Chapter # 01: Getting Started with Java

1.1 Introduction:

Java is a high-level, object-oriented programming language that was created in the middle of the 1990s by Sun Microsystems, which is now owned by Oracle Corporation. Java is a well-liked platform for creating reliable and scalable programs because of its portability, security, and stability. I'll give a thorough review of Java in this introduction, including its background, characteristics, and potential uses.

1.2 History of Java:

Sun Microsystems first made Java available in 1995, and it was intended to be a platform-neutral programming language that could be used with any hardware or operating system. James Gosling oversaw a group of programmers who created the language. Java was developed as a replacement for C++ and was intended to be more user-friendly, secure, and dependable.

1.3 Features of Java:

Java has many features that make it a popular choice for programming. Some of the most notable features of Java include:

1. **Object-oriented**: Java is an object-oriented programming language, which means that it uses objects to represent data and behavior.
2. **Platform-independent:** Java can run on any device or operating system, making it a highly portable programming language.
3. **Garbage Collection:** Java features automatic memory management, which means that it can manage the memory used by a program automatically.
4. **Multi-threaded:** Java supports multi-threading, which allows multiple threads to run simultaneously within a program.
5. **Security:** Java has a strong focus on security and includes many features that make it difficult to write insecure code.
6. **Robust:** Java is designed to be robust and can handle errors and exceptions more gracefully than many other programming languages.
7. **Applications of Java:** Java is used for a wide range of applications, from mobile app development to enterprise-level systems. Some of the most common applications of Java include:
8. **Web Development:** Java is a popular choice for building web applications, including server-side programming and web services.
9. **Mobile Development:** Java is used to build Android apps, one of the most popular mobile operating systems.
10. **Enterprise-level systems:** Java is often used to build large-scale enterprise systems that require high levels of scalability, reliability, and security.
11. **Desktop Applications:** Java can be used to build desktop applications for Windows, Mac, and Linux operating systems.

1.4 Why Choose Java:

Millions of developers utilize the well-liked, adaptable, and portable programming language Java on a global scale. It is a great option for creating complicated programs that are dependable and safe because of its object-oriented methodology, automatic memory management, and emphasis on security. There are numerous benefits to selecting Java as a programming language. Some of the strongest arguments are as follows:

1. **Popularity:** Millions of developers use Java, one of the most widely used programming languages in the world. This indicates that there are many resources, such as books, online courses, and discussion boards, available for learning Java.

2. **Versatility**: Java is a flexible language that can be used to create a variety of applications, including desktop programs, mobile apps, and complex business systems.

3. **Portability**: Java is a very portable programming language because of its "write once, run anywhere" philosophy, which allows Java code to execute on any hardware or operating system.

4. **Object-Oriented**: Programming in Java is object-oriented, which means that objects are used to represent data and action. Complex code management and the creation of modular, reusable components are made simpler as a result.

5. **Job Opportunities**: Several businesses are looking for skilled Java developers because Java is in high demand on the employment market. Many work opportunities, from entry-level positions to senior developer responsibilities, can be found by learning Java.

6. **Community Support:** The developer community for Java is large and vibrant. These individuals often contribute to open-source projects, create new libraries and frameworks, and help and mentor other developers.

7. **Security:** With features like a secure class loader and a security manager that aid in preventing the execution of unauthorized code, Java places a high priority on security.

8. **Automatic Memory Management:** Java has automatic memory management, which enables it to automatically manage the memory an application uses. This facilitates the development of reliable, bug-free code that is devoid of memory leaks and other memory-related problems.

1.5 Comparison of Java:

Each programming language is best suited for a certain class of jobs, and each language has a unique set of strengths and disadvantages. Every programming language has a distinct purpose because it was developed to effectively handle a certain problem. Each language has its own advantages and disadvantages, and some jobs are better suited to certain languages than others. It

is crucial that programmers select the appropriate language for the task at hand as a result. With Java, we have contrasted a few other languages.

1.5.1 Java vs. Python:

1. **Syntax**: Java's syntax is more complicated than Python's, which might make learning it more challenging for beginners. Python is renowned for its simple syntax and may be more intuitive for programmers who are just starting out.

2. **Performance:** Because Java is compiled to bytecode and run via the Java Virtual Machine, it is often faster than Python. The performance of Python, on the other hand, can be slower because it is an interpreted language.

3. **Versatility:** Python is more adaptable and may be used for a larger range of applications, including web development, data analysis, and machine learning, whereas Java is frequently used for developing large-scale enterprise systems.

4. **Libraries and Framework**s: Large libraries and frameworks are available for developers to use in both Java and Python, while Python has an advantage in the fields of data science and machine learning.

1.5.2 Java vs. C++:

1. **Performance:** Since C++ is a lower-level language and can directly access system resources, it performs generally faster than Java, which must run through the Java Virtual Machine, which can add overhead.

2. **Memory Management:** Java has automatic memory management, which facilitates the creation of reliable, error-free code that is free of memory leaks and other memory-related problems. The manual memory management required by C++, on the other hand, can make it more difficult to build reliable, error-free code.

3. **Flexibility:** Java is more flexible than C++ and may be used to create a variety of applications, from enterprise systems to mobile apps. Contrarily, C++ is frequently employed for applications that require high performance and systems programming.

4. **Object-Oriented Programming**: Java and C++ are both object-oriented programming languages, but Java offers a larger and more adaptable class hierarchy system than C++.

1.6 Compilation in Java:

The compilation process in Java is the process of translating source code written in the Java programming language into machine code that can be executed by a computer.

The compilation process in Java typically involves the following steps:

1. **Writing Java code**: The first step is to write Java code using a text editor or an integrated development environment (IDE) like Eclipse or NetBeans.

2. **Compilation:** The next step is to use a Java compiler, like javac, to compile the Java code. The Java compiler converts Java code into bytecode, a form of the language that can be run by a Java Virtual Machine at a low level (JVM).

3. **Bytecode:** Java's bytecode can be run on any platform that has the Java Virtual Machine (JVM) installed since it is a platform-neutral binary format.

