessed by any other class.
    2. **Private -** Can only be accessed by the class it is declared in.
    3. **Protected -** Can only be accessed by classes in the same package or subclasses.

    - Example:

    ```
    public class ClasOne{...} // public
    private class ClassTwo{...} // private
    protected class ClassThree{...} // protected
    ```

Inheritance

- **Inheritance -** The process of passing down data (methods and fields) from a one class to another (parent to child).
    - We use the ***"extends"*** keyword to inherit from a class.
    - Example:

    ```
    public class ChildClass extends ParentClass{...}
    ```

Polymorphism

- **Polymorphism -** The ability to take on many forms of a single method. A way of having multiple methods with the same name, but different parameters.

    - Two Types of Polymorphism:

        1. **Compile Time Polymorphism -** Different *number* of parameters

        - Example:

        ```
        public void add(int a, int b) {
            System.out.println(a + b);
        }

        public void add(int a, int b, int c) {
            System.out.println(a + b + c);
        }
        ```

        2. **Overloading Polymorphism -** Different *types* of parameters for a single method

- Example:

```
public void add(int a, int b) {
    System.out.println(a + b);
}

public void add(String a, String b) {
    System.out.println(a + b);
}
```

Day 2 - If Statements, If-Else Ladders, Ternary Operator, For/While/Do While Loops, Switch Case Statements, BreakStatements, and Continue Statements:

If Statements

- **If Statements -** A conditional statement that runs a certain block of code if a condition is true.
    - Example:

      ```
      if (condition) {...}
      ```

    - Use **==** to check if two values are equal, use **=** to assign a value to a variable.

      ```
      if (a == b) {...}
      ```

    - Use ***!=*** to check if two values are not equal.

      ```
      if (a != b) {...}
      ```

    - Use **>** to check if one value is greater than another.

      ```
      if (a > b) {...}
      ```

    - Use **<** to check if one value is less than another.

      ```
      if (a < b) {...}
      ```

    - Use **>=** to check if one value is greater than or equal to another.

      ```
      if (a >= b) {...}
      ```

    - Use _**<=**_ to check if one value is less than or equal to another.

      ```
      if (a <= b) {...}
      ```

    - Use ***&&*** to check if two conditions are both true.

      ```
      if ((a > b) && (a < c)) {...}
      ```

    - Use **||** to check if one of two conditions is true.

      ```
      if ((a > b) || (a < c)) {...}
      ```

    - Use ***!*** to check if a condition is false.

      ```
      if (!(a > b)) {...}
      ```

    - Use **^** to check if one of two conditions is true, but not both. (Called the exclusive or condition)

      ```
      if ((a > b) ^ (a < c)) {...}
      ```

- Use *%* to check if one value is a multiple of another (evaluates if not a perfect multiple, when remainder is not 0).

  ```
  if ((a % b) == 0) {...}
  ```

If-Else Ladders

- **If-Else Ladders -** A conditional statement that runs a certain block of code if a condition is true, and another block of code if the condition is false.
  - Example:

    ```
    if (condition) {...}
    else if (condition) {...}
    else {...}
    ```

Ternary Operator

- **Ternary Operator -** A conditional statement that runs a certain block of code if a condition is true, and another block of code if the condition is false.
  - Example:

    ```
    (condition) ? {...} : {...}
    ```

  - The code before the ":" is the code that runs if the condition is true, the code after the ":" is the code that runs if the condition is false.

For Loops

- **For Loops -** A conditional loop which iterates a block of code while true.
  - Example:

    ```
    for (int i = 0; i < 10; i++) {...}
    ```

While Loops

- **While Loops -** A conditional loop which iterates a block of code while true.
  - Example:

    ```
    while (condition) {...}
    ```

When To Use For Loop VS While Loop

- Use a ***for loop*** when you ***DO*** know how many times you want to iterate.
- Use a ***while loop*** when you ***DO NOT*** know how many times you want to iterate.

