MEERUT INSTITUTE OF ENGINEERING AND TECHNOLOGY MEERUT

Teachers' Manual

Object Oriented Programming with Java (BCS403)

B.Tech 2nd Year

CSE (AI&ML)

DR. A.P.J. ABDUL KALAM TECHNICAL UNIVERSITY LUCKNOW

Vision of Institute

To be an outstanding institution in the country imparting technical education, providing need-based, value-based and career-based programs and producing self-reliant, self-sufficient technocrats capable of meeting new challenges.

Mission of Institute

The mission of the institute is to educate young aspirants in various technical fields to fulfill global requirements of human resources by providing sustainable quality education, training and invigorating environment besides molding them into skilled competent and socially responsible citizens who will lead the building of a powerful nation.

Lecture Delivery Plan:

Lecture-1

1.1 Introduction: Why Java, History of Java

1.2 JVM, JRE, Java Environment

Lecture-2

2.1 Java Source File Structure

2.2 Compilation

2.3 Fundamental

Lecture-3

Programming Structures in Java:

3.1 Defining Classes in Java

3.2 Constructors

3.3 Methods

3.4 Access Specifiers

3.3 Static members

Lecture-4

4.1 Final Members

4.2 Comments

4.3 Data types

4.4 Variables

4.5 Operators

Lecture-5

5.1 Control Flow

5.2 Arrays and String

Lecture-6 Object Oriented Programming:

6.1 class, object

6.2 Inheritance Super Class, Sub Class

Lecture-7

7.1 Overriding, Overloading

7.2 Encapsulation, Polymorphism

Lecture-8

8.1 Abstraction

8.2 Interfaces

8.3 Abstract class

Lecture-9 Packages:

9.1 Defining Package

9.2 CLASSPATH Setting for Packages

Lecture-10

10.1 Making JAR Files for Library Packages

10.2 Import and Static Import Naming Convention For Packages

Lecture 1

Introduction:

Java is a class-based, object-oriented programming language that is designed to have as few implementation dependencies as possible. It is intended to let application developers write once, and run anywhere (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. Java was first released in 1995 and is widely used for developing applications for desktop, web, and mobile devices. Java is known for its simplicity, robustness, and security features, making it a popular choice for enterprise-level applications.

History of Java:

JAVA was developed by James Gosling at **Sun Microsystems** Inc in the year **1995** and later acquired by Oracle Corporation. It is a simple programming language. Java makes writing, compiling, and debugging programming easy. It helps to create reusable code and modular programs. Java is a class-based, object-oriented programming language and is designed to have as few implementation dependencies as possible. A general-purpose programming language made for developers to *write once run anywhere* that is compiled Java code can run on all platforms that support Java. Java applications are compiled to byte code that can run on any Java Virtual Machine. The syntax of Java is similar to c/c++.

History: Java's history is very interesting. It is a programming language created in 1991. James Gosling, Mike Sheridan, and Patrick Naughton, a team of Sun engineers known as the **Green team** initiated the Java language in 1991. **Sun Microsystems** released its first public implementation in 1996 as **Java 1.0**. It provides no-cost -run-times on popular platforms. Java1.0 compiler was re-written in Java by Arthur Van Hoff to strictly comply with its specifications. With the arrival of Java 2, new versions had multiple configurations built for different types of platforms.

In 1997, Sun Microsystems approached the ISO standards body and later formalized Java, but it soon withdrew from the process. At one time, Sun made most of its Java implementations available without charge, despite their proprietary software status. Sun generated revenue from Java through the selling of licenses for specialized products such as the Java Enterprise System.

On November 13, 2006, Sun released much of its Java virtual machine as free, open-source software. On May 8, 2007, Sun finished the process, making all of its JVM's core code available under open-source distribution terms.

The principles for creating java were simple, robust, secured, high-performance, portable, multi-threaded, interpreted, dynamic, etc. In 1995 Java was developed by **James Gosling**, who is known as the Father of Java. Currently, Java is used in mobile devices, internet programming, games, e-business, etc.

Implementation of a Java application program involves a following step. They include:

1. Creating the program
2. Compiling the program
3. Running the program

Remember that, before we begin creating the program, the Java Development Kit (JDK) must be properly installed on our system and also path will be set.

• **Compiling the program**
To compile the program, we must run the Java compiler (javac), with the name of the source file on "command prompt" like as follows:

If everything is OK, the "javac" compiler creates a file called "Test.class" containing byte code of the program.

• Running the program
We need to use the Java Interpreter to run a program.

Java programming language is named JAVA. Why?

After the name OAK, the team decided to give it a new name to it and the suggested words were Silk, Jolt, revolutionary, DNA, dynamic, etc. These all names were easy to spell and fun to say, but they all wanted the name to reflect the essence of technology. In accordance with James Gosling, Java the among the top names along with Silk, and since java was a unique name so most of them preferred it.

Java is the name of an **island** in Indonesia where the first coffee(named java coffee) was produced. And this name was chosen by James Gosling while having coffee near his office. Note that Java is just a name, not an acronym.

JVM, JRE, Java Environment:

Java Terminology

Before learning Java, one must be familiar with these common terms of Java.

1. Java Virtual Machine(JVM): This is generally referred to as JVM. There are three execution phases of a program. They are written, compile and run the program.

- Writing a program is done by a java programmer like you and me.
- The compilation is done by the **JAVAC** compiler which is a primary Java compiler included in the Java development kit (JDK). It takes the Java program as input and generates bytecode as output.
- In the Running phase of a program, **JVM** executes the bytecode generated by the compiler.

Now, we understood that the function of Java Virtual Machine is to execute the bytecode produced by the compiler. Every Operating System has a different JVM but the output they produce after the execution of bytecode is the same across all the operating systems. This is why Java is known as a **platform-independent language.**

2. Bytecode in the **Development Process:** As discussed, the Javac compiler of JDK compiles the java source code into bytecode so that it can be executed by JVM. It is saved as **.class** file by the compiler. To view the bytecode, a disassembler like javap can be used.

3. Java Development Kit(JDK): While we were using the term JDK when we learn about bytecode and JVM. So, as the name suggests, it is a complete Java development kit that includes everything including compiler, Java Runtime Environment (JRE), java debuggers, java docs, etc. For the program to execute in java, we need to install JDK on our computer in order to create, compile and run the java program.

4. Java Runtime Environment (JRE): JDK includes JRE. JRE installation on our computers allows the java program to run, however, we cannot compile it. JRE includes a browser, JVM, applet support, and plugins. For running the java program, a computer needs JRE.

5. Garbage Collector: In Java, programmers can't delete the objects. To delete or recollect that memory JVM has a program called Garbage Collector. Garbage Collectors can recollect the objects that are not referenced. So Java makes the life of a programmer easy by handling memory management. However, programmers should be careful about their code whether they are using objects that have been used for a long time. Because Garbage cannot recover the memory of objects being referenced.

6. ClassPath: The classpath is the file path where the java runtime and Java compiler look for **.class** files to load. By default, JDK provides many libraries. If you want to include external libraries they should be added to the classpath.

Primary/Main Features of Java

1. Platform Independent: Compiler converts source code to bytecode and then the JVM executes the bytecode generated by the compiler. This bytecode can run on any platform be it Windows, Linux, or macOS which means if we compile a program on Windows, then we can run it on Linux and vice versa. Each operating system has a different JVM, but the output produced by all the OS is the same after the execution of the bytecode. That is why we call java a platform-independent language.

2. Object-Oriented Programming Language: Organizing the program in the terms of a collection of objects is a way of object-oriented programming, each of which represents an instance of the class.

The four main concepts of Object-Oriented programming are:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

3. Simple: Java is one of the simple languages as it does not have complex features like pointers, operator overloading, multiple inheritances, and Explicit memory allocation.

4. Robust: Java language is robust which means reliable. It is developed in such a way that it puts a lot of effort into checking errors as early as possible, that is why the java compiler is able to detect even those errors that are not easy to detect by another programming language. The main features of java that make it robust are garbage collection, Exception Handling, and memory allocation.

5. Secure: In java, we don't have pointers, so we cannot access out-of-bound arrays i.e it shows **ArrayIndexOutOfBound Exception** if we try to do so. That's why several security flaws like stack corruption or buffer overflow are impossible to exploit in Java. Also, java programs run in an environment that is independent of the os(operating system) environment which makes java programs more secure.

6. Distributed: We can create distributed applications using the java programming language. Remote Method Invocation and Enterprise Java Beans are used for creating distributed applications in java. The java programs can be easily distributed on one or more systems that are connected to each other through an internet connection.

7. Multithreading: Java supports multithreading. It is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of the CPU.

8. Portable: As we know, java code written on one machine can be run on another machine. The platform-independent feature of java in which its platform-independent bytecode can be taken to any platform for execution makes java portable.

9. High Performance: Java architecture is defined in such a way that it reduces overhead during the runtime and at some times java uses Just In Time (JIT) compiler where the compiler compiles code on-demand basics where it only compiles those methods that are called making applications to execute faster.

10. Dynamic flexibility: Java being completely object-oriented gives us the flexibility to add classes, new methods to existing classes, and even create new classes through sub-classes. Java even supports functions written in other languages such as C, C++ which are referred to as native methods.

11. Sandbox Execution: Java programs run in a separate space that allows user to execute their applications without affecting the underlying system with help of a bytecode verifier. Bytecode verifier also provides additional security as its role is to check the code for any violation of access.

12. Write Once Run Anywhere: As discussed above java application generates a '.class' file that corresponds to our applications(program) but contains code in binary format. It provides ease t architecture-neutral ease as bytecode is not dependent on any machine architecture. It is the primary reason java is used in the enterprising IT industry globally worldwide.

13. Power of compilation and interpretation: Most languages are designed with the purpose of either they are compiled language or they are interpreted language. But java integrates arising enormous power as Java compiler compiles the source code to bytecode and JVM executes this bytecode to machine OS-dependent executable code.

class : class keyword is used to declare classes in Java

public : It is an access specifier. Public means this function is visible to all.

static : static is again a keyword used to make a function static. To execute a static function you do not have to create an Object of the class. The main() method here is called by JVM, without creating any object for class.

void : It is the return type, meaning this function will not return anything.

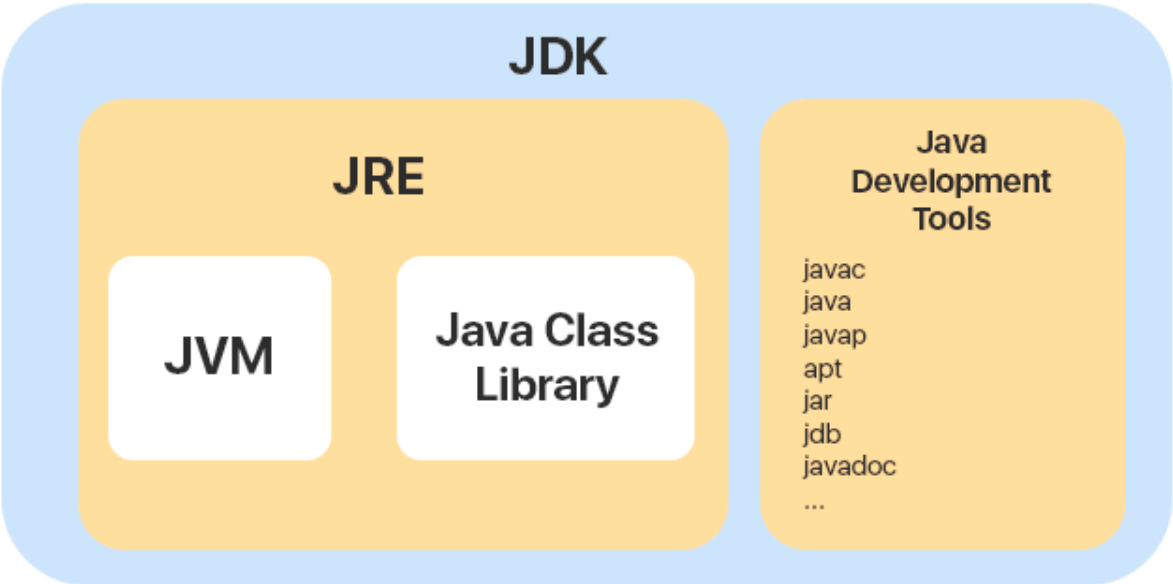
main : main() method is the most important method in a Java program. This is the method which is executed, hence all the logic must be inside the main() method. If a java class is not having a main() method, it causes compilation error.

String[] args : This is used to signify that the user may opt to enter parameters to the Java Program at command line. We can use both String[] args or String args[]. Java compiler would accept both forms.

System.out.println : This is used to print anything on the console like "printf" in C language.

Java Environment

- Java Environment includes large number of development tools.
- The development tools are part of the system known as Java Development Kit (JDK).



Java Virtual Machine(JVM): This is generally referred to as JVM. There are three execution phases of a program. They are written, compile and run the program.

- Writing a program is done by a java programmer like you and me.
- The compilation is done by the **JAVAC** compiler which is a primary Java compiler included in the Java development kit (JDK). It takes the Java program as input and generates bytecode as output.
- In the Running phase of a program, **JVM** executes the bytecode generated by the compiler.

Now, we understood that the function of Java Virtual Machine is to execute the bytecode produced by the compiler. Every Operating System has a different JVM but the output they produce after the execution of bytecode is the same across all the operating systems. This is why Java is known as a **platform-independent language.**

Bytecode in the **Development Process:** As discussed, the Javac compiler of JDK compiles the java source code into bytecode so that it can be executed by JVM. It is saved as **.class** file by the compiler. To view the bytecode, a disassembler like javap can be used.

Java Development Kit(JDK): While we were using the term JDK when we learn about bytecode and JVM. So, as the name suggests, it is a complete Java development kit that includes everything including compiler, Java Runtime Environment (JRE), java debuggers, java docs, etc. For the program to execute in java, we need to install JDK on our computer in order to create, compile and run the java program.

Java Runtime Environment (JRE): JDK includes JRE. JRE installation on our computers allows the java program to run, however, we cannot compile it. JRE includes a browser, JVM, applet support, and plugins. For running the java program, a computer needs JRE.

Lecture -2

JAVA SOURCE FILE STRUCTURE:

Java source file structure describes that the Java source code file must follow a schema or structure. In this article, we will see some of the important guidelines that a Java program must follow.

A Java program has the following structure:

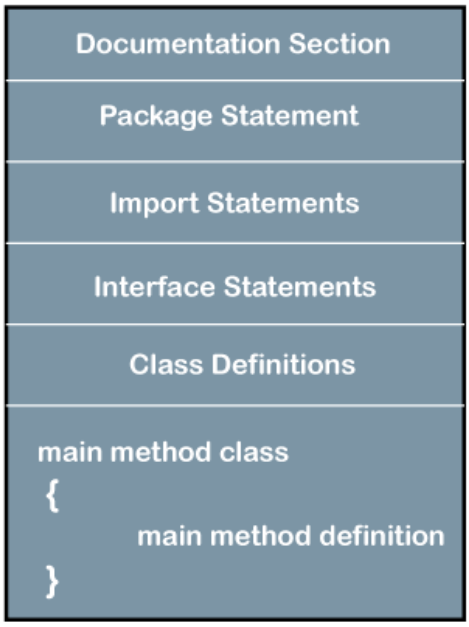
1. package statements: A package in Java is a mechanism to encapsulate a group of classes, sub-packages, and interfaces.

2. import statements: The import statement is used to import a package, class, or interface.

3. class definition: A class is a user-defined blueprint or prototype from which objects are created, and it is a passive entity.

```
package example;     //package
import java.util.*;   //import statement

class demo
{                    // class definition
  int x;
}
```



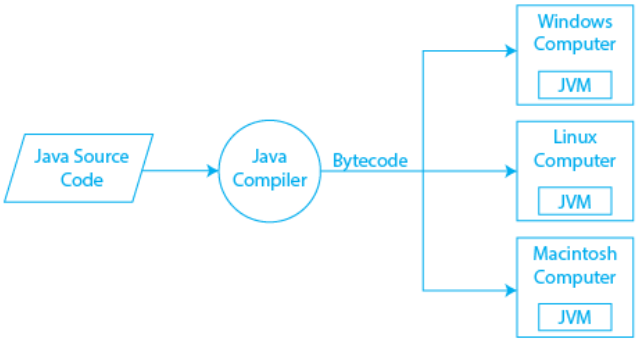
Structure of Java Program

Compilation

First, the source '.java' file is passed through the compiler, which then encodes the source code into a machine-independent encoding, known as Bytecode. The content of each class contained in the source file is stored in a separate '.class' file. While converting the source code into the bytecode, the compiler follows the following steps:

What is the Compilation Process in Java?

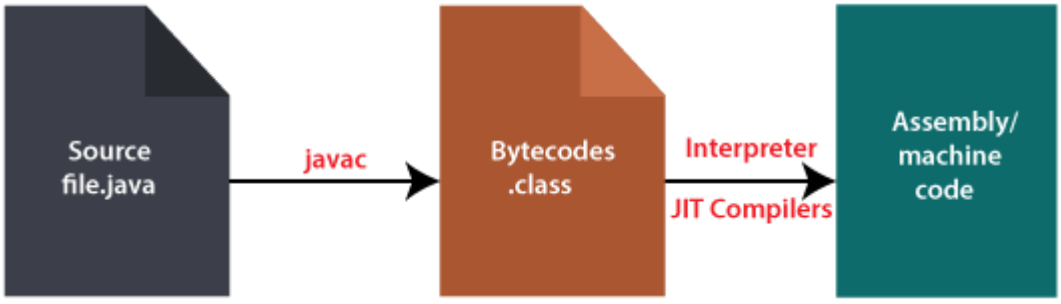
The source code of a Java code is compiled into an intermediate binary code known as the Bytecode during the Java compilation process. The machine cannot directly execute this Bytecode. A virtual machine known as the Java Virtual Machine, or JVM, understands it. JVM includes a Java interpreter that converts Bytecode to target computer machine code. JVM is platform-specific, which means that each platform has its own JVM. However, once the proper JVM is installed on the machine, any Java Bytecode code can be run. This is shown in the diagram below:



Java application development (implementation) and can be broken down into the following phases:

Compilation:
A special application called a compiler, executes our Java program on what is known as a virtual Java machine (JVM).

The compiler transforms source code into so-called JVM bytecode, or machine code read by JVM. In addition, the compiler should check the code for lexical and semantic issues and optimise it.



.java file

.java file contains the source code . We will write our code in *.java file* like Sum of two numbers ,Hello World, Java Application etc.

We need to convert source code to machine code so computer can understand.

Compiler — In C , C++ programming languages , the role of compiler is take the source file and convert it into the machine readable format for ex. 0's and 1's.

But in Java , compiler/loader compiles the entire .java file into byte code and the extension of that byte code is ***.class .***

.class file

Byte Code — Some intermediate language of Java .This code will not directly run on a system. We need JVM(Java Virtual Machine) to run this code.