4. **Class files:** The Java bytecode is stored in class files, which have a .class extension. The class files contain the compiled Java code and are used by the JVM to execute the Java program.

5. **Loading:** The JVM loads the class files into memory when the program is executed.

6. **Verification:** Before executing the bytecode, the JVM verifies the bytecode to ensure that it is safe to execute and does not violate the Java security model.

7. **Execution:** Finally, the JVM executes the bytecode by interpreting the bytecode or by compiling it to machine code, depending on the JVM implementation.

The compilation process in Java involves translating Java code into bytecode, which is stored in class files and executed by a Java Virtual Machine (JVM).

1.7 Type Handling in Java:

Type handling in Java relates to how the language controls the many data kinds that can be incorporated into a program. The data types of variables and expressions must be defined before they are used in a program since Java is a statically typed language.

Primitive types and reference types are the two basic forms of data in Java. Boolean, Byte, Short, Int, Long, Float, Double, and Char are examples of primitive types. For variables that contain straightforward values like numbers, characters, or true/false values, these kinds serve as the fundamental building blocks. Reference types are types that are constructed on top of primitive kinds, such as arrays, classes, interfaces, and others. Reference types are employed to build multivalued, complicated data structures.

Both at compile time and at runtime, Java offers type checking. The Java compiler verifies the compatibility of the various data types used in the program during compilation. An error is produced during compilation if the types are incompatible. This aids in the early detection of faults during the development process.

Java employs type information at runtime to make sure the program functions properly. Java, for instance, verifies the type of the arguments supplied to a method when it is called. A runtime error is produced if the parameters are of the incorrect type.

Java additionally provides type casting, which enables you to change a variable's type. Exact type casting, in which you specify the type to which you want to cast the variable, or implicit type casting, in which Java types the variable for you, are both possible.

Example of Type Handling in Java

```java
public class TypeHandlingExample {
    public static void main(String[] args) {
        // Declare and initialize primitive types
        int myInt = 5;
        double myDouble = 3.14;
        boolean myBoolean = true;
        char myChar = 'a';

        // Declare and initialize reference types
        String myString = "Hello, world!";
        int[] myIntArray = {1, 2, 3};

        // Type casting
        // Explicit casting from double to int
        int castedInt = (int) myDouble;
        System.out.println(castedInt); // Output: 3

        // Implicit casting from int to double
        double castedDouble = myInt;
        System.out.println(castedDouble); // Output: 5.0
    }
}
```

Chapter # 02: Object Oriented Design with Java

2.1 Classes:

Classes are a fundamental concept in object-oriented programming (OOP) and provide a way to encapsulate related data and behavior into a single unit. A class is essentially a blueprint or template for creating objects, which are instances of the class.

In Java, classes are used to define objects, which contain both data (known as instance variables or fields) and behavior (known as methods). These objects can be used to model real-world entities or concepts, such as a person, a bank account, or a car.

Java classes have the following primary advantages:

1. **Abstraction:** Classes provide you the ability to mask the complexity of the implementation that lies beneath and offer a streamlined user interface for interacting with objects. This makes using and understanding the code simpler.

2. **Encapsulation:** By enclosing an object's data and behavior in a class, you can shield it from outside disturbance and make it simpler to manage and update.

3. **Reusability:** Reusable classes make it simpler to develop modular, maintainable code because they can be used in different sections of one program or in separate applications altogether.

2.2 Class Creation:

To create a class in Java, you use the **class** keyword followed by the name of the class, which should be in CamelCase format (e.g. Person, BankAccount). Here's an example of a simple class in Java:

Sample Person Class

```java
public class Person {
    // instance variables
    private String name;
    private int age;

    // constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // methods
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

In this example above, the Person class has two instance variables (name and age) and three methods (getName(), setName(), and getAge(), setAge()). The constructor method is a special method that is called when a new instance of the class is created, and is used to initialize the instance variables.

2.3 Attributes and Methods in class

In Java, methods and attributes are the two main components of a class that encapsulate the behavior and state of objects, respectively.

Attributes (also known as instance variables or fields) represent the state of an object, and define the properties or data that are associated with the object. These can be any data type, such as primitive types (int, boolean, etc.) or complex types (other objects or arrays). Attributes are declared inside a class, and can be accessed using an object of the class. For example, here is a simple class with two attributes:

Attribute Sample in Java

```java
public class Person {
    private String name;
    private int age;
}
```

In this example, the Person class has two attributes: name, which is a String, and age, which is an int. These attributes are declared as private, which means they can only be accessed from within the class itself. We can also create public attributes which can accessed from outside the class. To Access the private members of the class we have to create Setter and Getters. Which are just methods that return or set the value of an attribute pass by the user through the arguments. We will discuss setter and getters after methods.

Methods, on the other hand, represent the behavior of an object, and define the actions that can be performed on the object. Methods are declared inside a class, and can be called using an object of the class. Methods can also take parameters (input) and return values (output). For example, here is a simple class with two methods:

Sample Methods Class

```java
public class Person {
    private String name;
    private int age;

    public void sayHello() {
        System.out.println("Hello, my name is " + name);
    }

    public void setAge(int newAge) {
```

```
        age = newAge;
    }
}
```

In this example, the Person class has two methods: sayHello(), which prints a greeting to the console, and setAge(), which sets the value of the age attribute to a new value. The sayHello() method does not take any parameters and does not return any value, while the setAge() method takes an integer parameter and does not return any value.

Just like attributes, methods can also be private or public. Which means we can set which method we want to allow access outside the class.

2.4 Setter and Getters:

Getters and setters are unique Java methods that are used to read and change the values of private attributes (instance variables) of a class. These methods are sometimes referred to as accessor and mutator methods.

Getters are methods that let you extract an attribute's value from an object. The pattern get<AttributeName>, where AttributeName is the name of the attribute, is generally used to name them. Here is an illustration of a getter method for a Person class's name attribute:

Sample Getters Class