Do While Loops

- **Do While Loops -** A conditional loop which iterates a block of code while true.
  - Example:

    ```
    do {...}
    while (condition);
    ```

Switch Case Statements

- **Switch Case Statements -** A conditional statement that runs a certain block of code if a condition is true.
  - Used in place of an if-else ladder when there are many conditions to check.
  - Example:

    ```
    switch (variable) {
        case 1:
            // code block
            break;
        case 2:
            // code block
            break;
        default:
            // code block
    }
    ```

Break Statements

- **Break Statements -** A statement that breaks out of a loop or switch case statement when a condition is met.

  - Example:

    ```
    for (int i = 0; i < 10; i++) {
        if (i == 5) {
            break;
        }
    }
    ```

- ***Can label which loop to break out of in a break statement.***

  - Example:

    ```
    outerloop:
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; j++) {
            if (i == 5 && j == 5) {
                break (outerloop);
            }
        }
    }
    ```

- Unless specified, the break statement will break the innermost loop

Continue Statements

- **Continue Statements -** A statement that skips the current iteration of a loop when a condition is met.
  - Example:

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        continue;
    }
}
```

Difference Between Break and Continue Statements

- **Break Statements**
  - Breaks out of a loop
  - *Stops* the loop
- **Continue Statements**
  - Skips the current iteration of a loop
  - *Continues* the loop

Day 3 - Constructors, Exception Handling, and Recursion:

Constructors

- **Constructor -** A special method that is used to initialize objects.
    - Example:

      ```
      public class ClassName{
          [data type] a;
          [data type] b;

          public ClassName([data type] a, [data type] b){...}
      }
      ```

- **"this." Operator -** Using the "this." operator allows a constructor to access the global variable.
    - Example:

      ```
        public class ClassName{
        [data type] a;
        public ClassName([data type] a){
            this.a = ...;
            }
        }
      ```

- **"super()" Operator -** Using the "super()" operator allows a constructor that is a child class to access the information of the parent class.
    - **MUST BE THE FIRST STATEMENT IN THE CONSTRUCTOR.**
    - **If constructor is parameterized, must have variables listed within "()" of super()**
        - Example:

          ```
            public class ClassChild extends ClassName{
                [data type] c;

                public ClassChild([data type] c){
                    super([data type] a, [data type] b);
                    this.c = ...;
                }
            }
          ```

Exception Handling

- **Exception Handling -** In the form of Try-Catch, it lets the code compile. If there is an error, it will return the error as the output.
    - Example:

```
  try{...}
  catch{...}
```

- Exceptions are of various types, which include:
    - ArithmeticException
    - NullPointerException
    - IOException
    - SQLException
    - ClassNotFoundException
    - ...
- Exceptions are classified into two different categories:
    1. **Unchecked Exceptions -** Runtime Exceptions, may or may not occur. No need to add exception handling.
    2. **Checked Exceptions -** Compile Time Exceptions, may or may not occur. Should add exception handling.
- Can declare checked exception using the *throws* keyword:
    - Example:

```
  [return type] [method name]([parameters]) throws [exception class name]{...}
```

Recursion:

- **Recursion -** The process in which a function calls itself directly or indirectly.
    - Example:

```
[Visbility] [Return Type] [Function Name]([Parameters]){
    if ([Base Case]){
        return ([Value]);
        }
    else{
        [Recursive Call of [Function Name]];
    }
}
```

- Recursive blocks contain two types of statements:
    1. **Base Case -** Breaks out of recursion, returns a value
    2. **Recursive Case -** Continues the recursion, returns a call of the function

Day 4 - Interfaces, and Strings:

Interfaces:

- **Interface Class -** Abstract type used to specify the behavior of a class. Blueprint of a Java class
  - Example:

    ```
    interface [Class Name]{
        [Visibility] [Return Type] [Method Name]([Parameters]);
        ... // Can have as many methods laid out as needed
    }
    ```

  - **Can have as many interface classes as desired but only one extends class (parent class).**
  - Example:

    ```
    [Visibility] [Class Name]([Parameters]) implements [implented interface classes], ... extends [Parent Class]
    ```

- Can use *final* keyword to make a function's behavior unable to be modified
- If a method is not going to be implemented, use the keyword *abstract*

Strings:

- **String -** A sequence of characters, used to store text.
  - Example:

    ```
      String [Variable Name] = "[String]";
    ```

- **String Methods -** Methods that can be used to manipulate strings.
- Examples:

1. **.length() -** Returns the length of the string.
   - Example:

     ```
       [String Name].length();
     ```

2. **.charAt() -** Returns the character at the specified index.
   - Example:

     ```
       [String Name].charAt([Index]);
     ```

3. **.substring() -** Returns a substring of the string.
   - Example:

     ```
       [String Name].substring([Index]);
     ```

4. **.indexOf() -** Returns the index of the first occurrence of the specified character.
   - Example:

     ```
       [String Name].indexOf([Character]);
     ```

5. **.equals() -** Compares two strings, returns true if the strings are equal.
    - Example:

      ```
      [String Name].equals([String]);
      ```

- Java supports the use of Regular Expressions (RegEx) to find patterns in strings.
- Examples:

1. **.matches() -** Returns true if the string matches the specified regular expression.
    - Example:

      ```
      [String Name].matches("[RegEx]");
      ```

2. **.replaceFirst() -** Replaces the first occurrence of the specified regular expression with the specified replacement.
    - Example:

      ```
      [String Name].replaceFirst("[RegEx]", "[Replacement]");
      ```

3. **.replaceAll() -** Replaces all occurrences of the specified regular expression with the specified replacement.
    - Example:

      ```
      [String Name].replaceAll("[RegEx]", "[Replacement]");
      ```

4. **.split() -** Splits the string at the matches of the specified regular expression and returns an array of strings.
    - Example:

      ```
      [String Name].split("[RegEx]");
      ```

5. **.toLowerCase() -** Converts the string to lower case letters.
    - Example:

      ```
      [String Name].toLowerCase();
      ```

6. **.toUpperCase() -** Converts the string to upper case letters.
    - Example:

      ```
      [String Name].toUpperCase();
      ```

Day 5 - Data Structures, Get, and Set:

Data Structures:

- **Data Structure -** A data structure is a particular way of organizing data in a computer so that it can be used effectively.
- **Array -** A collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together.
  - Functions that can be used on an Array:
    - **.add -** Adds an element to the end of the array.
    - **.remove -** Removes an element from the array.
  - Example:

    ```
    [Data Type] [Array Name][] = new [Data Type][Array Size];
    ```

- **ArrayList -** A resizable array. Elements can be added and removed after compilation phase.
  - Functions that can be used on an ArrayList:
    - **.add -** Adds an element to the end of the array.
    - **.remove -** Removes an element from the array.
  - Example:

    ```
    ArrayList<[Data Type]> [Array Name] = new ArrayList<[Data Type]>();
    ```

- **Vector -** A dynamic array similar to ArrayList. Elements can be added and removed after compilation phase.
  - Functions that can be used on a Vector:
    - **.add -** Adds an element to the end of the array.
    - **.remove -** Removes an element from the array.
  - Example:

    ```
    Vector<[Data Type]> [Array Name] = new Vector<[Data Type]>();
    ```

- **Hash Table -** A data structure that implements an associative array abstract data type, a structure that can map keys to values.
  - Functions that can be used on a Hash Table:
    - **.put -** Adds an element to the end of the array.
    - **.remove -** Removes an element from the array.
  - Example:

    ```
    Hashtable<[Data Type], [Data Type]> [Array Name] = new Hashtable<[Data Type], [Data Type]>();
    ```

- **Hash Map -** A data structure that implements an associative array abstract data type, a structure that can map keys to values.
  - Functions that can be used on a Hash Map:
    - **.put -** Adds an element to the end of the array.

        - **.remove -** Removes an element from the array.
    - Example:

      ```
        HashMap<[Data Type], [Data Type]> [Array Name] = new HashMap<[Data
      Type], [Data Type]>();
      ```

- **DIFFERENCES BETWEEN ARRAYLIST AND VECTOR:**
    - **Array List-** Array List is **NOT SYNCHRONIZED**, meaning it is not thread safe.
    - **Vector-** Vector is **SYNCHRONIZED**, meaning it is thread safe.
- **DIFFERENCES BETWEEN HASH MAP AND HASH SET:**
    - **Hash Map-** Hash Map **ALLOWS** Null Values
    - **Hash Set-** Hash Set **DOES NOT ALLOW** Null Values

Get and Set:

- **Getters -** Getters are used to access the value of a variable, provide read-only access.
- **Setters -** Setters are used to update the value of a variable, provide write-only access.
- **Get and Set -** Get and Set are methods used to access and update the value of a variable respectively.
    - Set Example:

      ```
      [Visibility] void set[Variable Name]([Data Type] [Variable Name]){
          this.[Variable Name] = [Variable Name];
      }
      ```

    - Get Example:

      ```
      [Visibility] [Return Type] get[Variable Name](){
          return [Variable Name];
      }
      ```

Day 6 - File Input, File Output, and Properties Files:

File Input:

- **Input Stream -** Input Stream is used to read data from a file. Another way of opening the file for reading.
- How to open file in Java:

  ```
  File [File Name] = new File("[File Path]");
  InputStream [Input Stream Name] = new FileInputStream([File Name]);
  ```

- How to Create a File in Java:

  ```
  File [File Name] = new File("[File Path]");
  [File Name].createNewFile();
  ```

  - **Java Will Automatically Create A File If Opening a File That Does Not Exist.**
- How to close a file:

  ```
  [File Variable Name].close();
  ```

File Output:

- **Output Stream -** Output Stream is used to write data to a file. Another way of opening the file for writing (add keyword "true" to end of the file opening function_).

Read and Write to File:

- How to read data from a file:

  ```
  [File Variable Name].nextLine();
  ```

- **".read()" Function -** Reads a single character from the file.
  - **NOTE: To read more than one character, must use while a loop.**
  - Example:

    ```
    Reader [Reader Name] = new FileReader("[File Path]");
    int [Variable Name] = [Reader Name].read();
    while([Variable Name] != -1){
        System.out.print((char)[Variable Name]);
        [Variable Name] = [Reader Name].read();
    }
    ```

- **".readLine()" Function -** Reads a single line from the file.

- How to write data to a file:

  ```
  FileWriter [File Writer Name] = new FileWriter("[File Path]");
  OutputStream [Output Stream Name] = new FileOutputStream([File Name]);
  [File Writer Name].write("[Data]");
  [File Writer Name].close();
  ```

- **NOTE: FileWriter will OVERWRITE the file if it already exists (if not in append mode).**
- **NOTE: Files written this way must be written in bytes, to write Strings in Bytes use the "[String].getBytes()" function.**
- Write data to file using writer (allows for writing Strings):

  ```
  Writer [Writer Name] = new FileWriter("[File Path]");
  [Writer Name].write("[Data]");
  [Writer Name].close();
  ```

Properties Files:

- **Properties File -** A file that contains a list of key-value pairs. Can be used to load information from external sources (i.e. server) into a Java program quickly.
  - File Extension: *".properties"*
  - Propertie Files are overwritten with each run of the program.
- How to create a properties file:

  ```
  OutputStream [Output Stream Name] = new FileOutputStream("[File Path]");
  Properties [Properties Name] = new Properties();
  [Properties Name].setProperty("[Key]", "[Value]");
  [Properties Name].store([Output Stream Name], [Comment at Top of File]);
  OutputStream.close();
  ```

  - **NOTE: Output Stream MUST be initialized before creating the properties file.**
- How to read from a properties file:

  ```
  InputStream [Input Stream Name] = new FileInputStream("[File Path]");
  Properties [Properties Name] = new Properties();
  [Properties Name].load([Input Stream Name]);
  [Properties Name].getProperty("[Key]");
  InputStream.close();
  ```

  - **NOTE: Input Stream MUST be initialized before reading from the properties file.**

Day 7 - Java Database Connectivity (JDBC):

JDBC:

- **JDBC -** Java Database Connectivity is an API for Java that defines how a client may access a database.
- **JDBC Driver -** A JDBC Driver is a software component enabling a Java application to interact with a database.
- **JDBC URL -** A JDBC URL is a string that identifies a database connection.
- **JDBC API Interfaces:**
  - ***TO INITIATE CONNECTION:***
    - **Driver Interface -** The Driver Interface is the core of JDBC. All JDBC drivers must implement this interface.
    - **Connection Interface -** The Connection Interface is used to establish a connection with a specific database.
  - ***TO HELP RUN QUERIES:***
    - **Statement Interface -** The Statement Interface is used to execute SQL queries.
    - **PreparedStatement Interface -** The PreparedStatement Interface is used to execute parameterized queries.
    - **CallableStatement Interface -** The CallableStatement Interface is used to execute stored procedures.
  - ***TO GET RESULTS:***
    - **ResultSet Interface -** The ResultSet Interface is used to represent the result of a query.
    - **ResultSetMetaData Interface -** The ResultSetMetaData Interface is used to retrieve metadata from a ResultSet object.
    - **DatabaseMetaData Interface -** The DatabaseMetaData Interface is used to retrieve metadata from a database.
    - **RowSet Interface -** The RowSet Interface is used to represent a set of rows from a ResultSet object.
- **DML -** Data Manipulation Language is used to retrieve, store, modify, delete, insert, and update data in a database.
  - **DML Commands:**
    - **SELECT -** Used to retrieve data from a database.
    - **INSERT -** Used to insert new data into a database.
    - **UPDATE -** Used to update existing data within a database.
    - **DELETE -** Used to delete records from a database.
- **CRUD -** CRUD is an acronym for the four basic types of SQL commands:
  - **C** (Create)
  - **R** (Read)
  - **U** (Update)
  - **D** (Delete)
- **DDL -** (Data Definition Language) Used to define the database structure or schema

- **TCL -** (Transaction Control Language) Used to manage different transactions occurring within a database

- **Collection Framework -** List, ArrayList, Vector, Set, HashSet, HashMap, etc.

- **File Operations -** FileReader, FileWriter, FileInputStream, FileOutputStream, Buffered Reader, Buffered Writer, etc.

- **Metadata -** Data that describes other data.

- **Pure Java Driver Layers:**

    - **Java Application -** Allows user to interact with the database.
    - **JDBC API -** Provides universal data access from the Java programming language.
    - **(Thin) Type 4 Driver -** Allows database to be Oracle, MySQL, DB2, etc. (Comes from companies like Oracle who own the database, which will be implementing these interfaces.)
    - **Database Management System (DBMS) -** Keeps data in a database.

- Packages and Classes are contained in a *".jar"* file.

- **JDBC Steps:**

    1. Create Connection
    2. Execute from Queries
    3. Obtain Results
    4. Close Connection

- **To Connect With Data Base, Need:**

    - Address of Database Server
    - Database Name
    - Credentials (Username and Password)

NOTE: Server can be held locally or on the cloud (i.e. AWS) with the leading `*` printed before it: *

- JDBC Connection Code:

```
    String jdbcURL = "[URL Of Database Server]";
    String jdbcDriver = "[Name of Driver]";
    String jdbcUsername = "[User Name]";
    String jdbcPassword = "[Password]";
```

- Breakinbg down jdbcURL:

    - **Protocol -** (i.e. "jdbc:mysql://")

    - **Server Address and Port Number -** (i.e. "localhost:3306")

    - **Database Name -** (i.e. "database_name")

    - Putting it all together: ***"jdbc:mysql://localhost:3306/database_name"***