Command to compile the code — ***javac filename.java***

Command to run the code — ***java filename***

Machine Code (set of instructions for the computer)

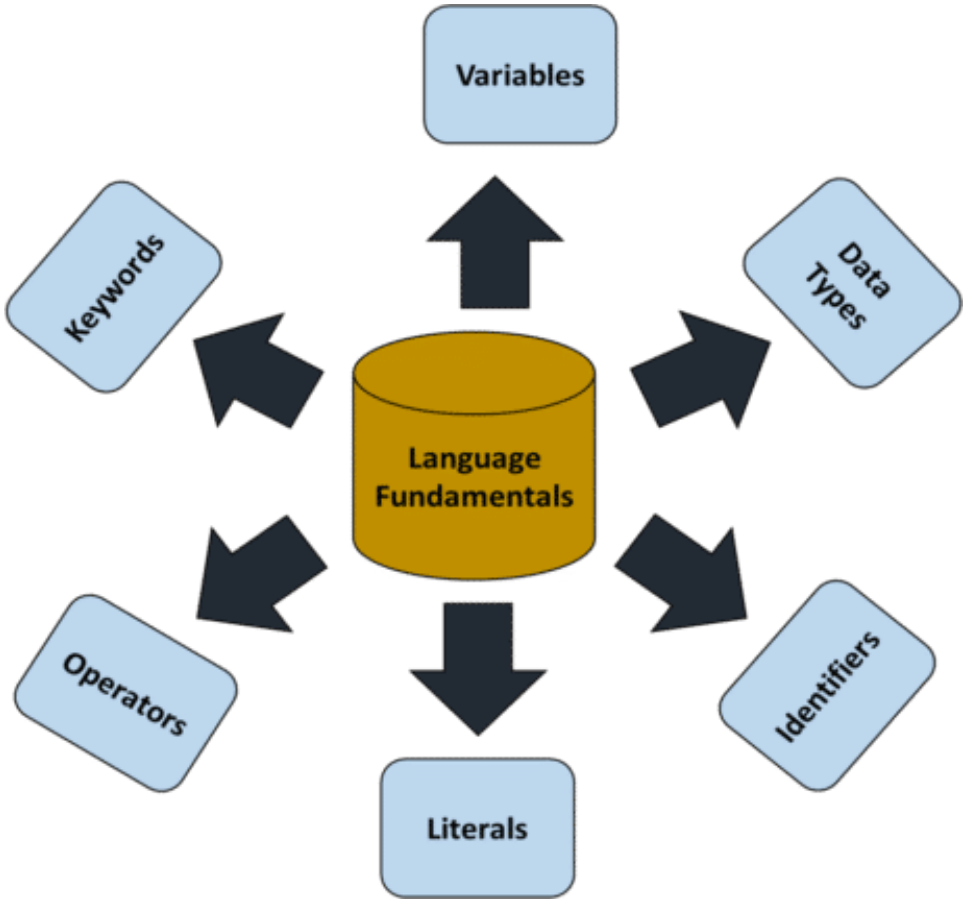
Interpreter — Runs the code line by line .

JVM converts byte code into machine code line by line .Here JVM works as Interpreter. .class file can be run on any operating system. This is the reason why Java is ***platform independent.***

More about platform independent — It means that byte code can run on all operating systems.

Fundamental of Java:

Language Fundamentals" topic which includes the basic concepts to write programs in Java.


Variables
Keywords
Data Types
Language Fundamentals
Operators
Identifiers
Literals

Lecture-3

Defining Classes in Java: Java provides a reserved keyword **class** to define a class. The keyword must be followed by the class name. Inside the class, we declare methods and variables.

In general, class declaration includes the following in the order as it appears:

1. **Modifiers:** A class can be public or has default access.
2. **class keyword:** The class keyword is used to create a class.
3. **Class name:** The name must begin with an initial letter (capitalized by convention).
4. **Superclass (if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
5. **Interfaces (if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
6. **Body:** The class body surrounded by braces, { }.

Syntax:

1. <access specifier> **class** class_name
2. {
3. // member variables
4. // class methods
5. }

Java Classes

A class in Java is a set of objects which shares common characteristics/ behavior and common properties/ attributes. It is a user-defined blueprint or prototype from which objects are created. For example, Student is a class while a particular student named Ravi is an object.

Properties of Java Classes

1. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
1. Class does not occupy memory.
1. Class is a group of variables of different data types and a group of methods.
1. A Class in Java can contain:
   - Data member
   - Method
   - Constructor
   - Nested Class
   - Interface

Class Declaration in Java

access_modifier **class** <*class_name*>

```
{
   data member;
   method;
   constructor;
   nested class;
   interface;
}
```

Example of Java Class

```
// Java Program for class example

 class Student {

   // data member (also instance variable)

   int id;

   // data member (also instance variable)

   String name;

    public static void main(String args[])

   {       // creating an object of Student

                          Student s1 = new Student();

      System.out.println(s1.id);

      System.out.println(s1.name);

   }

}
```

Constructor:

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.

What are Constructors in Java?

In Java, a Constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling the constructor, memory

for the object is allocated in the memory. It is a special type of method that is used to initialize the object. Every time an object is created using the new() keyword, at least one constructor is called.

```
// Create a Main class

public class Main {

  int x;  // Create a class attribute

  // Create a class constructor for the Main class

  public Main() {

    x = 5;  // Set the initial value for the class attribute x

  }

  public static void main(String[] args) {

    Main myObj = new Main(); // Create an object of class Main (This will call the constructor)

    System.out.println(myObj.x); // Print the value of x

  }

}
```

Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

How Java Constructors are Different From Java Methods?

- Constructors must have the same name as the class within which it is defined it is not necessary for the method in Java.
- Constructors do not return any type while method(s) have the return type or **void** if does not return any value.
- Constructors are called only once at the time of Object creation while method(s) can be called any number of times.

Note: *Default constructor provides the default values to the object like 0, null, etc. depending on the type.*

2. Parameterized Constructor in Java

A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with our own values, then use a parameterized constructor.

JAVA METHOD

The **method in Java** or Methods of Java is a collection of statements that perform some specific task and return the result to the caller. A Java method can perform some specific task without returning anything. Java Methods allow us to **reuse** the code without retyping the code. In Java, every method must be part of some class that is different from languages like C, C++, and Python.
1. A method is like a function i.e. used to expose the behavior of an object.
2. It is a set of codes that perform a particular task.

Syntax of Method

```
<access_modifier> <return_type> <method_name>( list_of_parameters)
{
   //body
}
```

Advantage of Method

- Code Reusability
- Code Optimization

Note: *Methods are time savers and help us to reuse the code without retyping the code.*

Method Declaration

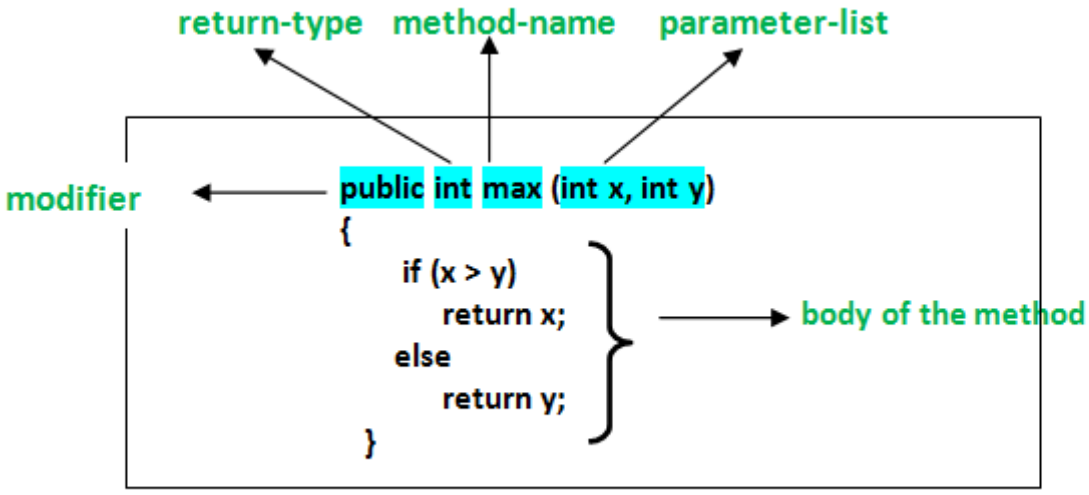
In general, method declarations have 6 components:

1. Modifier: It defines the **access type** of the method i.e. from where it can be accessed in your application. In Java, there 4 types of access specifiers.

- **public:** It is accessible in all classes in your application.
- **protected:** It is accessible within the class in which it is defined and in its subclass/es
- **private:** It is accessible only within the class in which it is defined.
- **default:** It is declared/defined without using any modifier. It is accessible within the same class and package within which its class is defined.

Note: *It is* ***Optional*** *in syntax.*
2. The return type: The data type of the value returned by the method or void if does not return a value. It is **Mandatory** in syntax.
3. Method Name: the rules for field names apply to method names as well, but the convention is a little different. It is **Mandatory** in syntax.
4. Parameter list: Comma-separated list of the input parameters is defined, preceded by their data type, within the enclosed parenthesis. If there are no parameters, you must use empty parentheses (). It is **Optional** in syntax.
5. Exception list: The exceptions you expect by the method can throw, you can specify these exception(s). It is **Optional** in syntax.

6. Method body: it is enclosed between braces. The code you need to be executed to perform your intended operations. It is **Optional** in syntax.



Types of Methods in Java

There are two types of methods in Java:

1. Predefined Method

In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the standard library method or built-in method. We can directly use these methods just by calling them in the program at any point.

2. User-defined Method

The method written by the user or programmer is known as a user-defined method. These methods are modified according to the requirement.

Access Specifier (in Java ACCESS MODIFIER)

1. Default Access Modifier

When no access modifier is specified for a class, method, or data member – It is said to be having the **default** access modifier by default. The data members, classes, or methods that are not declared using any access modifiers i.e. having default access modifiers are accessible **only within the same package**

2. Private Access Modifier

The private access modifier is specified using the keyword **private**. The methods or data members declared as private are accessible only **within the class** in which they are declared.

- Any other **class of** the **same package will not be able to access** these members.
- Top-level classes or interfaces can not be declared as private because
    - private means “only visible within the enclosing class”.
    - protected means “only visible within the enclosing class and any subclasses”

Hence these modifiers in terms of application to classes, apply only to nested classes and not on top-level classes

In this example, we will create two classes A and B within the same package p1. We will declare a method in class A as private and try to access this method from class B and see the result.

3. Protected Access Modifier

The protected access modifier is specified using the keyword **protected**.
The methods or data members declared as protected are **accessible within the same package or subclasses in different packages.**
In this example, we will create two packages p1 and p2. Class A in p1 is made public, to access it in p2. The method display in class A is protected and class B is inherited from class A and this protected method is then accessed by creating an object of class B.

2. Public Access modifier

The public access modifier is specified using the keyword **public**.

- The public access modifier has the **widest scope** among all other access modifiers.
- Classes, methods, or data members that are declared as public are **accessible from everywhere** in the program. There is no restriction on the scope of public data members.

Static members:

The **static keyword** in Java is mainly used for memory management. The static keyword in Java is used to share the same variable or method of a given class. The users can apply static keywords with variables, methods, blocks, and nested classes. The static keyword belongs to the class than an instance of the class. The static keyword is used for a constant variable or a method that is the same for every instance of a class.
The *static* keyword is a non-access modifier in Java that is applicable for the following:

1. Blocks
2. Variables
3. Methods
4. Classes

Example:

```
public class MyClass {
  public static void sample(){
    System.out.println("Hello");
```

```
  }
  public static void main(String args[]){
    MyClass.sample();
  }
}
```

```
public class MyClass {
  public static int data = 20;
  public static void main(String args[]){
    System.out.println(MyClass.data);
  }
  Java Arrays with Answers
  20
}
```

```
public class MyClass {
  static{
    System.out.println("Hello this is a static block");
  }
  public static void main(String args[]){
    System.out.println("This is main method");
  }
}
```

Characteristics of static keyword:

- **Shared memory allocation**: Static variables and methods are allocated memory space only once during the execution of the program. This memory space is shared among all instances of the class, which makes static members useful for maintaining global state or shared functionality.
- **Accessible without object instantiation:** Static members can be accessed without the need to create an instance of the class. This makes them useful for providing utility functions and constants that can be used across the entire program.
- **Associated with class, not objects:** Static members are associated with the class, not with individual objects. This means that changes to a static member are reflected in all instances of the class, and that you can access static members using the class name rather than an object reference.
- **Cannot access non-static members:** Static methods and variables cannot access non-static members of a class, as they are not associated with any particular instance of the class.
- **Can be overloaded, but not overridden**: Static methods can be overloaded, which means that you can define multiple methods with the same name but different parameters. However, they cannot be overridden, as they are associated with the class rather than with a particular instance of the class.

When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. For example, in the below java program, we are accessing static method *m1()* without creating any object of the *Test* class.

Lecture-4:

Final Members:

final keyword is used in different contexts. First of all, *final* is a non-access modifier applicable only to a variable, a method, or a class. The following are different contexts where final is used.

Characteristics of final keyword in java:

In Java, the final keyword is used to indicate that a variable, method, or class cannot be modified or extended. Here are some of its characteristics:

- **Final variables:** When a variable is declared as final, its value cannot be changed once it has been initialized. This is useful for declaring constants or other values that should not be modified.
- **Final methods**: When a method is declared as final, it cannot be overridden by a subclass. This is useful for methods that are part of a class's public API and should not be modified by subclasses.
- **Final classes:** When a class is declared as final, it cannot be extended by a subclass. This is useful for classes that are intended to be used as is and should not be modified or extended.
- **Initialization:** Final variables must be initialized either at the time of declaration or in the constructor of the class. This ensures that the value of the variable is set and cannot be changed.
- **Performance:** The use of final can sometimes improve performance, as the compiler can optimize the code more effectively when it knows that a variable or method cannot be changed.
- **Security:** final can help improve security by preventing malicious code from modifying sensitive data or behavior.

Overall, the final keyword is a useful tool for improving code quality and ensuring that certain aspects of a program cannot be modified or extended. By declaring variables, methods, and classes as final, developers can write more secure, robust, and maintainable code.

Java Comments

The Java comments are the statements in a program that are not executed by the compiler and interpreter.

- Comments are used to make the program more readable by adding the details of the code.
- It makes easy to maintain the code and to find the errors easily.
- The comments can be used to provide information or explanation about the variable, method, class, or any statement.
- It can also be used to prevent the execution of program code while testing the alternative code.

Types of Java Comments

There are three types of comments in Java.

1. Single Line Comment
2. Multi Line Comment
3. Documentation Comment

1) Java Single Line Comment

The single-line comment is used to comment only one line of the code. It is the widely used and easiest way of commenting the statements.

Single line comments starts with two forward slashes **(//)**. Any text in front of // is not executed by Java.

Syntax:

```
1. //This is single line comment
```

2) Java Multi Line Comment

The multi-line comment is used to comment multiple lines of code. It can be used to explain a complex code snippet or to comment multiple lines of code at a time (as it will be difficult to use single-line comments there).

Multi-line comments are placed between /* and */. Any text between /* and */ is not executed by Java.

Syntax:

```
1. /*
2. This
3. is
4. multi line
5. comment
6. */
```

3) Java Documentation Comment

Documentation comments are usually used to write large programs for a project or software application as it helps to create documentation API. These APIs are needed for reference, i.e., which classes, methods, arguments, etc., are used in the code.

To create documentation API, we need to use the **javadoc tool**. The documentation comments are placed between /** and */.

Syntax:

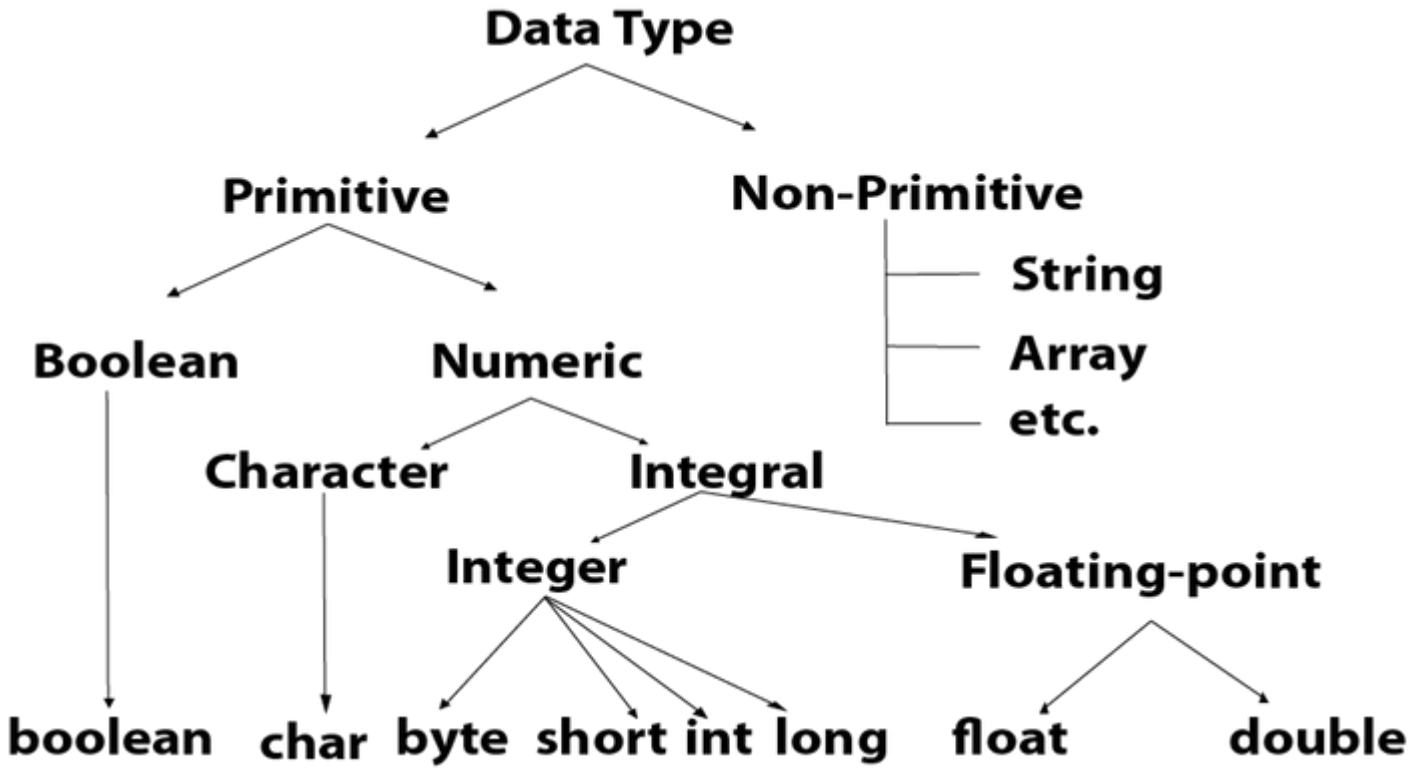
```
1. /**
2. *
3. *We can use various tags to depict the parameter
```

4. *or heading or author name
5. *We can also use HTML tags
6. *
7. */

Data Types in Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.