```
public class Person {
    private String name;

    public String getName() {
        return name;
    }
}
```

Setters, on the other hand, are methods that allow you to modify the value of an attribute in an object. They are typically named using the pattern set<AttributeName>, where AttributeName is the name of the attribute. Here's an example of a setter method for the age attribute of a Person class:

Sample Setters Class

```
public class Person {
    private int age;

    public void setAge(int newAge) {
        age = newAge;
    }
}
```

Using getters and setters is considered good practice in Java, because it provides a way to control access to the attributes of a class and enforce data encapsulation. By making the attributes private and providing getter and setter methods, you can ensure that the state of an object can only be modified in a controlled way, which can help to prevent bugs and ensure the correctness of your code.

2.5 This Keyword:

In Java, **this** is a special keyword that refers to the current object instance of the class in which it is used. The **this** keyword can be used in several ways:

1. **To refer to the current object's attributes:** For example, if a method has a parameter with the same name as an attribute of the class, you can use this to refer to the attribute instead of the parameter:

```
public class Person {
    private String name;

    public void setName(String name) {
        this.name = name;
    }
}
```

2. **To call a constructor from another constructor:** For example, if a class has multiple constructors, you can use this to call one constructor from another:

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public Person(String name) {
        this(name, 0); // calls the other constructor with default age
value
    }
}
```

3. **To return the current object from a method:** For example, if a method needs to return the current object, you can use this to return a reference to the current object:

```java
public class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }

    public Person self() {
        return this;
    }
}
```

The **this** keyword in Java is a way to refer to the current object instance of a class, and can be used to disambiguate between variables with the same name, call a constructor from another constructor, or return a reference to the current object.

2.6 Constructor Overloading:

The ability to declare numerous constructors with the same name but different parameters is known as constructor overloading in Java. Programmers can create objects using various initialization values depending on the particular requirements of the program thanks to constructor overloading.

A Java class is considered to have overloaded constructors if it defines numerous constructors with the same name but distinct argument lists. Based on the quantity, arrangement, and types of the parameters that each of these constructors accepts, the Java compiler distinguishes between them.

Take the Java class with two constructors as an illustration:

Contructor Overloading Class

```java
public class Person {
    private String name;
    private int age;

    public Person() {
        name = "";
        age = 0;
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // other methods...
}
```

The Java compiler determines which constructor to use based on the number and types of arguments passed to the constructor. If no arguments are passed, the first constructor is called. If two arguments of the correct types are passed, the second constructor is called.

Constructor overloading is a powerful feature of Java that allows programmers to create classes with flexible and customizable initialization options. By providing multiple constructors with different parameter lists, programmers can make their classes more versatile and easier to use.

Chapter # 03: Getting Deeper with Class Objects

3.1 Objects of Class:

In Java, objects are instances of a class, which encapsulate both the data (attributes) and behavior (methods) of the class. Objects are created using the new keyword, which allocates memory for the object and initializes its attributes to their default values.

Here's an example of a simple Person class and how to create an object of that class:

Sample Objects Of Class

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void sayHello() {
        System.out.println("Hello, my name is " + name);
    }
}

// Creating an object of the Person class
Person p = new Person("Alice", 30);
p.sayHello(); // prints "Hello, my name is Alice"
```

Java objects make it possible to package data and behavior into reusable and adaptable pieces, which can make designing and implementing complicated systems easier. You can develop custom types that meet the requirements of your application and represent real-world entities naturally and logically by defining classes and producing objects of those classes.

3.2 Static and Dynamic Functions/Variables:

Static and dynamic functions and variables are concepts in computer programming that refer to how data and functions are defined and used in a program.

- **Static Variables/Function:**
  Static functions and variables are defined as part of a class or object, and they exist independently of any instance of that class or object. They are stored in a fixed location in memory and retain their values throughout the program's lifetime.
  Static functions and variables are typically used for values that do not change during program execution, such as program-wide constants or utility functions that do not depend on the state of the object. Static functions and variables in Java are accessed by using the class name rather than creating an object of the class.
  Static variables and functions are associated with the class itself rather than with any particular instance of the class. This means that all objects of the class share the same static variable or function. They are initialized when the class is loaded and are available throughout the lifetime of the program.
  Static variables and functions can be accessed using the class name, without the need to create an object of the class. For example, in the following code, the static variable counter and the static function incrementCounter() are associated with the MyClass class:

```
public class MyClass {
    static int counter = 0;
    static void incrementCounter() {
        counter++;
    }
}
```

  To access the static variable or function, you can use the following syntax:

```
MyClass.counter = 10;
MyClass.incrementCounter();
```

- **Dynamic Functions/Variables:**
  Dynamic variables and functions are those that come with an instance of a class or object. They are destroyed when the instance of the class is destroyed and only exist when an instance of the class is created. Usually, dynamic functions and variables are used to hold values that are unique to a given instance of the class, such as an object's state or data that is produced while a program is being executed.
  An object of the class is used to access dynamic functions and variables. Java uses the word "this" to denote the current instance of the class, and the dot notation is used to access dynamic functions and variables.

Dynamic variables and functions are associated with the instance of the class. Each instance of the class has its own copy of the dynamic variable and can call the dynamic function independently of other instances of the class.
Dynamic variables and functions can be accessed using an object of the class. For example, in the following code, the dynamic variable x and the dynamic function printX() are associated with each instance of the MyClass class:

```
public class MyClass {
    int x;
    void printX() {
        System.out.println(x);
    }
}
```

To access the dynamic variable or function, you need to create an object of the class, as follows:

```
MyClass obj1 = new MyClass();
obj1.x = 5;
obj1.printX(); // Ouputs 5

MyClass obj2 = new MyClass();
obj2.x = 10;
obj2.printX(); // Outputs 10
```

3.3 Class Design:

When it comes to designing classes, there are a few key principles that can help guide the process:

- **Identify the responsibilities and behaviors of the class:**
  You must be completely aware of the purpose of the class before you can begin to create it. What duties does it have? What actions should it take? What issues should it address? This might assist you in determining the course's objectives and ensuring that it has a distinct purpose.

- **Consider the relationships between classes:**
  It's crucial to take into account how different classes in the system interact with one another because classes don't exist in a vacuum. How is the class dependent on other entities? With which other classes does it work together? You can create a class that is logically and effectively integrated into the wider system by understanding these relationships.

- **Use abstraction to simplify complexity:**
  Classes can easily grow complex, thus it's crucial to employ abstraction to make their design simpler. You may reduce duplication and improve modularity by defining a set of shared behaviors that can be shared by different classes with the aid of abstract classes and interfaces.

- **Ensure cohesion and encapsulation:**
  A well-designed class will have cohesiveness, which is the quality of having all of its duties tied to a single, clearly stated goal. Encapsulation is crucial since it enables you to keep class internals private and only disclose what's required for use by other classes.

- **Test early and often:**
  Testing is an important part of class design, as it allows you to identify and correct issues early in the development process. As you design your class, think about how you'll test it and what scenarios you'll need to cover to ensure that it works as expected.

Overall, class design is a complex process that requires careful consideration of many factors. By following these principles, you can create classes that are well-structured, maintainable, and easy to use.

Chapter # 04: Unit Testing

4.1 Test Driven Development:

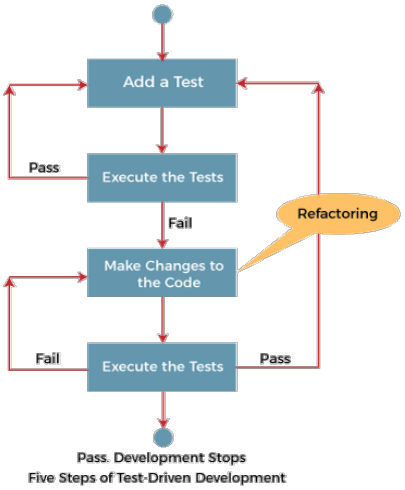
The basic idea behind test driven development (TDD) is to write tests before writing the actual code. This might seem counterintuitive at first, but it can be a powerful way to ensure that your code is correct and reliable. It is founded on the straightforward idea that creating and fixing flawed tests comes before writing new code (before development). To pass tests, we create a tiny bit of code at a time, so it is beneficial for developers to write as little duplicate code as possible. Tests are only requirements that must pass testing in order to be fulfilled. Here are some benefits of TDD:

- You may identify bugs and design issues early in the development process by writing tests before writing the code. You won't have to go back and address bugs later, which can save time and resources in the long term.
- While writing tests first, you must consider how the code should function from the user's point of view. This can assist you in creating more user-friendly and intuitive applications.
- You have to consider how to divide the problem into more manageable pieces when you develop tests initially. This can promote modularity and improve the maintainability of your code.
- You run the risk of unintentionally introducing flaws when refactoring code. You can make sure that your modifications don't affect any existing functionality by using a thorough test suite.

Now let's dive into the TDD process in more detail:

1. **Write a failing test case:** The first step in TDD is to write a test case that covers a specific behavior or functionality. This test case should fail because you haven't written any code yet.
2. **Write the code:** Once you have a failing test case, you can start writing the code to make it pass. The code should be designed to pass the specific test case you wrote.

3. **Run the tests:** After writing the code, run the test cases to ensure that they pass. If any test case fails, go back to step 2 and modify the code until all test cases pass.
4. **Refactor the code:** Once all the test cases pass, you can refactor the code to make it more efficient or easier to understand. The goal of this step is to improve the code without changing its behavior.
5. **Repeat the process:** Repeat the process for the next feature or functionality you want to add to the code.



Five Steps of Test-Driven Development

In Java, you can use testing frameworks such as JUnit, Mockito, or TestNG to write and run tests. These frameworks provide a way to define test cases and assertions that will be used to verify the correctness of your code. We will discuss JUnit later.

4.2 JUnit Testing:

With the help of the popular testing framework JUnit, developers may create automated tests for their code to validate its accuracy and usefulness. It is simple to confirm the anticipated behavior of specific components or modules of an application using the set of tools and protocols provided by JUnit.

At its core, JUnit is a set of Java classes and annotations that let programmers create test cases that the JUnit test runner can execute automatically. One or more test methods that are marked with the @Test annotation to denote their inclusion in a JUnit test suite make up a typical JUnit test case. The JUnit test runner then automatically locates and runs these test methods, reporting the outcomes of each test case and determining if the test suite was successful or unsuccessful in general.

JUnit has a variety of other annotations and tools, in addition to the @Test annotation, that make it simple to create reliable and efficient test cases. Developers can create methods that are executed prior to or following each test method, for instance, by using the @Before and @After annotations. This enables them to provide the proper environment and decompose any resources that their tests consumed. The @BeforeEach and @AfterEach annotations, which provide for even finer-grained control over the testing process, can be used to create methods that are executed before or after each individual test case.

JUnit also provides a number of assert methods that can be used to check the expected behavior of individual components or modules of an application. These assert methods include assertEquals, assertNotEquals, assertNull, assertNotNull, assertSame, and assertNotSame, among others, and make it easy to verify that the output of a particular method or component matches the expected result.

For Example: We want to create a Calculator and it has a Add Function that adds two numbers.

```
public class MyMath {
    public static int add(int a, int b) {
        return a + b;
    }
}
```

We can use Junit testing to test the correctness of this Add function.

