**Programming Fundamentals – Java Day 1**

* Introduction to OOAD for Java Developers
* Fundamental Programming Structures in Java
* Classes and Objects in Java
* Object Design and Programming with Java
* Java Interfaces
* Java Exception Handling
* Java Collections API
* Java Programming Best Practices

**Programming Fundamentals - C#**

* Learn the fundamentals of C# programming in Visual Studio.
* Using .Net Framework
* Working with variables, data types
* Work with standard programming skills
* Exception Handling in C#
* Object oriented techniques
* Working with Arrays
* Collections in C#
* Generate and test your own classes using the Class Designer and Object Test Bench tools.
* Use delegate types to provide flexibility and type safety.
* Use anonymous types, lambda expressions, extension methods, object initializers, and implicit type declarations

**Programming Fundamentals – Python**

* Introduction to Python
* Variables in Python
* Data Types in Python
* Data Type Conversion in Python
* If-Statements and Conditional Logic in Python
* Loops and Iteration in Python
* File Input/Output in Python
* Functions in Python
* Sending emails with Python
* Multithreading in Python

**Selenium Course Content** **Day 2**

1. Automation:
   * What is Automation Testing  
     • Use of automation Testing  
     • Tools for Automation Testing  
     • Why automation is important for you career?  
     • What is Selenium  
     • Advantage of Selenium  
     • Introduction to IDE, RC WebDriver & Grid
2. WebDriver Interface:
   * Architecture of WebDriver  
     • WebDriver Interface  
     • WebElement Interface  
     • Launching Firefox browser
   * Launching IE  
     • Challenges with IE Browser  
     • IE driver server and path settings  
     • Launching Chrome  
     • Chrome server and path settings
3. Browser & Navigation Commands:
   * How to Open a URL  
     • Verify Page title  
     • Strategy to get the Page Source   
     • Difference between Close & Quit  
     • Ways to Navigate Back & Forward  
     • How to Refresh Page  
     • Another way of Navigating to specific Page
4. WebElement Commands:
   * Difference between FindElement & FindElements  
     • Enter & Clear text from Input field  
     • How Click action works differently on different elements  
     • Managing Input fields, Buttons & Links  
     • Finding all links on the Page  
     • Strategy to check dead links on the page  
     • Extracting More than one object from a page   
     • Extracting Objects from a specific area of a web page   
     • Check if element is Present, Displayed, Enabled or Selected
5. Locators:
   * What are locators  
     • ID, Name, Xpath, CSS etc  
     • Difference between Absolute & Complete Xpath
6. Element Identification:
   * Element Inspector in Mozilla, Chrome and IE  
     • FireBug & FirePath Add-Ons in Mozilla   
     • Selection of Effective XPath  
     • Handling Dynamic objects/ids on the page
7. Tables, Checkboxes & Radio buttons:
   * Identify table rows and columns  
     • Extracting values from a cell  
     • Select class in Selenium  
     • Drop Down Handle  
     • Select multiple values from the list  
     • Select & Deselect operations by Index, Value & Visible Text
8. Selenium Waits, Alert & Switch Windows:
   * Implicit and Explicit waits  
     • How to use Expected Conditions with Waits  
     • PageLoadTimeout & SetScriptTimeout property  
     • Simple use of Thread Sleep  
     •  Ways to handle Simple, Confirmation & Prompt Alert  
     • Difference between Window Handle & Handles   
     • Switching & Closing Windows, Tabs & PopUps
9. TestNG Framework
   * What is TestNG  
     • How to run Test Suite in TestNG  
     • TestNG Reporters  
     • TestNG Assets  
     • TestNG Parameters  
     • Multi Browser testing in TestNG  
     • Parallel testing in TestNG

**Robot Framework Day 3**

* Robot Framework Tutorial – Overview  
  Robot Framework Tutorial – A complete example  
  Robot Framework IDE  
  How to Structure a Scalable And Maintainable Acceptance Test Suite  
  Robot Framework Tutorial – Writing Keyword Libraries in Java  
  Robot Framework Tutorial – Loops, Conditional Execution and more  
  Robot Framework – Testing Windows Applications
* Implementing Keywords in Java  
  Selenium2Library as a drop-in replacement for SeleniumLibrary  
  Integration with Jenkins  
  File Processing  
  Working with Collections  
  Wrap-Up and Conclusion  
  API automation (HTTP library, Rest Instance for Java)

·       DB automation (DB library Java based library)

·       UI automation (Selenium, Sikuli and Test FX library)

·       Robot Framework built-in functions and libraries (FTP, TFTP, HTTP, test data generator and others.)