Data Type	Default Value	Default size
boolean	false	1 bit

char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

```
public class JavaTester {
   public static void main(String args[]) {

      byte byteValue1 = 2;
      byte byteValue2 = 4;
      byte byteResult = (byte)(byteValue1 + byteValue2);

      System.out.println("Byte: " + byteResult);

      short shortValue1 = 2;
      short shortValue2 = 4;
      short shortResult = (short)(shortValue1 + shortValue2);

      System.out.println("Short: " + shortResult);

      int intValue1 = 2;
      int intValue2 = 4;
      int intResult = intValue1 + intValue2;

      System.out.println("Int: " + intResult);

      long longValue1 = 2L;
      long longValue2 = 4L;
      long longResult = longValue1 + longValue2;

      System.out.println("Long: " + longResult);

      float floatValue1 = 2.0f;
      float floatValue2 = 4.0f;
      float floatResult = floatValue1 + floatValue2;

      System.out.println("Float: " + floatResult);

      double doubleValue1 = 2.0;
      double doubleValue2 = 4.0;
```

```
    double doubleResult = doubleValue1 + doubleValue2;

    System.out.println("Double: " + doubleResult);

    boolean booleanValue = true;

    System.out.println("Boolean: " + booleanValue);

    char charValue = 'A';

    System.out.println("Char: " + charValue);
  }

}
```

Data types are divided into two groups:

- Primitive data types -
  includes byte, short, int, long, float, double, boolean and char
- Non-primitive data types - such as String, Arrays and Classes (you will learn more about these in a later chapter

Type	Description	Default	Size	Example Literals	Range of values
boolean	true or false	false	1 bit	true, false	true, false
byte	twos-complement integer	0	8 bits	(none)	-128 to 127
char	Unicode character	\u0000	16 bits	‘a’, ‘\u0041’, ‘\101’, ‘\\’, ‘\’, ‘\n’, ‘β’	characters representation of ASCII values 0 to 255
short	twos-complement integer	0	16 bits	(none)	-32,768 to 32,767
int	twos-complement intger	0	32 bits	-2,-1,0,1,2	-2,147,483,648 to 2,147,483,647
long	twos-compleme	0	64 bits	-2L,-1L,0L,1L,2L	-9,223,372,036,854,775,80

	nt integer				8 to 9,223,372,036,854,775,807
float	IEEE 754 floating point	0.0	32 bits	1.23e100f , -1.23e-100f , .3f ,3.14F	upto 7 decimal digits
double	IEEE 754 floating point	0.0	64 bits	1.23456e300d , -123456e-300d , 1e1d	upto 16 decimal digits

Variables in Java

Java variable is a name given to a memory location. It is the basic unit of storage in a program.

- The value stored in a variable can be changed during program execution.
- Variables in Java are only a name given to a memory location. All the operations done on the variable affect that memory location.
- In Java, all variables must be declared before use.

How to Declare Variables in Java?

We can declare variables in Java as pictorially depicted below as a visual aid.

1. **datatype**: Type of data that can be stored in this variable.
1. **data_name:** Name was given to the variable.

In this way, a name can only be given to a memory location. It can be assigned values in two ways:

- Variable Initialization
- Assigning value by taking input

How to Initialize Variables in Java?

It can be perceived with the help of 3 components that are as follows:

- **datatype**: Type of data that can be stored in this variable.
- **variable_name**: Name given to the variable.
- **value**: It is the initial value stored in the variable.

Types of Variables

There are three types of variables in Java:

- local variable
- instance variable
- static variable

1) Local Variable

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

A local variable cannot be defined with "static" keyword.

2) Instance Variable

A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as static.

It is called an instance variable because its value is instance-specific and is not shared among instances.

3) Static variable

A variable that is declared as static is called a static variable. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

Example to understand the types of variables in java

```
1. public class A
2. {
3.    static int m=100;//static variable
4.    void method()
5.    {
6.       int n=90;//local variable
7.    }
8.    public static void main(String args[])
9.    {
10.      int data=50;//instance variable
11.   }
12.}//end of class
```

Java divides the operators into the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators

ARITHMETIC OPERATOR

Operator	Name	Description	Example

--	Decrement	Decreases the value of a variable by 1	--x

JAVA ASSIGNMENT OPERATORS

Assignment operators are used to assign values to variables.

In the example below, we use the **assignment** operator (=) to assign the value **10** to a variable called **x**:

```
int x = 10;
```

JAVA COMPARISON OPERATORS

Comparison operators are used to compare two values (or variables). This is important in programming, because it helps us to find answers and make decisions.

JAVA LOGICAL OPERATORS

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4

!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

Java Operator Precedence

Operator Type	Category	Precedence
Unary	postfix	*expr*++ *expr*--
	prefix	++*expr* --*expr* +*expr* -*expr* ~ !
Arithmetic	multiplicative	* / %
	additive	+ -
Shift	shift	<< >> >>>
Relational	comparison	< > <= >= instanceof
	equality	== !=
Bitwise	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	|
Logical	logical AND	&&
	logical OR	||
Ternary	ternary	? :
Assignment	assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Lecture -5:

JAVA CONTROL STATEMENTS | CONTROL FLOW IN JAVA

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.

1. Decision Making statements
    - if statements
    - switch statement
2. Loop statements
    - do while loop
    - while loop
    - for loop
    - for-each loop
3. Jump statements
    - break statement
    - continue statement

1) Simple if statement:

It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax of if statement is given below.

```
1. if(condition) {
2. statement 1; //executes when condition is true
3. }
```

2) if-else statement

The if-else statement is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

Syntax:

```
1. if(condition) {
2. statement 1; //executes when condition is true
3. }
4. else{
5. statement 2; //executes when condition is false
6. }
```

3) if-else-if ladder:

Syntax of if-else-if statement is given below.

```
1. if(condition 1) {
```

```
2. statement 1; //executes when condition 1 is true
3. }
4. else if(condition 2) {
5. statement 2; //executes when condition 2 is true
6. }
7. else {
8. statement 2; //executes when all the conditions are false
9. }
```

4. Nested if-statement

```
1. if(condition 1) {
2. statement 1; //executes when condition 1 is true
3. if(condition 2) {
4. statement 2; //executes when condition 2 is true
5. }
6. else{
7. statement 2; //executes when condition 2 is false
8. }
9. }
```

Ex: //A Java Program to demonstrate the use of if-else statement.

```
1. //It is a program of odd and even number.
2. public class IfElseExample {
3. public static void main(String[] args) {
4.     //defining a variable
5.     int number=13;
6.     //Check if the number is divisible by 2 or not
7.     if(number%2==0){
8.         System.out.println("even number");
9.     }else{
10.        System.out.println("odd number");
11.    }
12.}
13.}
```

Ex: Example

```
int time = 22;

if (time < 10) {

  System.out.println("Good morning.");
```

```
} else if (time < 18) {
  System.out.println("Good day.");
} else {
  System.out.println("Good evening.");
}
// Outputs "Good evening."
```

Switch Statement:

In Java, Switch statements are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

The syntax to use the switch statement is given below.

```
1. switch (expression){
2.     case value1:
3.      statement1;
4.      break;
5.     .
6.     .
7.     .
8.     case valueN:
9.      statementN;
10.     break;
11.    default:
12.     default statement;
13.}
```

Ex:

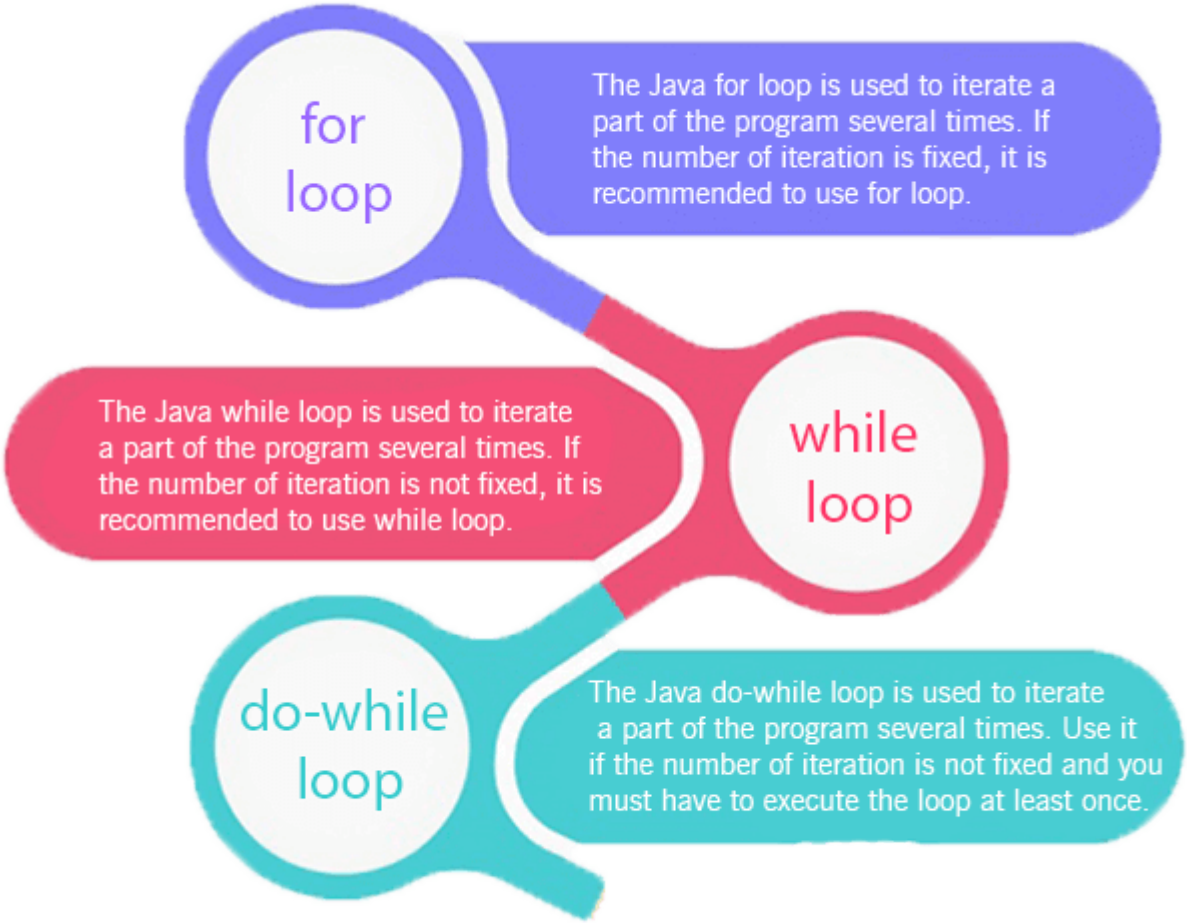
```
14.public class SwitchExample {
15.public static void main(String[] args) {
16.    //Declaring a variable for switch expression
17.    int number=20;
18.    //Switch expression
19.    switch(number){
20.    //Case statements
```

```
21.    case 10: System.out.println("10");
22.    break;
23.    case 20: System.out.println("20");
24.    break;
25.    case 30: System.out.println("30");
26.    break;
27.    //Default case statement
28.    default:System.out.println("Not in 10, 20 or 30");
29.    }
30.}
31.}
```

Loops in Java

The Java *for loop* is used to iterate a part of the program several times. If the number of iteration is **fixed**, it is recommended to use for loop.

There are three types of for loops in Java.



ForExample.java

```
1.  //Java Program to demonstrate the example of for loop
```

```
2. //which prints table of 1
3. public class ForExample {
4. public static void main(String[] args) {
5.     //Code of Java for loop
6.     for(int i=1;i<=10;i++){
7.         System.out.println(i);
8.     }
9. }
10.}
```

WhileExample.java

```
1. public class WhileExample {
2. public static void main(String[] args) {
3.     int i=1;
4.     while(i<=10){
5.         System.out.println(i);
6.     i++;
7.     }
8. }
9. }
```

DoWhileExample.java

```
1. public class DoWhileExample {
2. public static void main(String[] args) {
3.     int i=1;
4.     do{
5.         System.out.println(i);
6.     i++;
7.     }while(i<=10);
8. }
9. }
```

Java Arrays

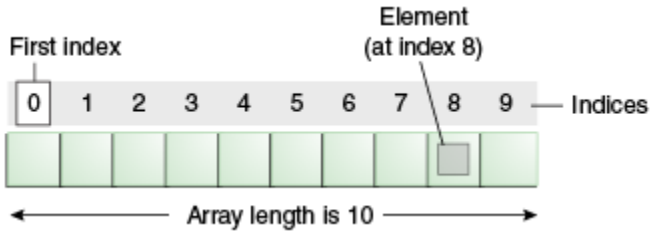
Normally, an array is a collection of similar type of elements which has contiguous memory location.

Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator. In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimentional or multidimentional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.



Advantages

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

Disadvantages

- **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in Java

Syntax to Declare an Array in Java

1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];

Instantiation of an Array in Java

1. arrayRefVar=**new** datatype[size];

Example of Java Array

1. Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array. //Java Program to illustrate how to declare, instantiate, initialize
2. //and traverse the Java array.
3. **class** Testarray{
4. **public static void** main(String args[]){
5. **int** a[]=**new int**[5];//declaration and instantiation
6. a[0]=10;//initialization
7. a[1]=20;
8. a[2]=70;
9. a[3]=40;
10. a[4]=50;
11. //traversing array
12. **for**(**int** i=0;i<a.length;i++)//length is the property of array
13. System.out.println(a[i]);
14. }}

Multidimensional Array in Java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in Java

1. dataType[][] arrayRefVar; (or)
2. dataType [][]arrayRefVar; (or)
3. dataType arrayRefVar[][]; (or)
4. dataType []arrayRefVar[];

Example to instantiate Multidimensional Array in Java

1. **int**[][] arr=**new int**[3][3];//3 row and 3 column

Example of Multidimensional Java Array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

1. //Java Program to illustrate the use of multidimensional array
2. **class** Testarray3{
3. **public static void** main(String args[]){
4. //declaring and initializing 2D array

```
5. int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
6. //printing 2D array
7. for(int i=0;i<3;i++){
8.  for(int j=0;j<3;j++){
9.    System.out.print(arr[i][j]+" ");
10. }
11. System.out.println();
12.}
13.}}
```

Java String

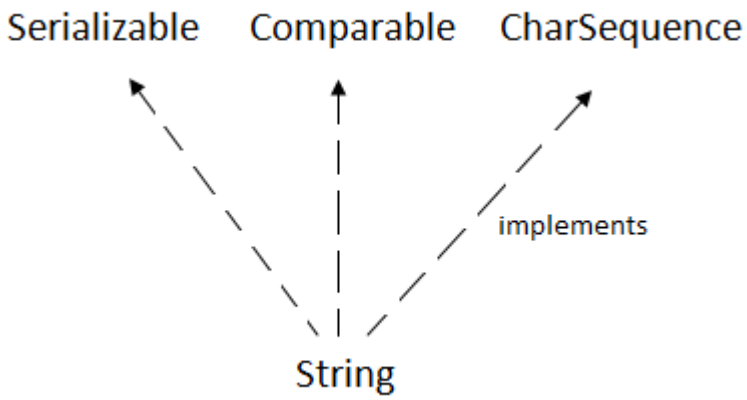
In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string. For example:

```
1. char[] ch={'j','a','v','a','t','p','o','i','n','t'};
2. String s=new String(ch);
```

is same as:

```
1. String s="javatpoint";
```

2. **Java String** class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.
3. The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces.



What is String in Java?

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

How to create a string object?

There are two ways to create String object:

1. By string literal

2. By new keyword

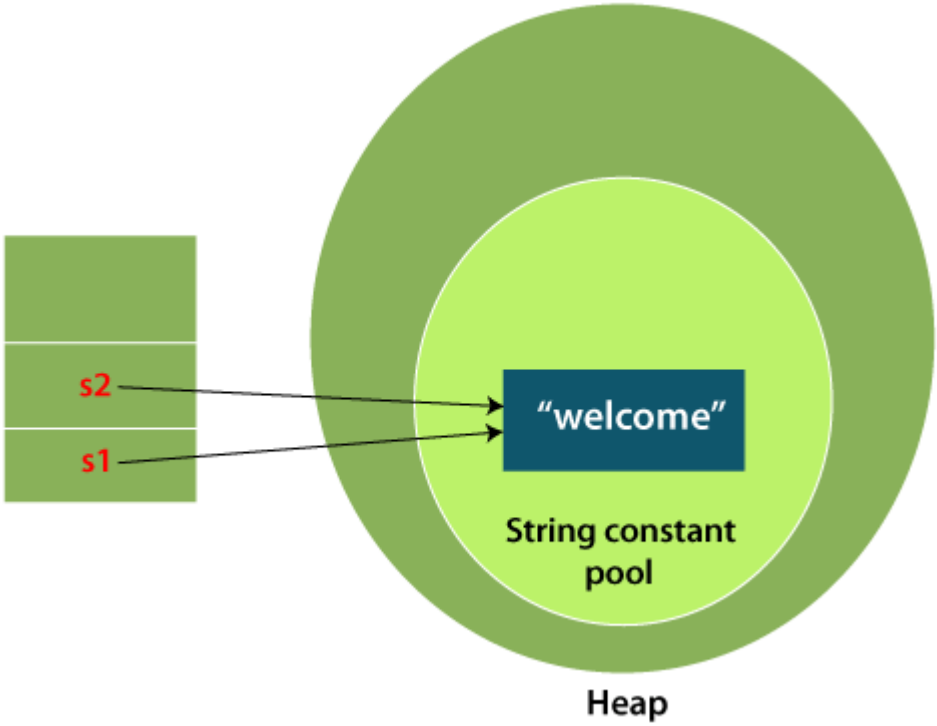
1) String Literal

Java String literal is created by using double quotes. For Example:

1. String s="welcome";

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

1. String s1="Welcome";
2. String s2="Welcome";//It doesn't create a new instance



Why Java uses the concept of String literal?

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

2) By new keyword

1.

   String s=**new** String("Welcome");//creates two objects and one reference vari able

In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

Java String Example

StringExample.java

```
1. public class StringExample{
2. public static void main(String args[]){
3. String s1="java";//creating string by Java string literal
4. char ch[]={'s','t','r','i','n','g','s'};
5. String s2=new String(ch);//converting char array to string
6. String s3=new String("example");//creating Java string by new keyword
7. System.out.println(s1);
8. System.out.println(s2);
9. System.out.println(s3);
10.}}
```

Lecture -6:

Object Oriented Programming:

UNDERSTANDING CLASSES AND OBJECTS IN JAVA

The term *Object-Oriented* explains the concept of organizing the software as a combination of different types of objects that incorporates both data and behavior. Hence, Object-oriented programming(OOPs) is a programming model, that simplifies software development and maintenance by providing some rules. Programs are organised around objects rather than action and logic. It increases the flexibility and maintainability of the program. Understanding the working of the program becomes easier, as OOPs brings data and its behavior(methods) into a single(objects) location.