```
import org.junit.Test;
import static org.junit.Assert.*;

public class MyMathTest {

    @Test
    public void testAdd() {
        int result = MyMath.add(2, 3);
        assertEquals(5, result);
    }
}
```

Overall, JUnit is a powerful and flexible testing framework that provides developers with the tools and conventions necessary to write effective and reliable automated tests for their Java applications. By using JUnit to test their code, developers can ensure that their applications are robust and function as intended, improving overall code quality and reducing the risk of bugs and errors in production environments.

4.3 JUnit vs Driver Testing:

Driver testing and JUnit testing are two different approaches to testing software, and they have some similarities and differences.

Driver testing is a sort of integration testing in which various software modules or components are tested as a unit. The majority of the time, when developing test cases for drivers, developers replicate user interactions with the software by simulating operations such button clicks, data entry, and page or screen switching. Driver testing is to find defects and errors that may result from system interactions between various components and to make sure that the program functions as intended from the user's point of view.

JUnit testing, on the other hand, is a type of unit testing, where individual components or modules of software are tested in isolation from the rest of the system. In JUnit testing, developers write automated test cases that verify the behavior of individual methods or classes, using a testing framework like JUnit to organize and run the tests. The goal of JUnit testing is to catch bugs and errors early in the development process, before they can affect other parts of the system.

The scope of the tests is a critical difference between driver testing and JUnit testing. Driver tests often examine a wider variety of functionality and the relationships between various system components, whereas JUnit tests tend to be more narrowly focused on specific methods or classes. Driver tests typically take longer and need more complexity to create than JUnit tests because they must manage the state of the system being tested and simulate user interactions. JUnit tests, on the other hand, can be written more quickly and with less effort because they can be performed independently and do not need as much setup and debugging as driver tests.

In summary, driver testing and JUnit testing are two different approaches to testing software, each with its own strengths and weaknesses. While driver tests are useful for verifying the behavior of the software as a whole, JUnit tests are essential for catching bugs and errors in individual components or modules of the system, and ensuring their correctness and functionality.

Chapter # 06: Inheritance

6.1 Inheritance:

Inheritance is a fundamental concept in object-oriented programming that enables a new class to be based on an existing class. Inheritance allows the child class to inherit the properties and methods of the parent class, which it can then use, modify, or extend.

The basic idea behind inheritance is that a child class is a more specialized version of its parent class. For example, a car class may have a number of properties and methods, and a sports car class could be a child class of the car class that inherits all of its properties and methods, while also adding new properties and methods that are specific to sports cars.

By using inheritance, programmers can reuse existing code, reduce duplication, and create more organized and modular code structures. It also allows for greater flexibility and extensibility in

software development, as new child classes can be created based on existing classes with minimal effort.

6.2 Purpose of Inheritance:

The main purpose of inheritance in programming is to promote code reuse and to facilitate the creation of new classes that are similar to existing classes but with some additional or modified functionality. Some of the Main purpose of Inheritance are given below:

1. **Code reuse:**
   When a new class is established by deriving from an existing class, all of the parent class's properties and methods are automatically passed down to the child class, allowing the child class to reuse the parent class's functionality without having to write any new code.

2. **Simplification of code**:
   Inheritance enables developers to create more simplified and modular code structures, by allowing them to break down complex classes into smaller, more specialized classes that are easier to manage and maintain.

3. **Improved efficiency:**
   By minimizing the amount of code that needs to be developed, tested, and reused, inheritance can increase the effectiveness of software development.

4. **Polymorphism**:
   Polymorphism, or the ability to treat objects of various classes as though they were of the same class, is one of the core elements of object-oriented programming that inheritance permits. This can simplify coding and increase the maintainability of programs.

Overall, inheritance is a powerful tool that allows developers to build more flexible and efficient software systems by leveraging the strengths of existing code and creating new classes that are better suited to their specific needs.

6.3 Complex Classes Through Inheritance:

By establishing new classes that inherit the attributes and methods of existing classes while simultaneously adding new properties and methods that are exclusive to the new class, developers can use inheritance to create more complicated classes. When using inheritance to create more complicated classes, you should do the following steps:

- **Identify the existing class:** First, identify an existing class that contains some of the functionality that you want to include in your new class.
- **Decide on the relationship:** Decide on the relationship between the new class and the existing class. Will the new class be a subtype of the existing class, or will it simply use the existing class as a template?
- **Create the new class:** Create the new class, making sure to include any new properties and methods that are specific to the new class.

- **Inherit from the existing class:** Use the inheritance mechanism to make the new class inherit the properties and methods of the existing class. In most programming languages, this is done using the "extends" keyword.
- **Override and extend methods:** Override any methods in the new class that need to be modified or customized, and extend any methods in the existing class that need to be augmented with additional functionality.
- **Test the new class:** Test the new class thoroughly to ensure that it is working as intended, and that all of the new and inherited properties and methods are functioning correctly.

Developers can utilize inheritance to create more intricate classes that are customized to their own requirements by following these steps while also gaining access to the functionality and effectiveness of pre-existing code.

6.4 Chaining:

Programmers can invoke numerous methods on an object in a single line of code by using the chaining approach. When creating a class, this strategy can be especially helpful because it makes the code simpler and more readable. Chaining can assist in creating a class in the following ways:

- Chaining allows developers to write code that is more concise and easier to read, as it eliminates the need to create temporary variables to hold intermediate results.
- By chaining methods together, developers can create code that reads more like natural language, making it easier to understand and maintain.
- Chaining can make it easier to debug code, as it allows developers to pinpoint the exact location of any errors or exceptions that may occur.
- Chaining can also improve the efficiency of code, as it reduces the amount of overhead associated with creating and managing temporary variables.

Here is a Little Example of Chaining in Java:

```java
public class Person {
    private String name;
    private int age;
    private String address;

    public Person setName(String name) {
        this.name = name;
        return this;
    }

    public Person setAge(int age) {
        this.age = age;
        return this;
    }

    public Person setAddress(String address) {
        this.address = address;
        return this;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public String getAddress() {
        return address;
    }
}

// Usage
Person person = new Person()
    .setName("ABC")
    .setAge(30)
    .setAddress("Street 10, London");

String name = person.getName();
int age = person.getAge();
String address = person.getAddress();
```

6.5 Dynamic Dispatching:

The practice of choosing the best method implementation to be executed at runtime based on the actual type of the object the method is being called on is known as dynamic dispatching. Runtime polymorphism or "late binding" are other names for this.

Due to the fact that it enables subclasses to replace the methods of their superclass with alternative implementations, dynamic dispatching in Java is strongly tied to inheritance. The JVM will first search for a method implementation in the subclass when a method is called on an object of a subclass. If it does, it will employ it rather than the superclass implementation. This preserves the functionality of the superclass while enabling subclasses to modify the behavior of their inherited methods. Here is an Example of Dynamic Dispatching:

```
class Animal {
    public void speak() {
        System.out.println("I am an animal.");
    }
}

class Dog extends Animal {
    @Override
    public void speak() {
        System.out.println("Woof!");
    }
}

class Cat extends Animal {
    @Override
    public void speak() {
        System.out.println("Meow!");
    }
}

Animal animal1 = new Animal();
animal1.speak(); // prints "I am an animal."