**BDD Fundamentals Cucumber + Specflow Day 4**

* Building a real web application with Cucumber
* Business Driven Development (BDD)
* Involving the business analysts in your organization
* Defining your project requirements
* The testing ecosystem (IDEs, programming languages, testing frameworks and build libraries)
* Setting up a web application testing system
* Gherkin: writing your stories in a user-friendly language
* Writing features
* Writing scenarios
* Writing step definitions
* Organizing features and scenarios with tagging
* Selenium Webdriver and Specflow, #C
* SpecFlow – Feature File, Gherkin Keywords, Step Definition File
* Specflow – Table, Specflow Hook Methods

**API Testing - Rest Assured Day 5**

* Install REST Assured
* Write your First REST Assured Test
* Running tests through a HTTP Proxy
* Setting logging in tests
* Request Specification
* Response Specification
* Adding Multiple Request & Response Specifications

**REST Assured - Different Request Types**

* GET Request
* POST Request with JSON
* POST Request with XML
* PUT Request
* DELETE Request
* Path Parameters
* Query Parameters

**REST Assured - Assert and Extract from the Response Body**

* Extract explicit data from the body with JSON path
* **REST Assured - Serialization, Schema Validation & Measuring Response Time**
* Validating Response against a XML Schema
* Validating Response against a JSON Schema
* Measuring Response Time in REST Assured
* **Using Groovy Gpath for XML in REST Assured**
* **Authentication**

**Appium Course Content Day 6**

Appium training provides in-depth knowledge on all the core concepts of the mobile test automation from basics to advanced level with real-time experts. You will learn how to automate native and hybrid apps in IOS/Android using Appium tool through hands-on sessions and real-time projects, and this will help you to advance in mobile automation testing domain.

* Introduction to Mobile automation testing  
  APIs and Selenium  
  Automation for Android and IOS devices.  
  Testable items through Appium vs. Selendroid  
  Mobile automation testing tools  
  The API Levels
* JDK Installation, Installation of TestNG on Eclipse  
  Downloading and installing Maven on Windows  
  Downloading Android SDK  
  Configuring Environment Variables

#### [Basic Installation For Appium On Windows](https://mindmajix.com/appium-training#collapse-curriculum-502)

* Downloading Appium for Windows  
  Conguring SDK Manager

#### [Downloading Appium Dependencies, Desired Capabilities, Packages & Activities Information](https://mindmajix.com/appium-training#collapse-curriculum-504)

* Desired Capabilities class  
  Extracting Packages and Activities information  
  Appium Maven Dependencies

### How to create a user-defined method?

Before you can use (call a method), you need to define it.

Here is how you define methods in Java.

public static void myMethod() {

System.out.println(“My Function called”);

}

Here, a method named myMethod() is defined.

You can see three keywords public, static and void before the function name.

* The public keyword makes myMethod() method public. Public members can be accessed from outside of the class. To learn more, visit: [Java public and private Modifiers.](https://www.programiz.com/java-programming/class-objects#private-public)
* The static keyword denotes that the method can be accessed without creating the object of the class. To learn more, visit: *Static Keyword in Java*
* The void keyword signifies that the method doesn’t return any value. You will learn about returning value from the method later in this article.

In the above program, our method doesn’t accept any arguments. Hence the empty parenthesis (). You will learn about passing arguments to a method later in this article.

The complete syntax for defining a Java method is:

modifier static returnType nameOfMethod (Parameter List) {

// method body

}

Here,

* **modifier** - defines access type whether the method is public, private and so on.
* **static** - If you use static keyword in a method then it becomes a static method. Static methods can be called without creating an instance of a class.  
    