Classes: A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. Classes are required in OOPs because:

- It provides template for creating objects, which can bind code into data.
- It has definitions of methods and data.
- It supports inheritance property of Object Oriented Programming and hence can maintain class hierarchy.
- It helps in maintaining the access specifications of member variables.

Objects: It is the basic unit of Object Oriented Programming and it represents the real life entities.

Real-life entities share two characteristics: they all have attributes and behaviour.

An object consists of:

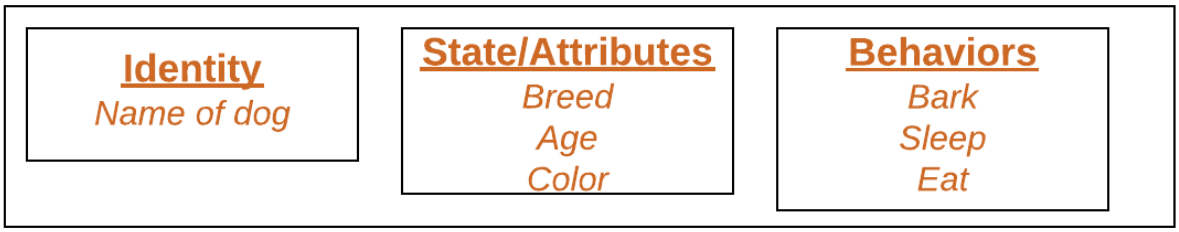
- **State:** It is represented by *attributes* of an object. It also shows properties of an object.

- **Behaviour:** It is represented by *methods* of an object. It shows response of an object with other objects.
- **Identity:** It gives a unique name to an object. It also grants permission to one object to interact with other objects.

Objects are required in OOPs because they can be created to call a non-static function which are not present inside the Main Method but present inside the Class and also provide the name to the space which is being used to store the data.

Example: For addition of two numbers, it is required to store the two numbers separately from each other, so that they can be picked and the desired operations can be performed on them. Hence creating two different objects to store the two numbers will be an ideal solution for this scenario.

Example to demonstrate the use of Objects and classes in OOPs

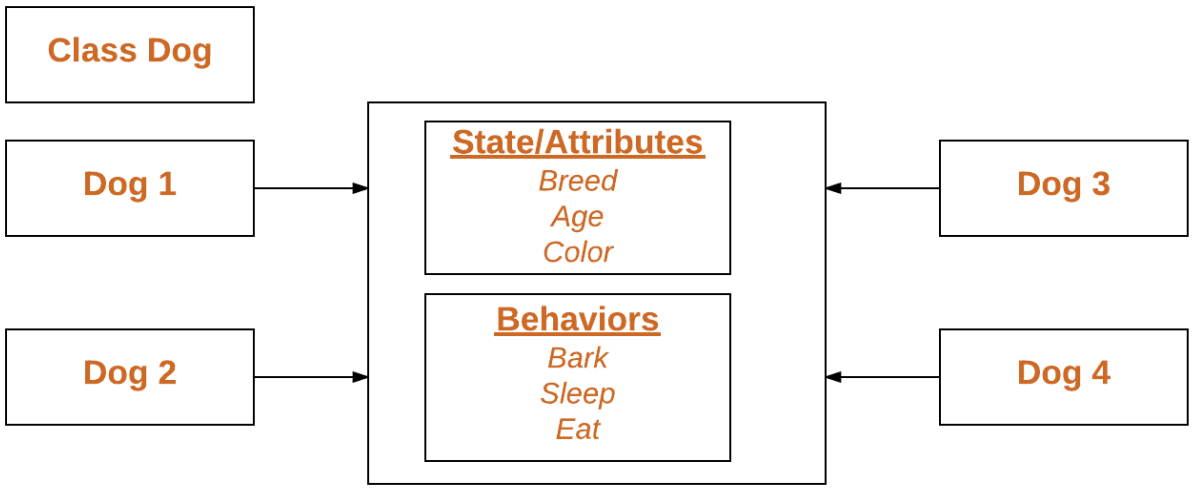


OBJECTS

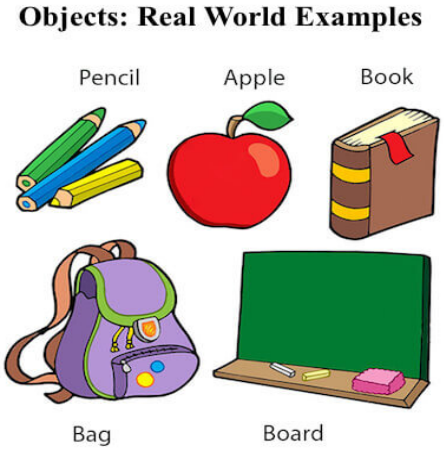
Objects relate to things found in the real world. For example, a graphics program may have objects such as “circle”, “square”, “menu”. An online shopping system might have objects such as “shopping cart”, “customer”, and “product”.

Declaring Objects (Also called instantiating a class)

When an object of a class is created, the class is said to be **instantiated**. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.



What is an object in Java



An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

- **State:** represents the data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

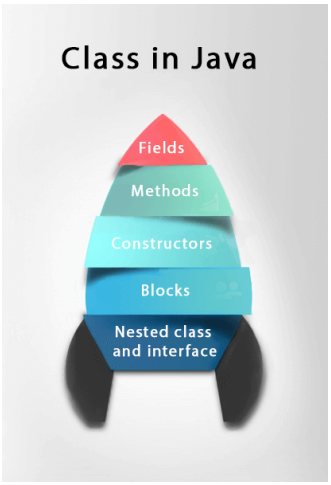
Object Definitions:

- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.

What is a class in Java

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

- **Fields**
- **Methods**
- **Constructors**
- **Blocks**
- 

Nested class and interface

Syntax to declare a class:

```
1. class <class_name>{
2.     field;
3.     method;
4. }
```

Instance variable in Java

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

Method in Java
In Java, a method is like a function which is used to expose the behavior of an object.

Advantage of Method

- Code Reusability
- Code Optimization

NEW KEYWORD IN JAVA

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

Object and Class Example: main within the class

In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.

Here, we are creating a main() method inside the class.

File: Student.java

```
//Java Program to illustrate how to define a class and fields
//Defining a Student class.
class Student{
    //defining fields
    int id;//field or data member or instance variable
    String name;
    //creating main method inside the Student class
    public static void main(String args[]){
     //Creating an object or instance
     Student s1=new Student();//creating an object of Student
  //Printing values of the object
    System.out.println(s1.id);//accessing member through reference variable
     System.out.println(s1.name);
    }
```

There are 3 ways to initialize object in Java.

1. By reference variable
2. By method
3. By constructor

OBJECT AND CLASS EXAMPLE: INITIALIZATION THROUGH REFERENCE

```
1. class Student{
2.  int id;
3.  String name;
4. }
5. class TestStudent2{
6.  public static void main(String args[]){
7.   Student s1=new Student();
8.   s1.id=101;
9.   s1.name="Sonoo";
10.  System.out.println(s1.id+" "+s1.name);//printing members with a white space
11. }
12.}
```

2) Object and Class Example: Initialization through method

```
1.  class Student{
2.   int rollno;
3.   String name;
4.   void insertRecord(int r, String n){
5.    rollno=r;
6.    name=n;
7.   }
8.   void displayInformation(){System.out.println(rollno+" "+name);}
9.  }
10.class TestStudent4{
11. public static void main(String args[]){
12.  Student s1=new Student();
13.  Student s2=new Student();
14.  s1.insertRecord(111,"Karan");
15.  s2.insertRecord(222,"Aryan");
16.  s1.displayInformation();
17.  s2.displayInformation();
18. }
19.}
```

3) Object and Class Example: Initialization through a constructor

```
public class Dog {
    // Instance Variables
    String name;
    String breed;
    int age;
    String color;
       Dog(String name, String breed,int age, String color)
    {
        this.name = name;
        this.breed = breed;
        this.age = age;
        this.color = color;
    }
public static void main(String... args)
{
Dog d=new Dog("puppy","reg",10,"red");
System.out.println(d.name+" "+ d.breed+" " + d.color);   } }
```

Inheritance in Java

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms used in Inheritance

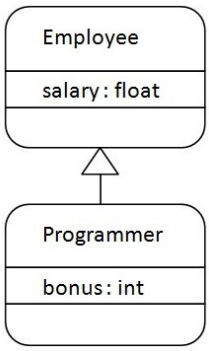
- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

```
1. class Subclass-name extends Superclass-name
2. {
3.   //methods and fields
4. }
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

Java Inheritance Example



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

```
1. class Employee{
2.  float salary=40000;
3. }
4. class Programmer extends Employee{
5.  int bonus=10000;
6.  public static void main(String args[]){
7.    Programmer p=new Programmer();
8.    System.out.println("Programmer salary is:"+p.salary);
9.    System.out.println("Bonus of Programmer is:"+p.bonus);
10.}
11.}
```

Multilevel Inheritance Example

When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

File: TestInheritance2.java

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class BabyDog extends Dog{
8. void weep(){System.out.println("weeping...");}
9. }
10.class TestInheritance2{
11.public static void main(String args[]){
```

```
12.BabyDog d=new BabyDog();
13.d.weep();
14.d.bark();
15.d.eat();
16.}}
```

Hierarchical Inheritance Example

When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

File: TestInheritance3.java

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class Cat extends Animal{
8. void meow(){System.out.println("meowing...");}
9. }
10.class TestInheritance3{
11.public static void main(String args[]){
12.Cat c=new Cat();
13.c.meow();
14.c.eat();
15.//c.bark();//C.T.Error
16.}}
```

NOTE:-To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Lecture -7:

Method Overloading in Java

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

So, we perform method overloading to figure out the program quickly.

Advantage of method overloading

Method overloading *increases the readability of the program*.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

> *In Java, Method Overloading is not possible by changing the return type of the method only.*

1) Method Overloading: changing no. of arguments

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create instance for calling methods.

```
1. class Adder{
2. static int add(int a,int b){return a+b;}
3. static int add(int a,int b,int c){return a+b+c;}
4. }
5. class TestOverloading1{
6. public static void main(String[] args){
7. System.out.println(Adder.add(11,11));
8. System.out.println(Adder.add(11,11,11));
9. }}
```

2) Method Overloading: changing data type of arguments

In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
1. class Adder{
2. static int add(int a, int b){return a+b;}
3. static double add(double a, double b){return a+b;}
4. }
5. class TestOverloading2{
6. public static void main(String[] args){
7. System.out.println(Adder.add(11,11));
8. System.out.println(Adder.add(12.3,12.6));
9. }}
```

Q) Why Method Overloading is not possible by changing the return type of method only?

In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

```
1. class Adder{
2. static int add(int a,int b){return a+b;}
3. static double add(int a,int b){return a+b;}
4. }
5. class TestOverloading3{
6. public static void main(String[] args){
7. System.out.println(Adder.add(11,11));//ambiguity
8. }
```

9. }

NOTE:-Compile Time Error: method add(int,int) is already defined in class Adder

POINT:-System.out.println(Adder.add(11,11)); //Here, how can java determine which sum() method should be called?

Can we overload java main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only. Let's see the simple example:

```
1. class TestOverloading4{
2. public static void main(String[] args){System.out.println("main with String[]");}
3. public static void main(String args){System.out.println("main with String");}
4. public static void main(){System.out.println("main without args");}
5. }
```

Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.

```
1. //Java Program to demonstrate why we need method overriding
2. //Here, we are calling the method of parent class with child
3. //class object.
```

```
4. //Creating a parent class
5. class Vehicle{
6.   void run(){System.out.println("Vehicle is running");}
7. }
8. //Creating a child class
9. class Bike extends Vehicle{
10.  public static void main(String args[]){
11.  //creating an instance of child class
12.  Bike obj = new Bike();
13.  //calling the method with child class instance
14.  obj.run();
15.  }
16.}
```

Example of method overriding

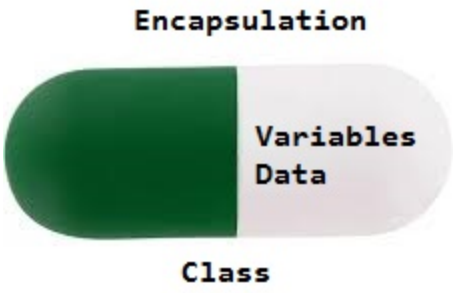
In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

```
1. //Java Program to illustrate the use of Java Method Overriding
2. //Creating a parent class.
3. class Vehicle{
4.   //defining a method
5.   void run(){System.out.println("Vehicle is running");}
6. }
7. //Creating a child class
8. class Bike2 extends Vehicle{
9.   //defining the same method as in the parent class
10.  void run(){System.out.println("Bike is running safely");}
11.
12.  public static void main(String args[]){
13.  Bike2 obj = new Bike2();//creating object
14.  obj.run();//calling method
15.  }
16.}
```

Encapsulation in Java is a fundamental concept in object-oriented programming (OOP) that refers to the bundling of data and methods that operate on that data within a single unit, which is called a class in Java. Java Encapsulation is a way of

hiding the implementation details of a class from outside access and only exposing a public interface that can be used to interact with the class.

In Java, encapsulation is achieved by declaring the instance variables of a class as private, which means they can only be accessed within the class. To allow outside access to the instance variables, public methods called getters and setters are defined, which are used to retrieve and modify the values of the instance variables, respectively. By using getters and setters, the class can enforce its own data validation rules and ensure that its internal state remains consistent.



Implementation of Java Encapsulation

Below is the example with Java Encapsulation:

```
// Java Encapsulation



// Person Class

class Person {

    // Encapsulating the name and age

    // only approachable and used using

    // methods defined

    private String name;

    private int age;



    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; }
}
 // Driver Class
public class Main {
    // main function
    public static void main(String[] args)
    {
        // person object created
        Person person = new Person();
        person.setName("John");
        person.setAge(30);

        // Using methods to get the values from the
        // variables
        System.out.println("Name: " + person.getName());
        System.out.println("Age: " + person.getAge());
    }
```

```
}
```

- Encapsulation can be achieved by Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.
- It is more defined with the setter and getter method.

Advantages of Encapsulation

- **Data Hiding:** it is a way of restricting the access of our data members by hiding the implementation details. Encapsulation also provides a way for data hiding. The user will have no idea about the inner implementation of the class. It will not be visible to the user how the class is storing values in the variables. The user will only know that we are passing the values to a setter method and variables are getting initialized with that value.
- **Increased Flexibility:** We can make the variables of the class read-only or write-only depending on our requirements. If we wish to make the variables read-only then we have to omit the setter methods like setName(), setAge(), etc. from the above program or if we wish to make the variables write-only then we have to omit the get methods like getName(), getAge(), etc. from the above program
- **Reusability:** Encapsulation also improves the re-usability and is easy to change with new requirements.
- **Testing code is easy:** Encapsulated code is easy to test for unit testing.
- **Freedom to programmer in implementing the details of the system:** This is one of the major advantage of encapsulation that it gives the programmer freedom in implementing the details of a system. The only constraint on the programmer is to maintain the abstract interface that outsiders see.

Disadvantages of Encapsulation in Java

- Can lead to increased complexity, especially if not used properly.
- Can make it more difficult to understand how the system works.
- May limit the flexibility of the implementation.

Polymorphism

The word “polymorphism” means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A real-life example of polymorphism is a person who at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee. So the same person exhibits different behavior in different situations. This is called polymorphism. Polymorphism is considered one of the important features of Object-Oriented Programming.

Types of Polymorphism are:

1. Compile-time polymorphism (Method overloading)
1. Run-time polymorphism (Method Overriding)

Note:Above topics already discuss in previous lecture

Lecture-8

ABSTRACT

In Java, abstract is a non-access modifier in java applicable for classes, and methods but **not** variables. It is used to achieve abstraction which is one of the pillars of Object Oriented Programming (OOP). Following are different contexts where *abstract* can be used in Java.

Characteristics of Java abstract Keyword

In Java, the abstract keyword is used to define abstract classes and methods. Here are some of its key characteristics:

- **Abstract classes cannot be instantiated**: An abstract class is a class that cannot be instantiated directly. Instead, it is meant to be extended by other classes, which can provide concrete implementations of its abstract methods.
- **Abstract methods do not have a body**: An abstract method is a method that does not have an implementation. It is declared using the abstract keyword and ends with a semicolon instead of a method body. Subclasses of an abstract class must provide a concrete implementation of all abstract methods defined in the parent class.
- **Abstract classes can have both abstract and concrete methods:** Abstract classes can contain both abstract and concrete methods. Concrete methods are implemented in the abstract class itself and can be used by both the abstract class and its subclasses.
- **Abstract classes can have constructors**: Abstract classes can have constructors, which are used to initialize instance variables and perform other initialization tasks. However, because abstract classes cannot be instantiated directly, their constructors are typically called constructors in concrete subclasses.
- **Abstract classes can contain instance variables**: Abstract classes can contain instance variables, which can be used by both the abstract class and its subclasses. Subclasses can access these variables directly, just like any other instance variables.
- **Abstract classes can implement interfaces:** Abstract classes can implement interfaces, which define a set of methods that must be implemented by any class that implements the interface. In this case, the abstract class must provide concrete implementations of all methods defined in the interface.

Overall, the abstract keyword is a powerful tool for defining abstract classes and methods in Java. By declaring a class or method as abstract, developers can provide a structure for subclassing and ensure that certain methods are implemented in a consistent way across all subclasses.

Abstract Methods in Java

Sometimes, we require just method declaration in super-classes. This can be achieved by specifying the **abstract** type modifier. These methods are sometimes

referred to as *subclasser responsibility* because they have no implementation specified in the super-class. Thus, a subclass must override them to provide a method definition. To declare an abstract method, use this general form:

Example of Abstract class that has an abstract method

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

```
1. abstract class Bike{
2.   abstract void run();
3. }
4. class Honda4 extends Bike{
5. void run(){System.out.println("running safely");}
6. public static void main(String args[]){
7.  Bike obj = new Honda4();
8.  obj.run();
9. }
10.}
```

Interface in Java

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also **represents the IS-A relationship**.