Animal animal2 = new Dog();
animal2.speak(); // prints "Woof!"

Animal animal3 = new Cat();
animal3.speak(); // prints "Meow!"
```

6.6 Abstract and Concrete Classes:

A class that cannot be instantiated directly but may be used as a foundation class to create other classes is known as an abstract class in object-oriented programming. An abstract class may have one or more abstract methods with no implementation, serving as a template for other classes to derive from. The abstract keyword is used to declare abstract classes.

Here is an Example of Abstract Class in Java:

```
abstract class Shape {
    public abstract double getArea();

    public abstract double getPerimeter();

    public void print() {
        System.out.println("This is a shape.");
    }
}

class Rectangle extends Shape {
    private double length;
    private double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    public double getArea() {
        return length * width;
    }

    public double getPerimeter() {
        return 2 * (length + width);
    }
}

Shape shape = new Rectangle(3.0, 4.0);
double area = shape.getArea();
double perimeter = shape.getPerimeter();
shape.print(); // prints "This is a shape."
```

The usage of abstract classes to create a common interface for a collection of related classes while enabling each subclass to offer its own implementation of the abstract methods is illustrated by the example given above. Subclasses can inherit from the abstract class's concrete methods, which enables code reuse and reduces duplication.

Contrarily, a concrete class is one that may be created directly and may or may not be descended from an abstract class. Every method in a concrete class, including any inherited methods from its base classes, has an implementation. Here is an Example:

```
class Dog {
    private String name;
    private int age;

    public Dog(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void bark() {
        System.out.println(name + " is barking.");
    }

    public void fetch() {
        System.out.println(name + " is fetching.");
    }

    public void sleep() {
        System.out.println(name + " is sleeping.");
    }
}

Dog dog = new Dog("Fido", 3);
dog.bark(); // prints "Fido is barking."
dog.fetch(); // prints "Fido is fetching."
dog.sleep(); // prints "Fido is sleeping."
```

When we have a specific implementation that we wish to use without additional customization, concrete classes come in handy. On the other side, abstract classes come in handy when we wish to specify a common interface for a collection of related classes while yet allowing each subclass to offer its own implementation of the abstract methods.

6.7 Composition and Inheritance:

In object-oriented programming, composition and inheritance are two distinct methods for achieving code reuse and creating connections between classes. The primary distinction between them is that while inheritance entails the creation of a subclass that derives properties and behavior from a parent class, composition entails building a class from other objects.

In composition, a class is constructed using other objects, which are typically passed as constructor arguments or instantiated within the class. The composed objects are used to provide specific functionality or behavior to the class. For example, a Car class may be composed of an Engine object, a Transmission object, and a Chassis object, each responsible for a different aspect of the car's behavior. Here is an example:

```
public class Car {
    private Engine engine;
    private Transmission transmission;
    private Chassis chassis;

    public Car(Engine engine, Transmission transmission, Chassis chassis) {
        this.engine = engine;
        this.transmission = transmission;
        this.chassis = chassis;
    }

    public void start() {
        engine.start();
        transmission.shiftToDrive();
        chassis.move();
    }

    public void stop() {
        engine.stop();
        transmission.shiftToPark();
        chassis.stop();
    }
}

public class Engine {
    public void start() {
        System.out.println("Engine started.");
    }

    public void stop() {
        System.out.println("Engine stopped.");
    }
```

```
}

public class Transmission {
    public void shiftToDrive() {
        System.out.println("Shifted to drive.");
    }

    public void shiftToPark() {
        System.out.println("Shifted to park.");
    }
}

public class Chassis {
    public void move() {
        System.out.println("Moving.");
    }

    public void stop() {
        System.out.println("Stopped.");
    }
}

Engine engine = new Engine();
Transmission transmission = new Transmission();
Chassis chassis = new Chassis();
Car car = new Car(engine, transmission, chassis);
car.start(); // starts the car
car.stop(); // stops the car
```

Inheritance, on the other hand, involves creating a subclass that inherits attributes and behavior from a parent class. The subclass can then add or modify the inherited behavior, as well as define its own behavior. For example, a Rectangle class may inherit from a Shape class and add its own behavior to calculate area and perimeter. Here is an example of it as well:

```
public class Shape {
    private int x;
    private int y;

    public Shape(int x, int y) {
        this.x = x;
        this.y = y;
    }
```

```
    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public void draw() {
        System.out.println("Drawing shape at (" + x + ", " + y + ")");
    }
}

public class Rectangle extends Shape {
    private int width;
    private int height;

    public Rectangle(int x, int y, int width, int height) {
        super(x, y);
        this.width = width;
        this.height = height;
    }

    public int getWidth() {
        return width;
    }

    public int getHeight() {
        return height;
    }

    public void draw() {
        System.out.println("Drawing rectangle at (" + getX() + ", " + getY() +
") with width " + width + " and height " + height);
    }

    public int getArea() {
        return width * height;
    }

    public int getPerimeter() {
        return 2 * (width + height);
    }
}
```

```
Shape shape = new Shape(10, 20);
shape.draw(); // draws shape at (10, 20)