  For example, the sqrt() method of standard [Math class](https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html) is static. Hence, we can directly call Math.sqrt() without creating an instance of Math class.
* **returnType** - A method can return a value.  
    
  It can return native data types (int, float, double etc.), native objects (String, Map, List etc.), or any other built-in and user defined objects.  
    
  If the method does not return a value, its return type is void.
* **nameOfMethod** - The name of the method is an [identifier](https://www.programiz.com/java-programming/keywords-identifiers#identifiers).  
    
  You can give any name to a method. However, it is more conventional to name it after the tasks it performs. For example, calculateInterest, calculateArea, and so on.
* **Parameters (arguments)** - Parameters are the values passed to a method. You can pass any number of arguments to a method.
* **Method body** - It defines what the method actually does, how the parameters are manipulated with programming statements and what values are returned. The codes inside curly braces { } is the body of the method.

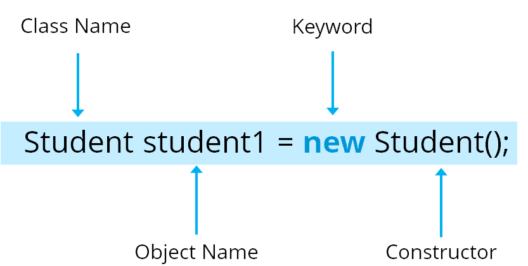
### What is Class?

A class is an entity that determines how an object will behave and what the object will contain. In other words, it is a blueprint or a set of instruction to build a specific type of object.

### What is an Object?

An object is nothing but a self-contained component which consists of methods and properties to make a particular type of data useful. Object determines the behavior of the class. When you send a message to an object, you are asking the object to invoke or execute one of its methods.

From a programming point of view, an object can be a data structure, a variable or a function. It has a memory location allocated. The object is designed as class hierarchies.



class Student()