Java Interface Example

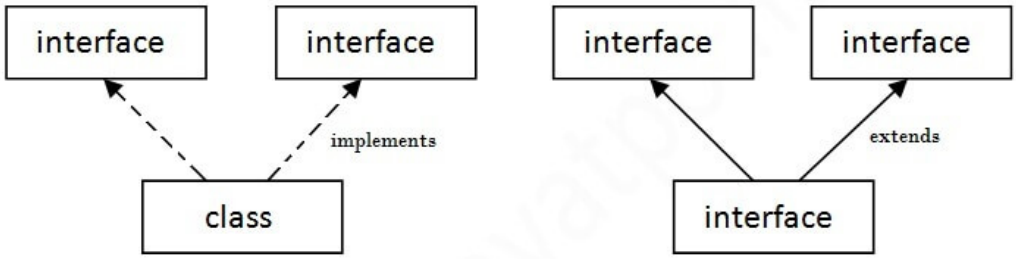
In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

```
1. interface printable{
2. void print();
3. }
4. class A6 implements printable{
5. public void print(){System.out.println("Hello");}
6.
7. public static void main(String args[]){
```

```
8. A6 obj = new A6();
9. obj.print();
10. }
11.}
```

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

```
1. interface Printable{
2. void print();
3. }
4. interface Showable{
5. void show();
6. }
7. class A7 implements Printable,Showable{
8. public void print(){System.out.println("Hello");}
9. public void show(){System.out.println("Welcome");}
10.
11.public static void main(String args[]){
12.A7 obj = new A7();
13.obj.print();
14.obj.show();
15. }
16.}
```

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Example of abstract class and interface in Java

Let's see a simple example where we are using interface and abstract class both.

```
1. Backward Skip 10sPlay //Creating interface that has 4 methods
2. interface A{
3. void a();//bydefault, public and abstract
4. void b();
5. void c();
6. void d();
7. }
8.
9.

   //Creating abstract class that provides the implementation of one method of A interf
   ace
10.abstract class B implements A{
11.public void c(){System.out.println("I am C");}
12.}
13.
14.

   //Creating subclass of abstract class, now we need to provide the implementation of
   rest of the methods
15.class M extends B{
16.public void a(){System.out.println("I am a");}
17.public void b(){System.out.println("I am b");}
18.public void d(){System.out.println("I am d");}
19.}
20.
21.//Creating a test class that calls the methods of A interface
22.class Test5{
23.public static void main(String args[]){
24.A a=new M();
25.a.a();
26.a.b();
27.a.c();
28.a.d();
29.}}
```

Lecture-9:

Java Package

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.

Simple example of java package

The **package keyword** is used to create a package in java.

```
1. //save as Simple.java
2. package mypack;
3. public class Simple{
4.  public static void main(String args[]){
5.     System.out.println("Welcome to package");
6.    }
7. }
```

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

Output:Welcome to package

How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

*1) Using packagename.**

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
1. //save by A.java
2. package pack;
3. public class A{
4.   public void msg(){System.out.println("Hello");}
5. }
1. //save by B.java
2. package mypack;
3. import pack.*;
4.
5. class B{
6.   public static void main(String args[]){
7.    A obj = new A();
8.    obj.msg();
9.   }
10.}
```

Output:Hello

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

```
1. //save by A.java
2. package pack;
3. public class A{
4.   public void msg(){System.out.println("Hello");}
5. }
```

```
1. //save by B.java
2. package mypack;
3. import pack.A;
4.
5. class B{
6.   public static void main(String args[]){
7.    A obj = new A();
8.    obj.msg();
9.   }
10.}
```

Output:Hello

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```
1. //save by A.java
2. package pack;
3. public class A{
4.   public void msg(){System.out.println("Hello");}
5. }
```

```
1. //save by B.java
2. package mypack;
3. class B{
```

```
4.   public static void main(String args[]){
5.    pack.A obj = new pack.A();//using fully qualified name
6.    obj.msg();
7.   }
8. }
```

Output:Hello

Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

CLASSPATH in Java

Package in Java is a mechanism to encapsulate a group of classes, sub-packages, and interfaces. Packages are used for:

- Preventing naming conflicts. For example, there can be two classes with the name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee
- Making searching/locating and usage of classes, interfaces, enumerations, and annotations easier
- Providing controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.

Packages can be considered as data encapsulation (or data-hiding). Here we will be discussing the responsibility of the CLASSPATH environment variable while programming in Java as we move forward we for sure short need usage of importing statements.

Illustration:

```
import org.company.Menu
```

What does this import mean? It makes the Menu class available in the package org.company to our current class. Such that when we call the below command as follows:

```
Menu menu = new Menu();
```

Example

```
// Java Program to Illustrate Usage of importing
// Classes from packages and sub-packages

// Here we are importing all classes from
// java.io (input-output package)
import java.io.*;

// Main class
```

```
class GFG {

    // Main driver method
    public static void main(String[] args)
    {

        // Print statement
        System.out.println("I/O classes are imported from java.io package");
    }
}
```

This package provides for system input and output through data streams, serialization, and the file system. Unless otherwise noted, passing a null argument to a constructor or method in any class or interface in this package will cause a NullPointerException to be thrown. All the classes listed here are imported or if we want to import a specific one then do use it as stated below.

```
import java.util.Scanner ;
```

*The JVM knows where to find the class **Menu**. Now, how will the JVM know this location?*

It is impractical for it to go through every folder on your system and search for it. Thus, using the CLASSPATH variable we provide it the place where we want it to look. We put directories and jars in the CLASSPATH variable.

Lecture-10

JAR FILE

A JAR (Java Archive) is a package file format typically used to aggregate many Java class files and associated metadata and resources (text, images, etc.) into one file to distribute application software or libraries on the Java platform.
In simple words, a JAR file is a file that contains a compressed version of .class files, audio files, image files, or directories. We can imagine a .jar file as a zipped file(.zip) that is created by using WinZip software. Even, WinZip software can be used to extract the contents of a .jar . So you can use them for tasks such as lossless data compression, archiving, decompression, and archive unpacking.
Let us see how to create a .jar file and related commands which help us to work with .jar files

1.1 Create a JAR file

In order to create a .jar file, we can use ***jar cf command*** in the following ways as

Jar -cf jarfilename.jar classfilename.class
Ex- jar –cf A.jar Dream.class

1. 2 View a JAR file

Jar tf A.jar

Note: *When we create .jar files, it automatically receives the default manifest file. There can be only one manifest file in an archive, and it always has the pathname.*

1.3 Extracting a JAR file

Jar xf A.jar

1.5 Running a JAR file

In order to run an application packaged as a JAR file (requires the Main-class manifest header), the following command can be used as listed:

Syntax:

C:\>java -jar A.jar

IMPORT AND STATIC IMPORT NAMING CONVENTION FOR PACKAGE

With the help of static import, we can access the static members of a class directly without class name or any object. For Example: we always use sqrt() method of Math class by using Math class i.e. **Math.sqrt()**, but by using static import we can access sqrt() method directly.
According to SUN microSystem, it will improve the code readability and enhance coding. But according to the programming experts, it will lead to confusion and not good for programming. If there is no specific requirement then we should **not** go for static import.
Advantage of static import:
If user wants to access any static member of class then less coding is required.

Disadvantage of static import:
Static import makes the program unreadable and unmaintainable if you are reusing this feature.

- JAVA

```
// Java Program to illustrate

// calling of predefined methods

// without static import

class Geeks {

    public static void main(String[] args)

    {

        System.out.println(Math.sqrt(4));

        System.out.println(Math.pow(2, 2));

        System.out.println(Math.abs(6.3));
```

```
    }
}
```

Output:
2.0
4.0
6.3

- JAVA

```
// Java Program to illustrate
// calling of predefined methods
// with static import
import static java.lang.Math.*;
class Test2 {
    public static void main(String[] args)
    {
        System.out.println(sqrt(4));
        System.out.println(pow(2, 2));
        System.out.println(abs(6.3));
    }
}
```

Output:
2.0
4.0
6.3

- JAVA

```
// Java to illustrate calling of static member of

// System class without Class name

import static java.lang.Math.*;

import static java.lang.System.*;

class Geeks {

    public static void main(String[] args)

    {

        // We are calling static member of System class

        // directly without System class name

        out.println(sqrt(4));

        out.println(pow(2, 2));

        out.println(abs(6.3));

    }

}
```

Output:
2.0

4.0

6.3

NOTE : System is a class present in java.lang package and out is a static variable present in System class. By the help of static import we are calling it without class name.

UNIT-2

Lecture Delivery Plan:

Lecture-11

11.1 The Idea behind Exception

11.2 Exceptions & Errors

11.3 Types of Exception

Lecture-12

12.1 Control Flow in Exceptions

12.2 JVM Reaction to Exceptions

12.3 Use of try

Lecture-13

13.1 catch

13.2 finally

13.3 throw

Lecture-14

14.1 throws in Exception Handling

14.2 In-built and User Defined Exceptions

14.3 Checked and Un-Checked Exceptions.

Lecture-15

15.1 Byte Streams and Character Streams

15.2 Reading and Writing File in Java.

Lecture-16

16.1 Thread

16.2 Thread Life Cycle

Lecture-17

17.1 Creating Threads

17.2 Thread Priorities

Lecture-18

18.1 Synchronizing Threads

18.2 Inter-thread Communication

Lecture11

Exception Handling

The Idea behind Exception, - The core idea behind exceptions is to create a **controlled mechanism** for handling **unexpected events** that occur during program execution. These events, also called exceptions, can disrupt the normal flow of the program and potentially lead to crashes.

Here are some key aspects of the exception concept:

- **Identifying unexpected events:** Exceptions represent situations that deviate from the program's expected behavior. This could be due to various reasons like:
  - **User errors:** Entering invalid data, attempting forbidden actions.
  - **Resource limitations:** Running out of memory, network issues.
  - **System errors:** Hardware failures, software bugs.
- **Signaling the event:** When an exception occurs, the program "throws" an exception object. This object contains information about the error, such as its type and details.
- **Handling the event:** The program can define specific code blocks called "try-catch" blocks to catch and handle different types of exceptions.
  - The `try` block contains the code that might potentially throw an exception.
  - The `catch` block specifies how to handle the exception if it occurs within the `try` block.
- **Maintaining program flow:** By using exceptions, the program can gracefully recover from errors or provide informative messages to the user, instead of simply crashing. This helps maintain the program's overall stability and user experience.

Overall, exceptions provide a structured approach to dealing with unexpected situations, making programs more robust, maintainable, and user-friendly

What is Exception in Java?

Dictionary Meaning: Exception is an abnormal condition.

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

In Java, Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions. Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object. It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.

Major reasons why an exception Occurs

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out-of-disk memory)
- Code errors
- Opening an unavailable file

Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc. Errors are usually beyond the control of the programmer, and we should not try to handle errors.

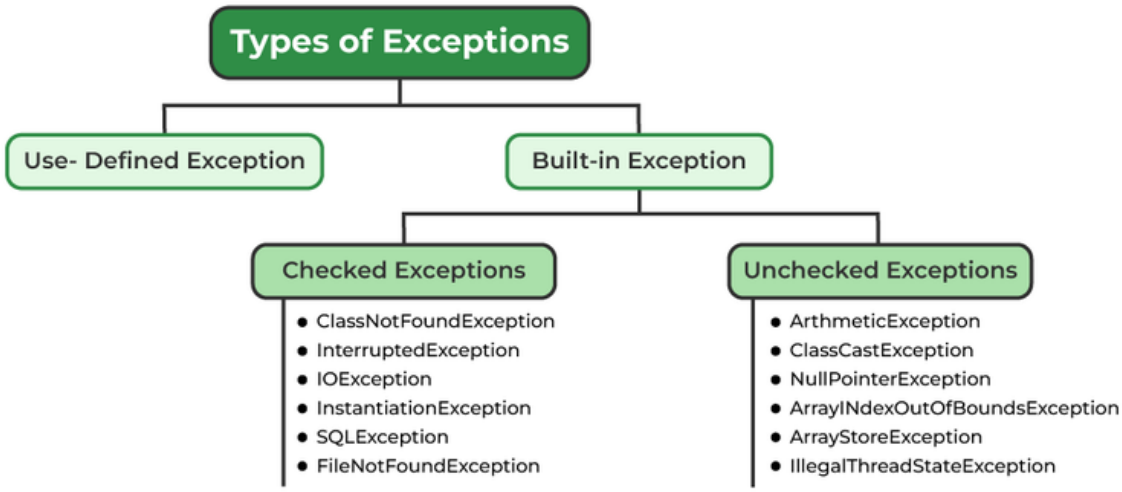
Difference between Error and Exception

Let us discuss the most important part which is the **differences between Error and Exception** that is as follows:

- **Error:** An Error indicates a serious problem that a reasonable application should not try to catch.
- **Exception:** Exception indicates conditions that a reasonable application might try to catch.

Types of Exceptions

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



Exceptions can be categorized in two ways:

1. **Built-in Exceptions**
   - Checked Exception
   - Unchecked Exception
1. **User-Defined Exceptions**

Let us discuss the above-defined listed exception that is as follows:

1. Built-in Exceptions

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

- **Checked Exceptions:** Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.

- **Unchecked Exceptions:** The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error.

Note: *For checked vs unchecked exception, see Checked vs Unchecked Exceptions*

2. User-Defined Exceptions:

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions, which are called 'user-defined Exceptions'.

Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

Keyword	Description
Try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
Catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
Finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
Throw	The "throw" keyword is used to throw an exception.
Throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Java Exception Handling Example

Let's see an example of Java Exception Handling in which we are using a try-catch statement to handle the exception.

JavaExceptionExample.java

```
1. public class JavaExceptionExample{
2. public static void main(String args[]){
3. try{
4. //code that may raise exception
5. int data=100/0;
6. }catch(ArithmeticException e){
7. System.out.println(e);
8. }
9. //rest code of the program
10. System.out.println("rest of the code...");
11. }
12. }
```

OUTPUT

```
Exception in thread main java.lang.ArithmeticException:/ by zero

rest of the code...
```

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Java Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

Let's see a simple example of java multi-catch block.

MultipleCatchBlock1.java

```
1. public class MultipleCatchBlock1 {
2.
3. public static void main(String[] args) {
4.
5. try{
6. int a[]=new int[5];
7. a[5]=30/0;
8. }
```

```
9. catch(ArithmeticException e)
10. {
11. System.out.println("Arithmetic Exception occurs");
12. }
13. catch(ArrayIndexOutOfBoundsException e)
14. {
15. System.out.println("ArrayIndexOutOfBounds Exception occurs");
16. }
17. catch(Exception e)
18. {
19. System.out.println("Parent Exception occurs");
20. }
21. System.out.println("rest of the code");
22. }
23. }
```

Java finally block

Java finally block is a block used to execute important code such as closing the connection, etc.

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
- The important statements to be printed can be placed in the finally block.

TestFinallyBlock.java

```
1. class TestFinallyBlock {
2. public static void main(String args[]){
3. try{
```

```
4. //below code do not throw any exception
5. int data=25/5;
6. System.out.println(data);
7. }
8. //catch won't be executed
9. catch(NullPointerException e){
10. System.out.println(e);
11. }
12. //executed regardless of exception occurred or not
13. finally {
14. System.out.println("finally block is always executed");
15. }
16.
17. System.out.println("rest of phe code...");
18. }
19. }
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock.java

C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock
5
finally block is always executed
rest of the code...
```

When an exception occurr but not handled by the catch block

Let's see the the fillowing example. Here, the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and then the program terminates abnormally.

TestFinallyBlock1.java

```
1. public class TestFinallyBlock1{
2. public static void main(String args[]){
3.
4. try {
5.
6. System.out.println("Inside the try block");
7.
8. //below code throws divide by zero exception
9. int data=25/0;
10. System.out.println(data);
11. }
12. //cannot handle Arithmetic type exception
```

```
13. //can only accept Null Pointer type exception
14. catch(NullPointerException e){
15. System.out.println(e);
16. }
17.
18. //executes regardless of exception occured or not
19. finally {
20. System.out.println("finally block is always executed");
21. }
22. System.out.println ("rest of the code...");
23. }
24. }
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock1.java

C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock1
Inside the try block
finally block is always executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at TestFinallyBlock1.main(TestFinallyBlock1.java:9)
```

Java throw keyword

The Java throw keyword is used to throw an exception explicitly.

We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception. We will discuss custom exceptions later in this section.

```
1. public class TestThrow1 {
2. //function to check if person is eligible to vote or not
3. public static void validate(int age) {
4. if(age<18) {
5. //throw Arithmetic exception if not eligible to vote
6. throw new ArithmeticException("Person is not eligible to vote");
7. }
8. else {
9. System.out.println("Person is eligible to vote!!");
10.}
11.}
12.//main method
13.public static void main(String args[]){
```

```
14. //calling the function
15. validate(13);
16. System.out.println("rest of the code...");
17. }
18. }
```

Lecture -12

Exceptions as Control Flow

Using exceptions as control flow is generally considered a bad practice. While exceptions are a mechanism to stop unexpected behavior in software development, abusing them in expected behaviors can lead to negative consequences.

This practice can reduce the performance of the code

and make it less readable

.**Reasons to Avoid Using Exceptions as Control Flow**

1. **Performance Impact**: Using exceptions as control flow can reduce the performance of the code as a response per unit time
2. **Readability**: It makes the code less readable and hides the programmer's intention, which is considered a bad practice

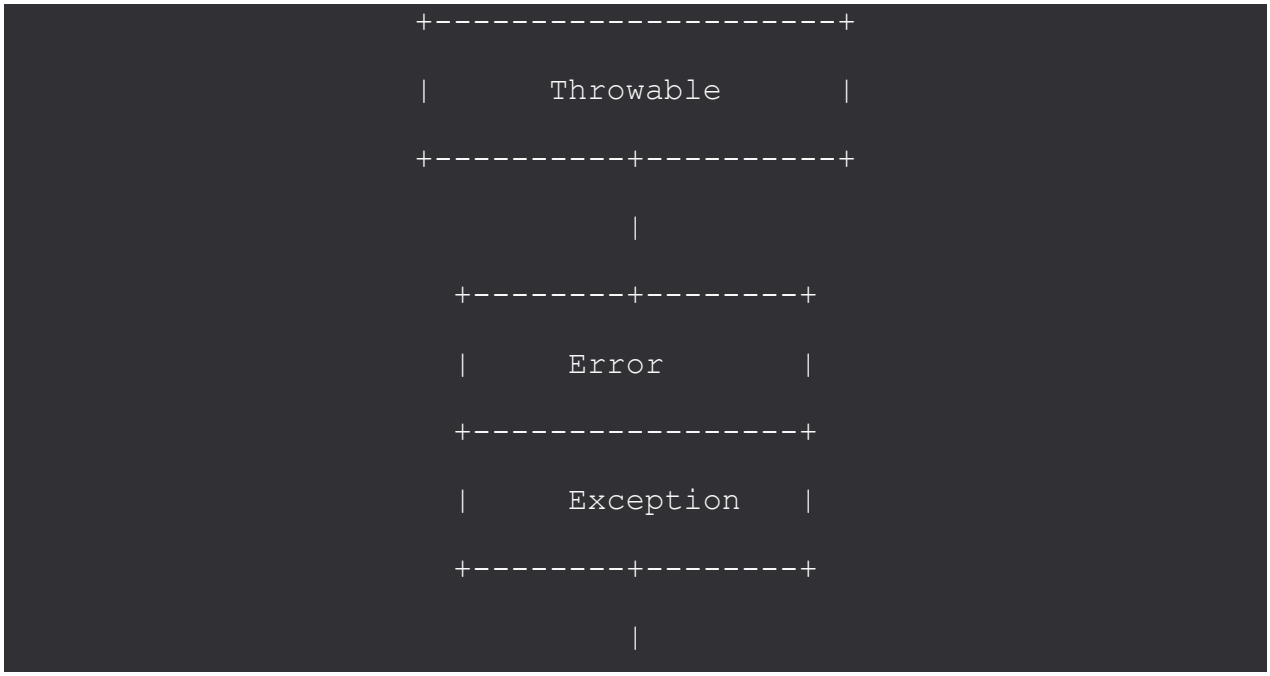
Alternatives to Using Exceptions for Control Flow

Instead of using exceptions for expected behaviours, it is recommended to use control flow statements to handle the logic.