Rectangle rectangle = new Rectangle(30, 40, 50, 60);
rectangle.draw(); // draws rectangle at (30, 40) with width 50 and height 60
System.out.println("Area: " + rectangle.getArea()); // prints "Area: 3000"
System.out.println("Perimeter: " + rectangle.getPerimeter()); // prints
"Perimeter: 220"
```

Both composition and inheritance have their own strengths and weaknesses, and the choice between them ultimately depends on the specific needs of the application being developed. Here are some of the Key Differences between them:

- Inheritance entails constructing a subclass that draws characteristics and behavior from a parent class, whereas composition involves using objects to build up a class.
- In contrast to inheritance, which keeps the subclass closely tied to the parent class, composition allows for the easy swapping out or replacement of the composed objects.
- In contrast to inheritance, which can result in a rigid and inflexible class hierarchy, composition permits greater flexibility and modularity.
- While inheritance is better suitable for creating class hierarchies when there is a clear "is-a" relationship between classes, composition tends to be better suited for creating complicated systems with several replaceable elements.

Chapter # 7: Interfaces

7.1 Interface:

In Java, an interface is a collection of abstract methods (methods without implementation) and constants (public static final variables). An interface defines a contract that a class can choose to implement. When a class implements an interface, it agrees to provide an implementation for all the abstract methods defined in the interface.

An interface can be thought of as a blueprint or a set of guidelines for how classes should behave. It allows for polymorphism, which means that objects of different classes that implement the same interface can be treated in a uniform way.

Interfaces are declared using the interface keyword, and can be implemented by a class using the implements keyword. An interface can extend another interface using the extends keyword, allowing for the creation of a hierarchy of interfaces.

Interfaces offer a way to obtain some of the advantages of multiple inheritance, such as code sharing between classes, without some of the disadvantages, such as the potential for conflicts between inherited methods, since Java does not permit multiple inheritance of classes.

Here is an example interface in Java:

```
public interface Animal {
   public void makeSound();
   public void move();
}
```

We also need to implement the interface so that we can create an instance:

```
public class Dog implements Animal {
    public void makeSound() {
       System.out.println("Bark");
    }

    public void move() {
       System.out.println("Running");
    }
 }
```

Now any instance of Dog can be treated as an Animal:

```
Animal myPet = new Dog();
myPet.makeSound(); // prints "Bark"
myPet.move(); // prints "Running"
```

7.2 Abstract Class and Interfaces:

Both interfaces and abstract classes are used in Java to achieve abstraction and create a blueprint for other classes to follow. However, there are some key differences between them:

1. Implementation: Whereas an interface can only contain abstract methods, an abstract class can contain both abstract and non-abstract methods. Also, whereas an interface cannot give any implementation for its methods, an abstract class can supply implementations for some of its methods.

2. Inheritance: A class can extend only one abstract class, but can implement multiple interfaces. This is because Java does not support multiple inheritance of classes, but allows multiple inheritance of interfaces.
3. Constructor: An abstract class can have a constructor, whereas an interface cannot have a constructor.
4. Access modifiers: In contrast to interface methods and variables, which are always public, abstract class methods and variables might have access modifiers of public, protected, or package-private.
5. Usage: Abstract classes are useful when you want to create a base class that provides some default implementations that can be inherited by its subclasses, whereas interfaces are useful when you want to define a set of methods that must be implemented by any class that wants to adhere to the interface.

In conclusion, abstract classes are useful when you want to create a base class that offers some default implementations that can be inherited by its subclasses, as opposed to interfaces, which are useful when you want to define a set of methods that must be implemented by any class that wants to adhere to the interface. Interfaces can be inherited several times, whereas abstract classes can only be extended by a single subclass, making interfaces more flexible than abstract classes.

7.3 Purpose of an Interface:

In Java, an interface serves as a contract or a set of guidelines for how a class should behave. It defines a set of abstract methods that must be implemented by any class that wants to adhere to the interface.

The purpose of an interface is to provide a way to achieve abstraction and polymorphism. Abstraction means hiding implementation details and providing a high-level view of functionality. Polymorphism means the ability to treat objects of different classes as if they are objects of the same class.

The definition of a class can be distinguished from the specifics of its implementation by creating an interface. You may therefore develop code that isn't dependent on any one particular implementation, which makes it simpler to maintain and alter. By establishing a set of methods that may be implemented by many classes in various ways while still being handled uniformly, interfaces also enable polymorphism.

Moreover, interfaces aid in enforcing a standard set of behaviors among many classes that implement the same interface. Without having to worry about the unique implementation details of each object, this makes it simpler to design code that can function with any object that complies with a particular interface.

Overall, the purpose of an interface in Java is to provide a contract or a set of guidelines for how a class should behave, to separate the definition of a class from its implementation details, and to achieve abstraction and polymorphism.

7.4 When you should choose an interface:

Whenever you wish to specify a set of methods that any class that wants to abide to the interface must implement without providing any implementation details, you would use an interface in Java. When you wish to provide certain of a class's methods a default implementation while leaving some of its other methods as abstract so that their subclasses might implement them, you would use an abstract class.

Here are some specific scenarios where you might choose to use an interface instead of an abstract class:

- When you want to define a common set of behaviors across multiple unrelated classes: An interface allows you to define a common set of methods that must be implemented by any class that wants to adhere to the interface. This is useful when you have multiple unrelated classes that need to adhere to a common set of behaviors.
- When you want to achieve multiple inheritance: Since Java does not allow multiple inheritance of classes, you can achieve multiple inheritance of behaviors by implementing multiple interfaces. This is useful when you want to share behaviors across multiple classes that do not share a common ancestor.
- When you want to define a lightweight contract: Interfaces provide a lightweight way to define a contract for a class without providing any implementation details. This makes interfaces easy to read, understand, and modify.

7.5 Restrictions on Interfaces:

In Java, there are several restrictions on interfaces:

- **An interface cannot be instantiated**
  An interface is a purely abstract concept and cannot be instantiated on its own. Instead, you must create a class that implements the interface and instantiate the class.

- **An interface cannot have constructors:**
  An interface cannot have a constructor because it cannot be instantiated. Therefore, you cannot define any constructor-related logic inside an interface.

- **All methods in an interface are abstract:**
  All methods in an interface are implicitly abstract and must be implemented by any class that implements the interface. They cannot have any implementation details.

- **All variables in an interface are implicitly public, static, and final:**
  All variables defined inside an interface are implicitly public, static, and final. They cannot be changed or modified by any class that implements the interface.

- **An interface cannot have static or instance blocks:**
  An interface cannot have static or instance blocks because it cannot be instantiated. Therefore, you cannot define any block-related logic inside an interface.

- **An interface cannot extend a class:**
  An interface can only extend other interfaces, but not classes. This is because Java does not allow multiple inheritance of classes, but allows multiple inheritance of interfaces.

- **An interface cannot implement another interface:**
  An interface can extend multiple interfaces, but it cannot implement another interface. This is because implementing an interface means providing concrete implementations for its methods, which is not allowed in an interface.

These restrictions are in place to ensure that interfaces remain a purely abstract concept that defines a set of methods and variables that must be implemented by any class that wants to adhere to the interface. This makes interfaces a useful tool for achieving abstraction and polymorphism in Java.

Chapter # 08: The Dangers of Inheritance

8.1 Dangers of Inheritance:

Inheritance in software development refers to the ability of a subclass or child class to inherit characteristics and behaviors from a parent or superclass. While inheritance can be a powerful tool in software development, allowing for code reuse and creating hierarchies of objects with common properties, it can also become dangerous when it gets out of control.

The main danger of inheritance is that it can lead to complex and tightly-coupled code that is difficult to maintain and extend over time. This is particularly true when inheritance hierarchies become deep and complex, with many layers of inheritance and multiple levels of abstraction.

Changes to the base class may have unanticipated effects on all of its child classes, which can be a problem. This may have an impact on the entire codebase, making it challenging to forecast the effects of any changes and possibly resulting in errors and other problems.

The possibility of code repetition and duplication is another risk associated with inheritance. Subclasses may wind up with more code than necessary if they inherit many attributes and behaviors from their parent classes. This could slow down the development process and make the codebase more challenging to read and manage.

Finally, inheritance can also create tight coupling between different parts of the codebase, making it more difficult to test and debug. This can be particularly problematic in large and complex applications, where bugs and other issues can be difficult to track down and fix.

8.2 Avoid Inheritance Dangers

To avoid these dangers, it is important to use inheritance judiciously and to keep inheritance hierarchies as simple and shallow as possible. Code should be designed with modularity and loose coupling in mind, so that changes to one part of the codebase do not have unintended consequences elsewhere. Additionally, developers should take care to avoid code duplication and to keep the codebase well-organized and easy to read and understand.

Here are some of the ways that we can use to avoid dangers of inheritance in Java.

8.2.1 Use Composition Instead of Inheritance:

To create complex objects from simpler ones, composition can be used in place of inheritance. According to this method, an item is constructed from other objects that can be switched in and out as necessary. Due to the fact that modifications to one item do not affect other objects in the composition, this can help prevent the Fragile Base Class problem. For instance:

```
public interface Engine {
    public void start();
    public void stop();
}

public class GasEngine implements Engine {
    @Override
    public void start() {
        System.out.println("Gas engine started.");
    }

    @Override
    public void stop() {
        System.out.println("Gas engine stopped.");
    }
}

public class ElectricEngine implements Engine {
    @Override
    public void start() {
        System.out.println("Electric engine started.");
    }

    @Override
    public void stop() {
        System.out.println("Electric engine stopped.");
    }
}
```

```
public class Vehicle {
    private Engine engine;

    public Vehicle(Engine engine) {
        this.engine = engine;
    }

    public void startEngine() {
        engine.start();
    }

    public void stopEngine() {
        engine.stop();
    }
}
```

8.2.2 Use Abstract Class and Interface:

We can specify similar behavior for related classes using abstract classes and interfaces to save code duplication and boost flexibility. While interfaces offer a method to express a contract between classes without providing any implementation details, abstract classes offer a way to define a base implementation that can be expanded by subclasses. For instance:

```
public interface Animal {
    public void eat();
    public void sleep();
}

public abstract class Mammal implements Animal {
    public abstract void giveSpeak();
}

public class Dog extends Mammal {
    @Override
    public void eat() {
        System.out.println("Dog is eating.");
    }

    @Override
    public void sleep() {
        System.out.println("Dog is sleeping.");
```

```
    }

    @Override
    public void giveSpeak() {
        System.out.println("Dog is Speaking.");
    }
}
```

8.2.3 Use final Keyword:

To prohibit a class, method, or variable from being overridden or altered, use the final keyword. This can assist prevent unexpected behavior and guarantee that important system components are not altered by accident. For instance:

```
public final class ImmutableClass {
    private final int value;

    public ImmutableClass(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

public class MutableClass extends ImmutableClass {
    public MutableClass(int value) {
        super(value);
    }

    // This method will not compile, because ImmutableClass is final
    // and cannot be extended
    @Override
    public int getValue() {
        return super.getValue() + 1;
    }
}
```

8.2.4 Follow the Liskov Substitution Principle (LSP):

A key tenet of object-oriented programming is the Liskov Substitution Principle, which asserts that any object from a derived class should be able to be swapped out for an object from its base class without compromising the program's validity. Or, to put it another way, derived classes ought to be allowed to modify the behavior of their base classes without impairing the system's functionality. Breaking the LSP may result in bugs and ad hoc behavior. For instance:

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public int getWidth() {
        return width;
    }

    public void setWidth(int width) {
        this.width = width;
    }

    public int getHeight() {
        return height;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public int getArea() {
        return width * height;
    }
}

public class Square extends Rectangle {
    public Square(int size) {
        super(size, size);
    }

    @Override
```

```
    public void setWidth(int width) {
        super.setWidth(width);
        super.setHeight(width);
    }

    @Override
    public void setHeight(int height) {
        super.setWidth(height);
        super.setHeight(height);
    }
}

public class Client {
    public static void main(String[] args) {
        Rectangle rect = new Rectangle(5, 10);
        System.out.println("Rectangle area: " + rect.getArea());

        Rectangle square = new Square(5);
        square.setWidth(10);
        System.out.println("Square area: " + square.getArea());
    }
}
```