      void display(); {             // Method

                                 // logic of method

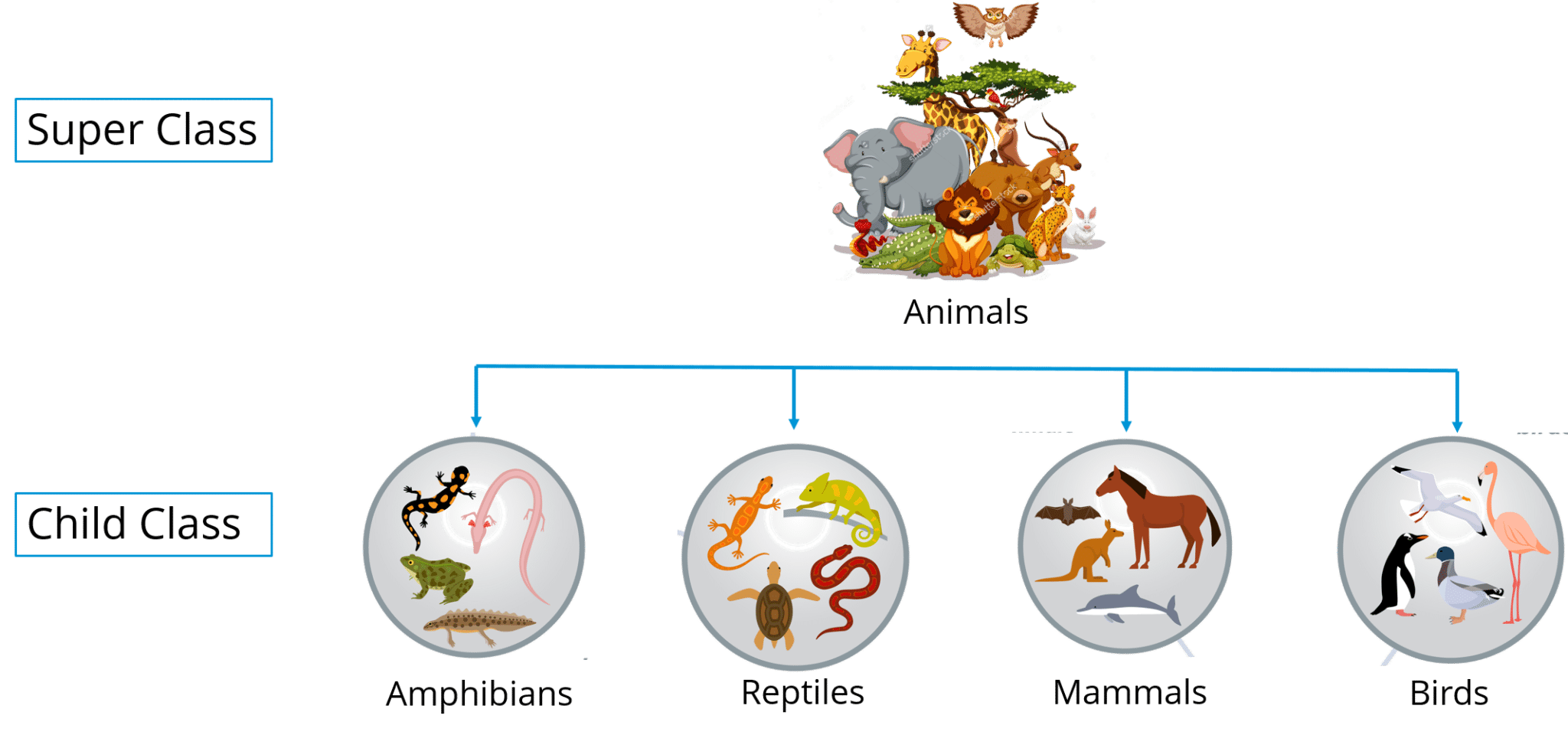
}

public static void main(String args[]){

      Student obj=new Student();    // Created an object

      obj.display();               // Method called

}



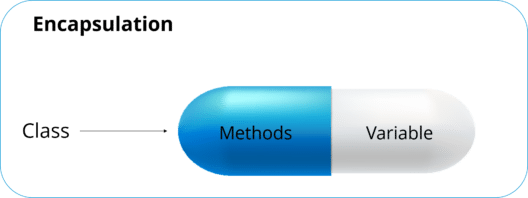
**Inheritance** can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order.

The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

Rules for Method Overriding

* The argument list should be exactly the same as that of the overridden method.
* The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
* The access level cannot be more restrictive than the overridden method's access level. For example: If the superclass method is declared public then the overridding method in the sub class cannot be either private or protected.
* Instance methods can be overridden only if they are inherited by the subclass.
* A method declared final cannot be overridden.
* A method declared static cannot be overridden but can be re-declared.
* If a method cannot be inherited, then it cannot be overridden.
* A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
* A subclass in a different package can only override the non-final methods declared public or protected.
* An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However, the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
* Constructors cannot be overridden.

**Encapsulation:** Encapsulation in Java is a mechanism of wrapping up the data and code together as a single unit. Refer to the below image where all your methods, variables are binded together in a single class.



In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class.it is also known as **data hiding**.

To achieve encapsulation in Java −

* Declare the variables of a class as private.
* Provide public setter and getter methods to modify and view the variables values.

The public setXXX() and getXXX() methods are the access points of the instance variables of the EncapTest class. Normally, these methods are referred as getters and setters. Therefore, any class that wants to access the variables should access them through these getters and setters.

Rules for Method Overriding

* The argument list should be exactly the same as that of the overridden method.
* The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
* The access level cannot be more restrictive than the overridden method's access level. For example: If the superclass method is declared public then the overridding method in the sub class cannot be either private or protected.
* Instance methods can be overridden only if they are inherited by the subclass.
* A method declared final cannot be overridden.
* A method declared static cannot be overridden but can be re-declared.
* If a method cannot be inherited, then it cannot be overridden.
* A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
* A subclass in a different package can only override the non-final methods declared public or protected.
* An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However, the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
* Constructors cannot be overridden.

**Polymorphism** is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

**Abstraction** is the quality of dealing with ideas rather than events.

Abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user.

Abstraction is achieved using Abstract classes and interfaces.

## **Abstract Class**

A class which contains the **abstract** keyword in its declaration is known as abstract class.