If the behaviours are expected, using control flow statements is generally considered better than throwing an exception and handling it by yourself

JVM Reaction to Exceptions,

Certainly! Here is a diagram illustrating the Java exception hierarchy and the various types of exceptions:

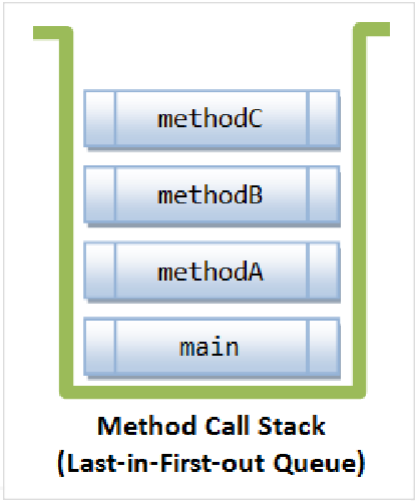


In this hierarchy:

- **Throwable** is the base class for all exceptions and errors in Java.
- **Error** is used by the Java runtime system (JVM) to indicate errors related to the runtime environment itself.
- **Exception** is used for exceptional conditions that user programs should catch. It has two main branches:
    - **RuntimeException**: These are unchecked exceptions that occur at runtime and are not checked by the compiler.
    - **Checked Exception**: These are exceptions that occur at compile time and are checked by the compiler.

This diagram provides a clear overview of the Java exception hierarchy and the different types of exceptions.

```
Exit
methodA
() Exit
main()
```



As seen from the output, the sequence of events is:

1. JVM invoke the main().
2. main() pushed onto call stack, before invoking methodA().
3. methodA() pushed onto call stack, before invoking methodB().
4. methodB() pushed onto call stack, before invoking methodC().
5. methodC() completes.
6. methodB() popped out from call stack and completes.
7. methodA() popped out from the call stack and completes.
8. main() popped out from the call stack and completes. Program exits.

suppose that we modify methodC() to carry out a "divide-by-0" operation, which triggers a ArithmeticException:

```
public static void methodC() {
    System.out.println("Enter  methodC()");
    System.out.println(1 / 0);   // divide-by-0 triggers an ArithmeticException
    System.out.println("Exit methodC()");
}
```

The exception message clearly shows the *method call stack trace* with the relevant statement line numbers:

```
Enter main()

Enter    methodA()
Enter    methodB()
Enter methodC()

Exception in thread "main" java.lang.ArithmeticException: / by zero

        at    MethodCallStackDemo.methodC(MethodCallStackDemo.java:22)    at
        MethodCallStackDemo.methodB(MethodCallStackDemo.java:16)        at
```

MethodC() triggers an ArithmeticException. As it does not handle this exception, it popped off from the call stack immediately. MethodB() also does not handle this exception and popped off the call stack. So does methodA() and main() method. The main() method passes back to JVM, which abruptly terminates the program and print the call stack trace, as shown.

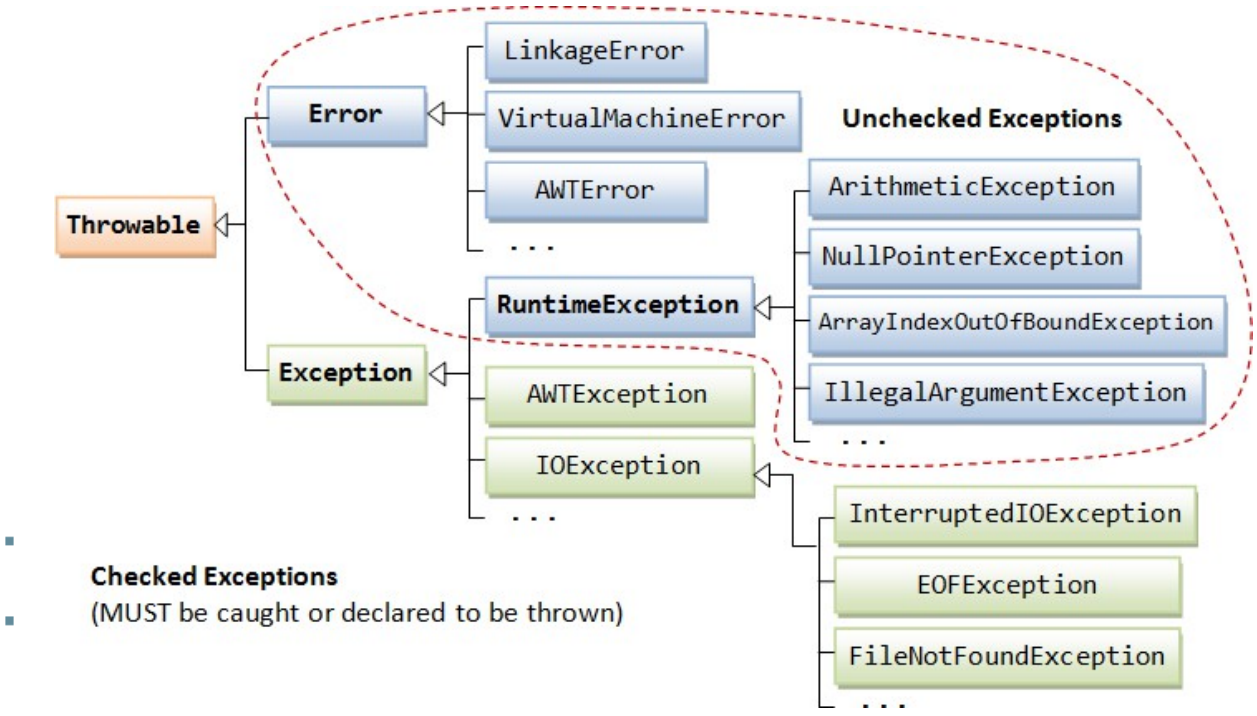
1.3 Exception & Call Stack

When an exception occurs inside a Java method, the method creates an Exception object and passes the Exception object to the JVM (in Java term, the method "throw" an Exception). The Exception object contains the type of the exception, and the state of the program when the exception occurs. The JVM is responsible for finding an *exception handler* to process the Exception object. It searches backward through the call stack until it finds a matching exception handler for that particular class of Exception object (in Java term, it is called "catch" the Exception). If the JVM cannot find a matching exception handler in all the methods in the call stack, it terminates the program.

This process is illustrated as follows. Suppose that methodD() encounters an abnormal condition and throws a XxxException to the JVM. The JVM searches backward through the call stack for a matching exception handler. It finds methodA() having a XxxException handler and passes the exception object to the handler. Notice that methodC() and methodB() are required to declare "throws XxxException" in their method signatures in order to compile the program.

1.4 Exception Classes - Throwable, Error, Exception & RuntimeException

The figure below shows the hierarchy of the Exception classes. The base class for all Exception objects is java.lang.Throwable, together with its two subclasses java.lang.Exception and java.lang.Error.



The Error class describes internal system errors (e.g., VirtualMachineError, LinkageError) that rarely occur. If such an error occurs, there is little that you can do and the program will be terminated by the Java runtime.

The Exception class describes the error caused by your program (e.g. FileNotFoundException, IOException). These errors could be caught and handled by your program (e.g., perform an alternate action or do a graceful exit by closing all the files, network and database connections).

Use of try

Certainly! Here's a diagram illustrating the use of the try...catch block in Java:

```
public class Main {

    public static void main(String[] args) {

        try {

            // Code that may throw an exception

            int[] myNumbers = {1, 2, 3};

            System.out.println(myNumbers[10]); // This line may
throw an exception

        } catch (Exception e) {

            // Code to handle the exception

            System.out.println("Something went wrong.");

        } finally {

            // Code that always executes, regardless of the result

            System.out.println("The 'try catch' is finished.");

        }

    }

}
```

In this diagram:

- The `try` block contains the code that may throw an exception.
- The `catch` block is used to catch and handle the exception. It contains the code that executes when an exception is thrown.
- The `finally` block contains the code that always executes, regardless of whether an exception was thrown or caught.

Lecture-13

CATCH

1.7 try-catch-finally

The syntax of try-catch-finally is:

```
try {
    // main logic, uses methods that may throw Exceptions
    ......
} catch (Exception1 ex) {
    // error handler for Exception1
    ......
} catch (Exception2 ex) {
    // error handler for Exception1
    ......
}  finally  {       // finally is optional
    // clean up codes, always executed regardless of exceptions
    ......
}
```

If no exception occurs during the running of the try-block, all the catch-blocks are skipped, and finally-block will be executed after the try-block. If one of the statements in the try-block throws an exception, the Java runtime ignores the rest of the statements in the try-block, and begins searching for a matching exception handler. It matches the exception type with each of the catch-blocks *sequentially*. If a catch-block catches that exception class or catches a *superclass* of that exception, the statement in that catch-block will be executed. The statements in the finally-block are then executed after that catch-block. The program continues into the next statement after the try-catch-finally, unless it is pre-maturely terminated or branch-out.

If none of the catch-block matches, the exception will be passed up the call stack. The current method executes the finally clause (if any) and popped off the call stack. The caller follows the same procedures to handle the exception.

The finally block is almost certain to be executed, regardless of whether or not exception occurs (unless JVM encountered a severe error or a System.exit()is called in the catchblock).

Example 1

```
1  import java.util.Scanner; import
2  java.io.File;
3  import java.io.FileNotFoundException; public class
4  TryCatchFinally {
5      public static void main(String[] args) { try { // main
6          logic
7              System.out.println("Start of the main logic"); System.out.println("Try
8              opening a file ..."); Scanner in = new Scanner(new File("test.in"));
9
10             System.out.println("File Found, processing the file ..."); System.out.println("End of the main logic");
11
12         } catch (FileNotFoundException ex) {                    // error handling separated from the main logic
13             System.out.println("File Not Found caught ...");
14
           } finally {              // always run regardless of exception status
               System.out.println("finally-block runs regardless of the state of exception");
```

```
15              }
16              // after the try-catch-finally
17              System.out.println("After try-catch-finally, life goes on...");
18
```

```
} finally {       // finally is optional
    // clean up codes, always executed regardless of exceptions
    ......
}
```

Catch

In Java, the `catch` keyword is used as part of the try...catch block to handle exceptions. Here's a summary of its usage based on the provided search results:

- The `catch` statement allows you to define a block of code to be executed if an error occurs in the try block. It comes in pairs with the `try` statement.
    - Syntax:

```
try {
    // Block of code to try
} catch(Exception e) {
    // Block of code to handle errors
}
```

    - The `catch` block catches and handles the exceptions by declaring the type of exception within the parameter. It includes the code that is executed if an exception inside the try block occurs.
    - It is used to handle uncertain conditions of a try block and must be followed by the try block.
    - Multiple catch blocks can be used with a single try block to handle different types of exceptions.

The `catch` block is an essential part of exception handling in Java, allowing developers to gracefully handle errors and prevent the abnormal termination of the program.

Finally

The `finally` block is used to execute important code such as releasing resources, regardless of whether an exception is thrown or not in the `try` block.

- Syntax:

```
try {
    // Block of code to try
} catch(Exception e) {
    // Block of code to handle errors
} finally {
    // Block of code to execute regardless of an exception
}
```

    - The `finally` block always executes, even if an exception is thrown and caught.
    - It is often used to release resources like file handles, database connections, or network connections, to ensure that these resources are properly closed or released.
    - The `finally` block is optional, but when used, it ensures that certain code will always be executed, making it useful for cleanup or finalization tasks.

Throw

the `throw` keyword is used to explicitly throw an exception. Here's a summary of its usage based on the provided search results:

- The `throw` statement is used to throw an exception explicitly within a method or block of code.
    - Syntax:
    - `throw new ExceptionType("Error message");`
    - The `throw` statement is followed by the keyword `new` and the constructor of the exception type to create and throw an instance of that exception.
    - It is typically used to handle exceptional situations where the program encounters an error or an unexpected condition.
    - The exception that is thrown can be a built-in exception class provided by Java, or it can be a custom exception class created by the programmer.
-

throws in Exception Handling,

`throws` keyword is used in method declarations to specify which exceptions are not handled within the method but are instead propagated to the calling code to be handled. Here's an overview of its usage:

Usage of the `throws` Keyword:

- **Syntax:**

```
returnType methodName(parameters) throws ExceptionType1, ExceptionType2, ... {
    // Method body
}
```

- The `throws` keyword is used in the method signature to indicate that the method may throw one or more exceptions of the specified types.
- When a method defines a `throws` clause, it informs the calling code that it is not handling the specified exceptions internally, and it's the responsibility of the calling code to handle or propagate these exceptions.
- The calling code must either handle the exceptions using a `try-catch` block or declare the exceptions to be thrown further up the call stack.
- Example:

```
public void readFile(String fileName) throws IOException {
    // Method implementation that may throw IOException
}
```

In the example above, the `readFile` method declares that it may throw an `IOException`. This means that any code calling the `readFile` method must either handle the `IOException` or declare it to be thrown further up the call stack

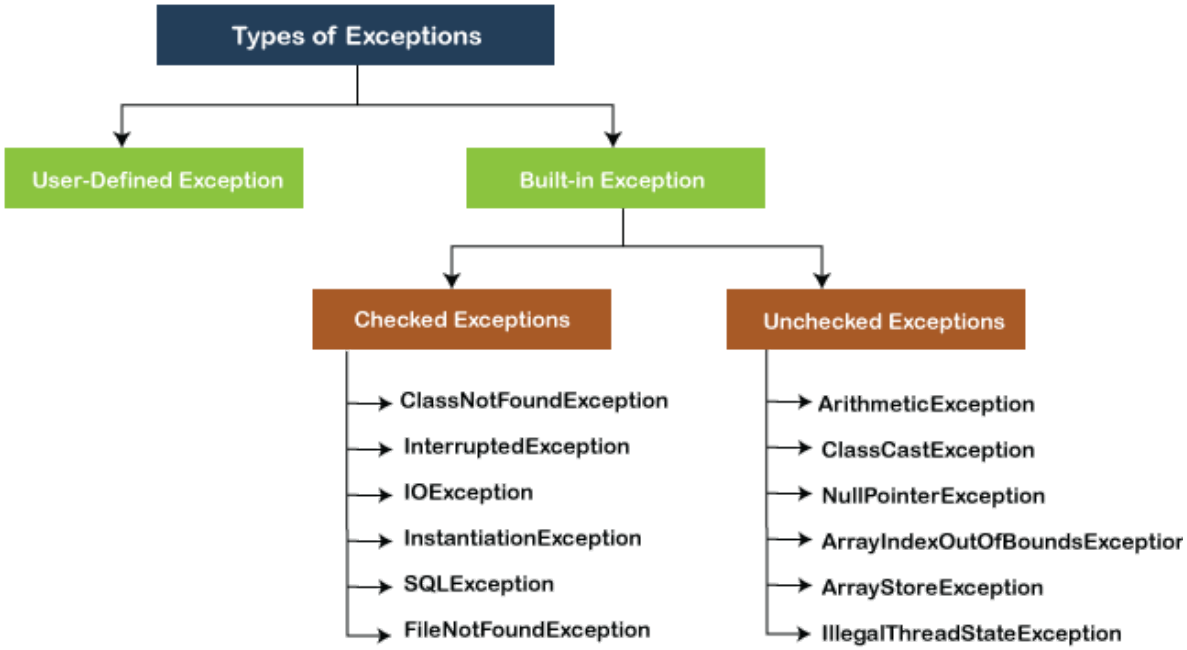
Lecture-14

Types of Exception in Java

In Java, **exception** is an event that occurs during the execution of a program and disrupts the normal flow of the program's instructions. Bugs or errors that we don't want and restrict our program's normal execution of code are referred to as **exceptions**. In this section, we will focus on the **types of exceptions in Java** and the differences between the two.

Exceptions can be categorized into two ways:

1. Built-in Exceptions
    - Checked Exception
    - Unchecked Exception
2. User-Defined Exceptions



Built-in Exception

Exceptions that are already available in **Java libraries** are referred to as **built-in exception**. These exceptions are able to define the error situation so that we can understand the reason of getting this error. It can be categorized into two broad categories, i.e., **checked exceptions** and **unchecked exception**.

Checked Exception

Checked exceptions are called **compile-time** exceptions because these exceptions are checked at compile-time by the compiler. The compiler ensures whether the programmer handles the exception or not. The programmer should have to handle the exception; otherwise, the system has shown a compilation error.

CheckedExceptionExample.java

```
1. import java.io.*;
2. class CheckedExceptionExample {
3.     public static void main(String args[]) {
4.         FileInputStream file_data = null;
5.         file_data = new FileInputStream("C:/Users/ajeet/OneDrive/Desktop/Hello.txt");
6.         int m;
7.         while(( m = file_data.read() ) != -1) {
8.             System.out.print((char)m);
9.         }
10.        file_data.close();
11.    }
12. }
```

In the above code, we are trying to read the **Hello.txt** file and display its data or content on the screen. The program throws the following exceptions:

1. The **FileInputStream(File filename)** constructor throws the **FileNotFoundException** that is checked exception.
2. The **read()** method of the **FileInputStream** class throws the **IOException**.
3. The **close()** method also throws the IOException.

Output:

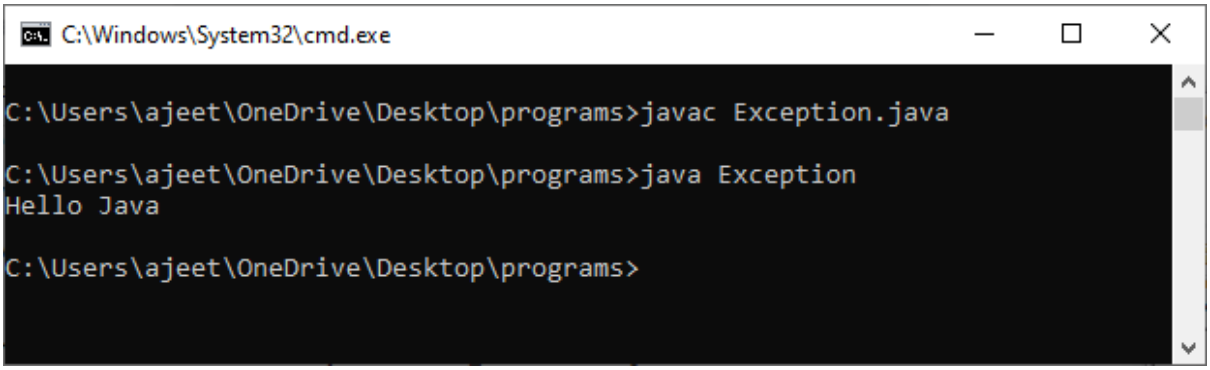
How to resolve the error?

There are basically two ways through which we can solve these errors.

1) The exceptions occur in the main method. We can get rid from these compilation errors by declaring the exception in the main method using **the throws** We only declare the IOException, not FileNotFoundException, because of the child-parent relationship. The IOException class is the parent class of FileNotFoundException, so this exception will automatically cover by IOException. We will declare the exception in the following way:

```
1. class Exception{
2.     public static void main(String args[])  throws IOException {
3.         ...
4.         ...
5. }
```

If we compile and run the code, the errors will disappear, and we will see the data of the file.