* Abstract classes may or may not contain *abstract methods*, i.e., methods without body ( public void get(); )
* But, if a class has at least one abstract method, then the class **must** be declared abstract.
* If a class is declared abstract, it cannot be instantiated.
* To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.
* If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

**interface** is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

An interface is similar to a class in the following ways −

* An interface can contain any number of methods.
* An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
* The byte code of an interface appears in a **.class** file.
* Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including −

* You cannot instantiate an interface.
* An interface does not contain any constructors.
* All of the methods in an interface are abstract.
* An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
* An interface is not extended by a class; it is implemented by a class.
* An interface can extend multiple interfaces.

**Exception** (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

* A user has entered an invalid data.
* A file that needs to be opened cannot be found.
* A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Based on these, we have three categories of Exceptions. You need to understand them to know how exception handling works in Java.

* **Checked exceptions** − A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called as compile time exceptions. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.

For example, if you use **FileReader** class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

**Unchecked exceptions** − An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

Multiple Catch Blocks

try {

// Protected code

} catch (ExceptionType1 e1) {

// Catch block

} catch (ExceptionType2 e2) {

// Catch block

} catch (ExceptionType3 e3) {

// Catch block

}

The Finally Block

try {

// Protected code

} catch (ExceptionType1 e1) {

// Catch block

} catch (ExceptionType2 e2) {

// Catch block

} catch (ExceptionType3 e3) {

// Catch block

}finally {

// The finally block always executes.

}

Collections Framework

A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following −

* **Interfaces** − These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
* **Implementations, i.e., Classes** − These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
* **Algorithms** − These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.

In addition to collections, the framework defines several map interfaces and classes. Maps store key/value pairs. Although maps are not *collections* in the proper use of the term, but they are fully integrated with collections.

## **The Collection Interfaces**

The collections framework defines several interfaces. This section provides an overview of each interface −

|  |  |
| --- | --- |
| **Sr.No.** | **Interface & Description** |
| 1 | The Collection Interface  This enables you to work with groups of objects; it is at the top of the collections hierarchy. |
| 2 | The List Interface  This extends **Collection** and an instance of List stores an ordered collection of elements. |
| 3 | The Set  This extends Collection to handle sets, which must contain unique elements. |
| 4 | The SortedSet  This extends Set to handle sorted sets. |
| 5 | The Map  This maps unique keys to values. |
| 6 | The Map.Entry  This describes an element (a key/value pair) in a map. This is an inner class of Map. |
| 7 | The SortedMap  This extends Map so that the keys are maintained in an ascending order. |
| 8 | The Enumeration  This is legacy interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. This legacy interface has been superceded by Iterator. |

## **The Collection Classes**

Java provides a set of standard collection classes that implement Collection interfaces. Some of the classes provide full implementations that can be used as-is and others are abstract class, providing skeletal implementations that are used as starting points for creating concrete collections.

The standard collection classes are summarized in the following table −

|  |  |
| --- | --- |
| **Sr.No.** | **Class & Description** |
| 1 | **AbstractCollection**  Implements most of the Collection interface. |
| 2 | **AbstractList**  Extends AbstractCollection and implements most of the List interface. |
| 3 | **AbstractSequentialList**  Extends AbstractList for use by a collection that uses sequential rather than random access of its elements. |
| 4 | LinkedList  Implements a linked list by extending AbstractSequentialList. |
| 5 | ArrayList  Implements a dynamic array by extending AbstractList. |
| 6 | **AbstractSet**  Extends Abstract Collection and implements most of the Set interface. |
| 7 | HashSet  Extends AbstractSet for use with a hash table. |
| 8 | LinkedHashSet  Extends HashSet to allow insertion-order iterations. |
| 9 | TreeSet  Implements a set stored in a tree. Extends AbstractSet. |
| 10 | **AbstractMap**  Implements most of the Map interface. |
| 11 | HashMap  Extends AbstractMap to use a hash table. |
| 12 | TreeMap  Extends AbstractMap to use a tree. |
| 13 | WeakHashMap  Extends AbstractMap to use a hash table with weak keys. |
| 14 | LinkedHashMap  Extends HashMap to allow insertion-order iterations. |
| 15 | IdentityHashMap  Extends AbstractMap and uses reference equality when comparing documents. |