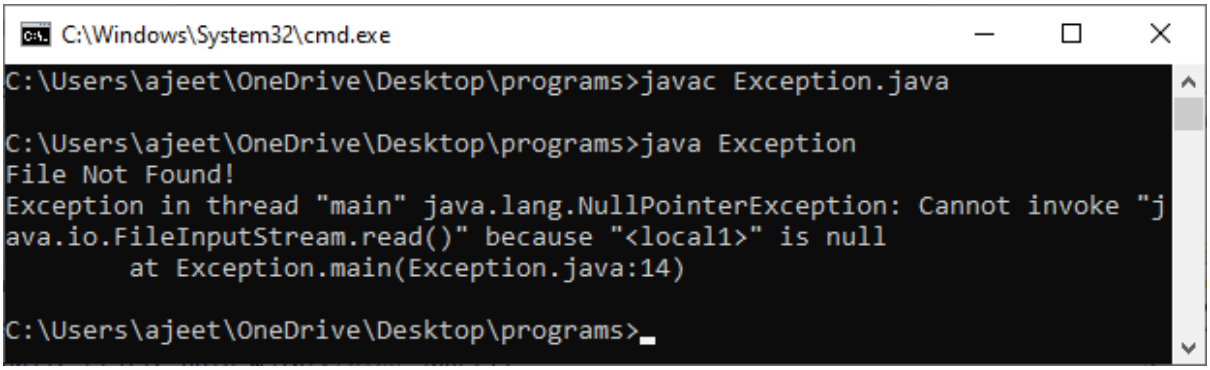


2) We can also handle these exception using **try-catch** However, the way which we have used above is not correct. We have to a give meaningful message for each exception type. By doing that it would be easy to understand the error. We will use the try-catch block in the following way:

Exception.java

```
1. import java.io.*;
2. class Exception{
3.     public static void main(String args[]) {
4.         FileInputStream file_data = null;
5.         try{
6.             file_data = new FileInputStream("C:/Users/ajeet/OneDrive/Desktop/programs/Hell.txt");
7.         }catch(FileNotFoundException fnfe){
8.             System.out.println("File Not Found!");
9.         }
10.        int m;
11.        try{
12.            while(( m = file_data.read() ) != -1) {
13.                System.out.print((char)m);
14.            }
15.            file_data.close();
16.        }catch(IOException ioe){
17.            System.out.println("I/O error occurred: "+ioe);
18.        }
19.    }
20. }
```

We will see a proper error message **"File Not Found!"** on the console because there is no such file in that location.



Unchecked Exceptions

The **unchecked** exceptions are just opposite to the **checked** exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error. Usually, it occurs when the user provides bad data during the interaction with the program.

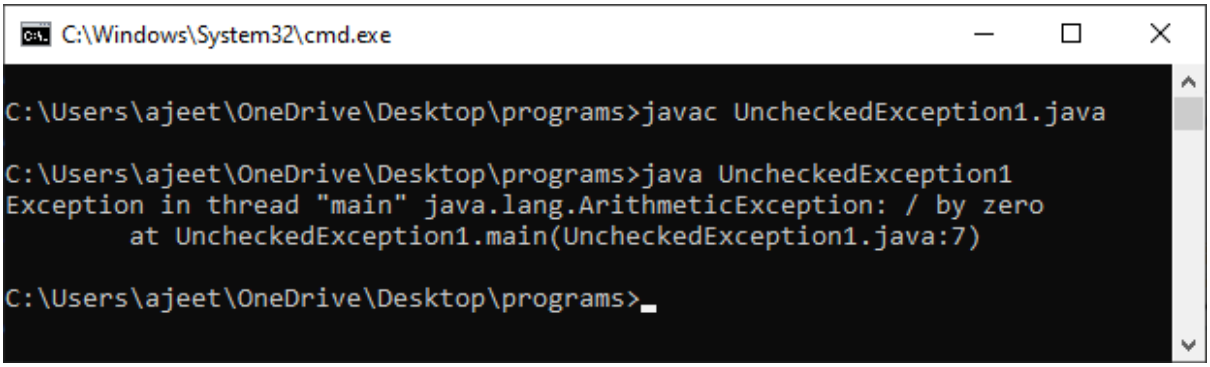
Note: The RuntimeException class is able to resolve all the unchecked exceptions because of the child-parent relationship.

UncheckedExceptionExample1.java

```
1. class UncheckedExceptionExample1 {
2.    public static void main(String args[])
3.    {
4.     int postive = 35;
5.     int zero = 0;
6.     int result = positive/zero;
7.     //Give Unchecked Exception here.
8. System.out.println(result);
9.    }
10. }
```

In the above program, we have divided 35 by 0. The code would be compiled successfully, but it will throw an ArithmeticException error at runtime. On dividing a number by 0 throws the divide by zero exception that is a uncheck exception.

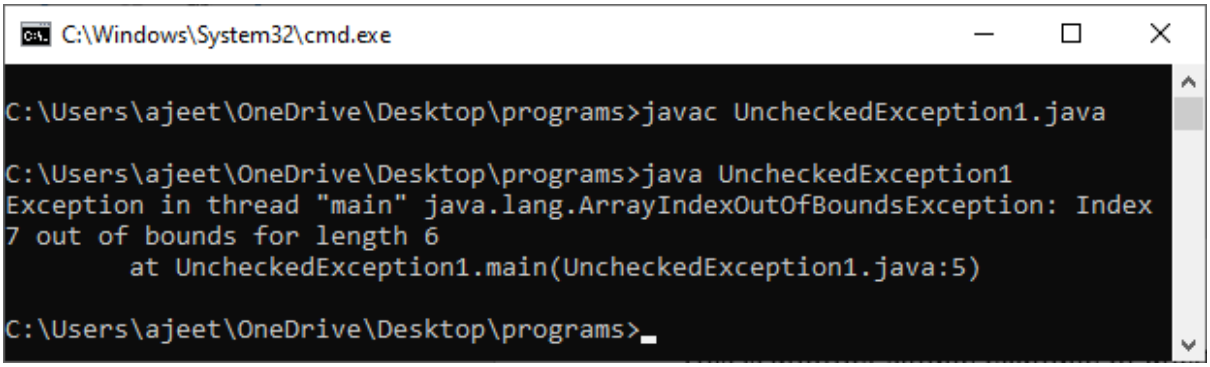
Output:



UncheckedException1.java

```
1. class UncheckedException1 {
2.   public static void main(String args[])
3.   {
4.    int num[] ={10,20,30,40,50,60};
5.    System.out.println(num[7]);
6.   }
7. }
```

Output:



In the above code, we are trying to get the element located at position 7, but the length of the array is 6. The code compiles successfully, but throws the ArrayIndexOutOfBoundsException at runtime.

User-defined Exception

In Java, we already have some built-in exception classes like **ArrayIndexOutOfBoundsException**,**NullPointerException**, and **ArithmeticException**. These exceptions are restricted to trigger on some predefined conditions. In Java, we can write our own exception class by extends the Exception class. We can throw our own exception on a particular condition using the throw keyword. For creating a user-defined exception, we should have basic knowledge of **the try-catch** block and **throw** keyword.

Let's write a Java program and create user-defined exception.

Difference between Checked and Unchecked Exception

S.No	Checked Exception	Unchecked Exception
1.	These exceptions are checked at compile time. These exceptions are handled at compile time too.	These exceptions are just opposite to the checked exceptions. These exceptions are not checked and handled at compile time.
2.	These exceptions are direct subclasses of exception but not extended from RuntimeException class.	They are the direct subclasses of the RuntimeException class.
3.	The code gives a compilation error in the case when a method throws a checked exception. The compiler is not able to handle the exception on its own.	The code compiles without any error because the exceptions escape the notice of the compiler. These exceptions are the results of user-created errors in programming logic.
4.	These exceptions mostly occur when the probability of failure is too high.	These exceptions occur mostly due to programming mistakes.
5.	Common checked exceptions include IOException, DataAccessException, InterruptedException, etc.	Common unchecked exceptions include ArithmeticException, InvalidClassException, NullPointerException, etc.
6.	These exceptions are propagated using the throws keyword.	These are automatically propagated.
7.	It is required to provide the try-catch and try-finally block to handle the checked exception.	In the case of unchecked exception it is not mandatory.

Lecture-15

ByteStream Classes in Java

ByteStream classes are used to read bytes from the input stream and write bytes to the output stream. In other words, we can say that ByteStream classes read/write the data of 8-bits. We can store video, audio, characters, etc., by using ByteStream classes. These classes are part of the java.io package.

The ByteStream classes are divided into two types of classes, i.e., InputStream and OutputStream. These classes are abstract and the super classes of all the Input/Output stream classes.

InputStream Class

The InputStream class provides methods to read bytes from a file, console or memory. It is an abstract class and can't be instantiated; however, various classes inherit the InputStream class and override its methods. The subclasses of InputStream class are given in the following table.

SN	Class	Description
1	BufferedInputStream	This class provides methods to read bytes from the buffer.
2	ByteArrayInputStream	This class provides methods to read bytes from the byte array.
3	DataInputStream	This class provides methods to read Java primitive data types.
4	FileInputStream	This class provides methods to read bytes from a file.
5	FilterInputStream	This class contains methods to read bytes from the other input streams, which are used as the primary source of data.
6	ObjectInputStream	This class provides methods to read objects.
7	PipedInputStream	This class provides methods to read from a piped output stream to which the piped input stream must be connected.
8	SequenceInputStream	This class provides methods to connect multiple Input Stream and read data from them.

The InputStream class contains various methods to read the data from an input stream. These methods are overridden by the classes that inherit the InputStream class. However, the methods are given in the following table.

SN	Method	Description
1	int read()	This method returns an integer, an integral representation of the next available byte of the input. The integer -1 is returned once the end of the input is encountered.

2	int read (byte buffer [])	This method is used to read the specified buffer length bytes from the input and returns the total number of bytes successfully read. It returns -1 once the end of the input is encountered.
3	int read (byte buffer [], int loc, int nBytes)	This method is used to read the 'nBytes' bytes from the buffer starting at a specified location, 'loc'. It returns the total number of bytes successfully read from the input. It returns -1 once the end of the input is encountered.
4	int available ()	This method returns the number of bytes that are available to read.
5	Void mark(int nBytes)	This method is used to mark the current position in the input stream until the specified nBytes are read.
6	void reset ()	This method is used to reset the input pointer to the previously set mark.
7	long skip (long nBytes)	This method is used to skip the nBytes of the input stream and returns the total number of bytes that are skipped.
8	void close ()	This method is used to close the input source. If an attempt is made to read even after the closing, IOException is thrown by the method.

OutputStream Class

The OutputStream is an abstract class that is used to write 8-bit bytes to the stream. It is the superclass of all the output stream classes. This class can't be instantiated; however, it is inherited by various subclasses that are given in the following table.

SN	Class	Description
1	BufferedOutputStream	This class provides methods to write the bytes to the buffer.
2	ByteArrayOutputStream	This class provides methods to write bytes to the byte array.
3	DataOutputStream	This class provides methods to write the java primitive data types.
4	FileOutputStream	This class provides methods to write bytes to a file.
5	FilterOutputStream	This class provides methods to write to other output streams.
6	ObjectOutputStream	This class provides methods to write objects.

7	PipedOutputStream	It provides methods to write bytes to a piped output stream.
8		

The
met

SN	Method	Description
1	void write (int i)	This method is used to write the specified single byte to the output stream.
2	void write (byte buffer [])	It is used to write a byte array to the output stream.
3	Void write(bytes buffer[],int loc, int nBytes)	It is used to write nByte bytes to the output stream from the buffer starting at the specified location.
4	void flush ()	It is used to flush the output stream and writes the pending buffered bytes.
5	void close ()	It is used to close the output stream. However, if we try to close the already closed output stream, the IOException will be thrown by this method.

Example:

The following example uses the ByteArrayInputStream to create an input stream from a byte array "content". We use the read() method to read the content from an input stream. We have also used the write() method on a FileOutputStream object to write the byte array content in the file.

Consider the following example.

```
1. import java.io.ByteArrayInputStream;
2. import java.io.File;
3. import java.io.FileInputStream;
4. import java.io.FileNotFoundException;
5. import java.io.FileOutputStream;
6. import java.io.IOException;
7. public class InputOutputStreamExample {
8. public static void main(String[] args) throws IOException {
9. // TODO Auto-generated method stub
10. byte content[] = "Jtp is the best website to learn new technologies".getBytes();
11. ByteArrayInputStream inputStream = new ByteArrayInputStream(content);
12.
13. inputStream.read(content);
```

```
14.
15. File newFile = new File("/Users/MyUser/Desktop/MyNewFile.doc");
16. FileOutputStream outputStream = new FileOutputStream(newFile);
17. outputStream.write(content);
18.
19. }
20.
21.
22. }
```

Output:

```
A new file MyNewFile.doc will be created on desktop with the content "Jtp is the best web
```

CharacterStream Classes in Java

The java.io package provides CharacterStream classes to overcome the limitations of ByteStream classes, which can only handle the 8-bit bytes and is not compatible to work directly with the Unicode characters. CharacterStream classes are used to work with 16-bit Unicode characters. They can perform operations on characters, char arrays and Strings.

However, the CharacterStream classes are mainly used to read characters from the source and write them to the destination. For this purpose, the CharacterStream classes are divided into two types of classes, I.e., Reader class and Writer class.

Reader Class

Reader class is used to read the 16-bit characters from the input stream. However, it is an abstract class and can't be instantiated, but there are various subclasses that inherit the Reader class and override the methods of the Reader class. All methods of the Reader class throw an IOException. The subclasses of the Reader class are given in the following table.

SN	Class	Description
1.	BufferedReader	This class provides methods to read characters from the buffer.
2.	CharArrayReader	This class provides methods to read characters from the char array.
3.	FileReader	This class provides methods to read characters from the file.
4.	FilterReader	This class provides methods to read characters from the underlying character input stream.
5	InputStreamReader	This class provides methods to convert bytes to characters.

6	PipedReader	This class provides methods to read characters from the connected piped output stream.
7	StringReader	This class provides methods to read characters from a string.

The Reader class methods are given in the following table.

SN	Method	Description
1	int read()	This method returns the integral representation of the next character present in the input. It returns -1 if the end of the input is encountered.
2	int read(char buffer[])	This method is used to read from the specified buffer. It returns the total number of characters successfully read. It returns -1 if the end of the input is encountered.
3	int read(char buffer[], int loc, int nChars)	This method is used to read the specified nChars from the buffer at the specified location. It returns the total number of characters successfully read.
4	void mark(int nchars)	This method is used to mark the current position in the input stream until nChars characters are read.
5	void reset()	This method is used to reset the input pointer to the previous set mark.
6	long skip(long nChars)	This method is used to skip the specified nChars characters from the input stream and returns the number of characters skipped.
7	boolean ready()	This method returns a boolean value true if the next request of input is ready. Otherwise, it returns false.
8	void close()	This method is used to close the input stream. However, if the program attempts to access the input, it generates IOException.

Writer Class

Writer class is used to write 16-bit Unicode characters to the output stream. The methods of the Writer class generate IOException. Like Reader class, Writer class is also an abstract class that cannot be instantiated; therefore, the subclasses of the Writer class are used to write the characters onto the output stream. The subclasses of the Writer class are given in the below table.

SN	Class	Description

1	BufferedWriter	This class provides methods to write characters to the buffer.
2	FileWriter	This class provides methods to write characters to the file.
3	CharArrayWriter	This class provides methods to write the characters to the character array.
4	OutpuStreamWriter	This class provides methods to convert from bytes to characters.
5	PipedWriter	This class provides methods to write the characters to the piped output stream.
6	StringWriter	This class provides methods to write the characters to the string.

To write the characters to the output stream, the Write class provides various methods given in the following table.

SN	Method	Description
1	void write()	This method is used to write the data to the output stream.
2	void write(int i)	This method is used to write a single character to the output stream.
3	Void write(char buffer[])	This method is used to write the array of characters to the output stream.
4	void write(char buffer [],int loc, int nChars)	This method is used to write the nChars characters to the character array from the specified location.
5	void close ()	This method is used to close the output stream. However, this generates the IOException if an attempt is made to write to the output stream after closing the stream.
6	void flush ()	This method is used to flush the output stream and writes the waiting buffered characters.

Java Create and Write To Files

Create a File

To create a file in Java, you can use the `createNewFile()` method. This method returns a boolean value: `true` if the file was successfully created, and `false` if the file

already exists. Note that the method is enclosed in a `try...catch` block. This is necessary because it throws an `IOException` if an error occurs (if the file cannot be created for some reason):

Write To a File

In the following example, we use the `FileWriter` class together with its `write()` method to write some text to the file we created in the example above. Note that when you are done writing to the file, you should close it with the `close()` method:

```
import java.io.FileWriter;   // Import the FileWriter class
import java.io.IOException;  // Import the IOException class to handle errors


public class WriteToFile {
  public static void main(String[] args) {
    try {
      FileWriter myWriter = new FileWriter("filename.txt");
      myWriter.write("Files in Java might be tricky, but it is fun enough!");
      myWriter.close();
      System.out.println("Successfully wrote to the file.");
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

The output will be:

```
Successfully wrote to the file.
```

Lecture-16

Thread

What is Thread in java

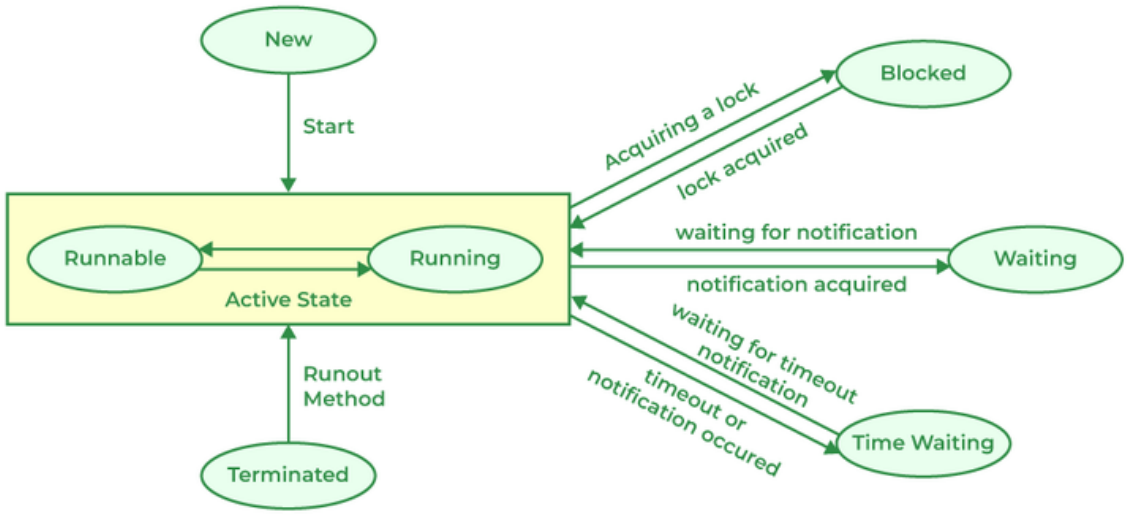
A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

1. New State
1. Runnable State
1. Blocked State
1. Waiting State
1. Timed Waiting State
1. Terminated State

The diagram shown below represents various states of a thread at any instant in time.



States of Thread in its Lifecycle

Life Cycle of a Thread

There are multiple states of the thread in a lifecycle as mentioned below:

1. **New Thread:** When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn't started to execute.
1. **Runnable State:** A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.
   A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lie in a runnable state.
1. **Blocked:** The thread will be in blocked state when it is trying to acquire a lock but currently the lock is acquired by the other thread. The thread will move from the blocked state to runnable state when it acquires the lock.
1. **Waiting state:** The thread will be in waiting state when it calls wait() method or join() method. It will move to the runnable state when other thread will notify or that thread will be terminated.
1. **Timed Waiting:** A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is

received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.

1. **Terminated State:** A thread terminates because of either of the following reasons:
   - Because it exits normally. This happens when the code of the thread has been entirely executed by the program.
   - Because there occurred some unusual erroneous event, like a segmentation fault or an unhandled exception.

Lecture-17

How to Create a Thread in Java

There are two ways to create a thread:

First, you can create a thread using the thread class (extend syntax). This provides you with constructors and methods for creating and operating on threads.

The thread class extends the object class and implements a runnable interface. The thread class in Java is the main class on which Java's multithreading system is based.

Second, you can create a thread using a runnable interface. You can use this method when you know that the class with the instance is intended to be executed by the thread itself.

The runnable interface is an interface in Java which is used to execute concurrent thread. The runnable interface has only one method which is `run()`.
Now let's see the syntax of both of them:

How to use the extend syntax:

```
public class Main extends thread {
  public void test() {
    System.out.println("Threads are very helpful in java");
  }
}
```

Priority of a Thread (Thread Priority)

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. Note that not only JVM a Java programmer can also assign the priorities of a thread explicitly in a Java program.

Setter & Getter Method of Thread Priority

Let's discuss the setter and getter method of the thread priority.

public final int getPriority(): The java.lang.Thread.getPriority() method returns the priority of the given thread.

public final void setPriority(int newPriority): The java.lang.Thread.setPriority() method updates or assign the priority of the thread to newPriority. The method throws IllegalArgumentException if the value newPriority goes out of the range, which is 1 (minimum) to 10 (maximum).

3 constants defined in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example

- Java

```
// Java Program to Illustrate Priorities in Multithreading

// via help of getPriority() and setPriority() method



// Importing required classes

import java.lang.*;



// Main class

class ThreadDemo extends Thread {



    // Method 1

    // run() method for the thread that is called

    // as soon as start() is invoked for thread in main()

    public void run()

    {

        // Print statement

        System.out.println("Inside run method");

    }
```

```
    // Main driver method
    public static void main(String[] args)
    {
        // Creating random threads
        // with the help of above class
        ThreadDemo t1 = new ThreadDemo();
        ThreadDemo t2 = new ThreadDemo();
        ThreadDemo t3 = new ThreadDemo();

        // Thread 1
        // Display the priority of above thread
        // using getPriority() method
        System.out.println("t1 thread priority : "
                           + t1.getPriority());

        // Thread 1
        // Display the priority of above thread
        System.out.println("t2 thread priority : "
                           + t2.getPriority());

        // Thread 3
        System.out.println("t3 thread priority : "
                           + t3.getPriority());
```

```
// Setting priorities of above threads by

// passing integer arguments

t1.setPriority(2);

t2.setPriority(5);

t3.setPriority(8);



// t3.setPriority(21); will throw

// IllegalArgumentException



// 2

System.out.println("t1 thread priority : "

                   + t1.getPriority());



// 5

System.out.println("t2 thread priority : "

                   + t2.getPriority());



// 8

System.out.println("t3 thread priority : "

                   + t3.getPriority());



// Main thread
```

```
        // Displays the name of
        // currently executing Thread
        System.out.println(
            "Currently Executing Thread : "
            + Thread.currentThread().getName());

        System.out.println(
            "Main thread priority : "
            + Thread.currentThread().getPriority());

        // Main thread priority is set to 10
        Thread.currentThread().setPriority(10);

        System.out.println(
            "Main thread priority : "
            + Thread.currentThread().getPriority());
    }
}
```

Output

```
t1 thread priority : 5
t2 thread priority : 5
t3 thread priority : 5
t1 thread priority : 2
t2 thread priority : 5
```

```
t3 thread priority : 8

Currently Executing Thread : main

Main thread priority : 5

Main thread priority : 10
```

Output explanation:

- Thread with the highest priority will get an execution chance prior to other threads. Suppose there are 3 threads t1, t2, and t3 with priorities 4, 6, and 1. So, thread t2 will execute first based on maximum priority 6 after that t1 will execute and then t3.
- The default priority for the main thread is always 5, it can be changed later. The default priority for all other threads depends on the priority of the parent thread.

Lecture-18

Synchronization in Java

Synchronization in Java is the capability to control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization?

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
    3. Static synchronization.
2. Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. It can be achieved by using the following three ways:

1. By Using Synchronized Method
2. By Using Synchronized Block
3. By Using Static Synchronization

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package java.util.concurrent.locks contains several lock implementations.

Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

TestSynchronization1.java

```
1. class Table{
2. void printTable(int n){//method not synchronized
3.   for(int i=1;i<=5;i++){
```

```
4.       System.out.println(n*i);
5.       try{
6.        Thread.sleep(400);
7.       }catch(Exception e){System.out.println(e);}
8.     }
9.
10. }
11.}
12.
13.class MyThread1 extends Thread{
14.Table t;
15.MyThread1(Table t){
16.this.t=t;
17.}
18.public void run(){
19.t.printTable(5);
20.}
21.
22.}
23.class MyThread2 extends Thread{
24.Table t;
25.MyThread2(Table t){
26.this.t=t;
27.}
28.public void run(){
29.t.printTable(100);
30.}
31.}
32.
33.class TestSynchronization1{
34.public static void main(String args[]){
35.Table obj = new Table();//only one object
36.MyThread1 t1=new MyThread1(obj);
37.MyThread2 t2=new MyThread2(obj);
38.t1.start();
39.t2.start();
40.}
41.}
```

Output:

```
5
```

```
100
10
200
15
300
20
400
25
500
```

Java Synchronized Method

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

TestSynchronization2.java

```
1. //example of java synchronized method
2. class Table{
3.  synchronized void printTable(int n){//synchronized method
4.    for(int i=1;i<=5;i++){
5.      System.out.println(n*i);
6.      try{
7.       Thread.sleep(400);
8.      }catch(Exception e){System.out.println(e);}
9.    }
10.
11. }
12.}
13.
14.class MyThread1 extends Thread{
15.Table t;
16.MyThread1(Table t){
17.this.t=t;
18.}
19.public void run(){
```

```
t.printTable(5);
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}

public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

Inter-thread Communication in Java

Inter-thread communication or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of **Object class**:

ADVERTISEMENT

- wait()
- notify()
- notifyAll()

1) wait() method

The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
public final void wait()throws InterruptedException	It waits until object is notified.
public final void wait(long timeout)throws InterruptedException	It waits for the specified amount of time.

2) notify() method

The notify() method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

Syntax:

```
1. public final void notify()
```

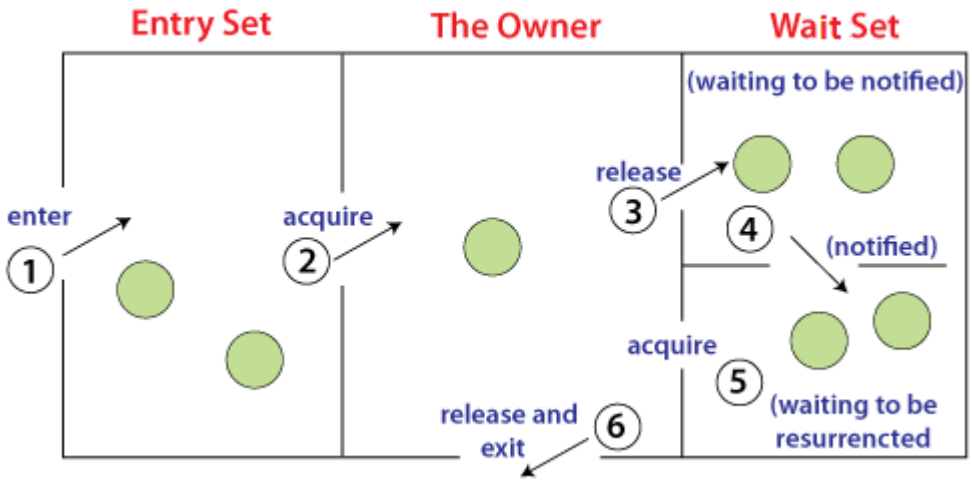
3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor.

Syntax:

```
1. public final void notifyAll()
```

Understanding the process of inter-thread communication



The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by on thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?

It is because they are related to lock and object has a lock.

Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

wait()	sleep()
The wait() method releases the lock.	The sleep() method doesn't release the lock.
It is a method of Object class	It is a method of Thread class
It is the non-static method	It is the static method

It should be notified by notify() or notifyAll() methods	After the specified amount of time, sleep is completed.

Example of Inter Thread Communication in Java

Let's see the simple example of inter thread communication.

Test.java

```
1. class Customer{
2. int amount=10000;
3.
4. synchronized void withdraw(int amount){
5. System.out.println("going to withdraw...");
6.
7. if(this.amount<amount){
8. System.out.println("Less balance; waiting for deposit...");
9. try{wait();}catch(Exception e){}
10.}
11.this.amount-=amount;
12.System.out.println("withdraw completed...");
13.}
14.
15.synchronized void deposit(int amount){
16.System.out.println("going to deposit...");
17.this.amount+=amount;
18.System.out.println("deposit completed... ");
19.notify();
20.}
21.}
22.
23.class Test{
24.public static void main(String args[]){
25.final Customer c=new Customer();
26.new Thread(){
27.public void run(){c.withdraw(15000);}
28.}.start();
29.new Thread(){
30.public void run(){c.deposit(10000);}
31.}.start();
32.
```

33. }}

Output:

```
going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed
```

This operation is used to rename the existing file.

UNIT-3

Lecture Delivery Plan:

Lecture-19 Functional Interfaces

19.1 Lambda Expression

19.2 Method References

Lecture-20

20.1 Stream API,

20.2 Default Methods

20.3 Static Method

Lecture-21

22.1 Base64 Encode and Decode

22.2 ForEach Method

22.3 Try-withresources

Lecture-22

32.1 Type Annotations

32.2 Repeating Annotations

32.3 Java Module System,

Lecture-23

23.1 Diamond Syntax with Inner Anonymous Class

23.2 Local Variable Type Inference

Lecture-24

24.1 Switch Expressions

24.2 Yield Keyword

Lecture-25

25.1 Text Blocks,

25.2 Records

Lecture-26

26.1 Sealed Classes

Lecture 19

Java Functional Interfaces

An Interface that contains exactly one abstract method is known as functional interface. It can have any number of default, static methods but can contain only one abstract method. It can also declare methods of object class.

Functional Interface is also known as Single Abstract Method Interfaces or SAM Interfaces. It is a new feature in Java, which helps to achieve functional programming approach.

Functional Interface is additionally recognized as **Single Abstract Method Interfaces**. In short, they are also known as **SAM interfaces**. Functional interfaces in Java are the new feature that provides users with the approach of fundamental programming.
Functional interfaces are included in Java SE 8 with Lambda expressions and Method references in order to make code more readable, clean, and straightforward. Functional interfaces are interfaces that ensure that they include precisely only one abstract method. Functional interfaces are used and executed by representing the interface with an **annotation called @FunctionalInterface**. As described earlier, functional interfaces can contain only one abstract method. However, they can include any quantity of default and static methods.

Java Lambda Expressions

Lambda expression is a new and important feature of Java which was included in Java SE 8.

It provides a clear and concise way to represent one method interface using an expression.

It is very useful in collection library.

It helps to iterate, filter and extract data from collection.

The Lambda expression is used to provide the implementation of an interface which has

functional interface.

It saves a lot of code. In case of lambda expression, we don't need to define the method again

For providing the implementation.

Here, we just write the implementation code.

Java lambda expression is treated as a function, so compiler does not create .class file.

Lambda expression provides implementation of *functional interface*. An interface which has only

one abstract method is called functional interface.

Java provides an anotation @*FunctionalInterface*, which is used to declare an interface as

functional interface.

Java lambda expression is consisted of three components.

1) Argument-list: It can be empty or non-empty as well.

2) Arrow-token: It is used to link arguments-list and body of expression.

3) Body: It contains expressions and statements for lambda expression.

Java Method References

Java provides a new feature called method reference in Java 8.

Method reference is used to refer method of functional interface.

It is compact and easy form of lambda expression. Each time when you are using lambda expression

to just referring a method, you can replace your lambda expression with method reference.

Types of Method References

There are following types of method references in java:

1. Reference to a static method.
2. Reference to an instance method.
3. Reference to a constructor.

Java 8 Stream

Java provides a new additional package in Java 8 called java.util.stream.

This package consists of class interfaces and enum to allows functional-style operations on

the elements. You can use stream by importing java.util.stream package.

Stream does not store elements. It simply conveys elements from a source such as a

data structure, an array, or an I/O channel, through a pipeline of computational operations.

- Stream is functional in nature. Operations performed on a stream does not modify it's source.
- For example, filtering a Stream obtained from a collection produces a new Stream without
- the filtered elements, rather than removing elements from the source collection.
- Stream is lazy and evaluates code only when required.
- The elements of a stream are only visited once during the life of a stream. Like an Iterator,
- a new stream must be generated to revisit the same elements of the source.

You can use stream to filter, collect, print, and convert from one data structure to other etc.

In the following examples, we have apply various operations with the help of stream.

Lecture -20

Streams API

The newly added Stream API (**java.util.stream**) introduces real-world functional-style programming into the Java. This is by far the most comprehensive addition to Java library intended to make Java developers significantly more productive by allowing them to write effective, clean, and concise code.

Stream API makes collections processing greatly simplified (but it is not limited to Java collections only as we will see later). Let us take start off with simple class called Task.

```
public class Streams  {
    private enum Status {

        OPEN, CLOSED

    };


    private static final class Task {
        private final Status status;
        private final Integer points;


        Task( final Status status, final Integer points ) {
            this.status = status;

            this.points = points;

        }


        public Integer getPoints() {
            return points;

        }


        public Status getStatus() {
            return status;
```

Task has some notion of points (or pseudo-complexity) and can be either **OPEN** or **CLOSED**. And then let us introduce a small collection of tasks to play with.

```
final Collection< Task > tasks = Arrays.asList(
    new Task( Status.OPEN, 5 ),

    new Task( Status.OPEN, 13 ),
    new Task( Status.CLOSED, 8 )
```

The first question we are going to address is how many points in total all **OPEN** tasks have? Up to Java 8, the usual solution for it would be some sort of **foreach** iteration. But in Java 8 the answers is streams: a sequence of elements supporting sequential and parallel aggregate operations.

```
// Calculate total points of all active tasks using sum()
final long totalPointsOfOpenTasks = tasks

    .stream()

    .filter( task -> task.getStatus() == Status.OPEN )

    .mapToInt( Task::getPoints )
```

And the output on the console looks like that:

```
Total points: 18
```

There are a couple of things going on here. Firstly, the tasks collection is converted to its stream representation. Then, the **filter** operation on stream filters out all **CLOSED** tasks. On next step, the **mapToInt** operation converts the stream of **Tasks** to the stream of **Integers** using **Task::getPoints** method of the each task instance. And lastly, all points are summed up using **sum** method, producing the final result.

Before moving on to the next examples, there are some notes to keep in mind about streams (more details here). Stream operations are divided into intermediate and terminal operations.

Intermediate operations return a new stream. They are always lazy, executing an intermediate operation such as **filter** does not actually perform any filtering, but instead creates a new stream that, when traversed, contains the elements of the initial stream that match the given predicate

Terminal operations, such as **forEach** or **sum**, may traverse the stream to produce a result or a side-effect. After the terminal operation is performed, the stream pipeline is considered consumed, and can no longer be used. In almost all cases, terminal operations are eager, completing their traversal of the underlying data source.

Yet another value proposition of the streams is out-of-the box support of parallel processing. Let us take a look on this example, which does sums the points of all the tasks.

```
// Calculate total points of all tasks
final double totalPoints = tasks

   .stream()

   .parallel()

   .map( task -> task.getPoints() ) // or map( Task::getPoints )
```

It is very similar to the first example except the fact that we try to process all the

tasks in **parallel** and calculate the final result using **reduce** method.

Here is the console output:

```
Total points (all tasks): 26.0
```

Often, there is a need to performing a grouping of the collection elements by some criteria. Streams can help with that as well as an example below demonstrates.

```
// Group tasks by their status

final Map< Status, List< Task > > map = tasks

    .stream()
```

The console output of this example looks like that:

```
{CLOSED=[[CLOSED, 8]], OPEN=[[OPEN, 5], [OPEN, 13]]}
```

To finish up with the tasks example, let us calculate the overall percentage (or weight) of each task across the whole collection, based on its points.

```
// Calculate the weight of each tasks (as percent of total points)
final Collection< String > result = tasks

    .stream()                                       // Stream< String >

    .mapToInt( Task::getPoints )                    // IntStream

    .asLongStream()                                 // LongStream
```

```
    .collect( Collectors.toList() );                    // List< String >

System.out.println( result );
```

The console output is just here:

```
[19%, 50%, 30%]
```

And lastly, as we mentioned before, the Stream API is not only about Java collections. The typical I/O operations like reading the text file line by line is a very good candidate to benefit from stream processing. Here is a small example to confirm that.

```
final Path path = new File( filename ).toPath();

try( Stream< String > lines = Files.lines( path, StandardCharsets.UTF_8 ) ) {
    lines.onClose( () -> System.out.println("Done!") ).forEach( System.out::println );
```

The **onClose** method called on the stream returns an equivalent stream with an additional close handler. Close handlers are run when the **close()** method is called on the stream.

Stream API together with Lambdas and Method References baked by Interface's Default and Static Methods is the Java 8 response to the modern paradigms in software development. For more details, please refer to official documentation.

4.1 Date/Time API (JSR 310)

Java 8 makes one more take on date and time management by delivering New Date-Time API (JSR 310). Date and time manip- ulation is being one of the worst pain points for Java developers. The standard **java.util.Date** followed by **java.util.Calendar** hasn't improved the situation at all (arguably, made it even more confusing).

That is how Joda-Time was born: the great alternative date/time API for Java. The Java 8's New Date-Time API (JSR 310) was heavily influenced by Joda-Time and took the best of it. The new **java.time** package contains all the classes for date, time, date/time, time zones, instants, duration, and clocks manipulation. In the design of the API the immutability has been taken into account very seriously: no change allowed (the tough lesson learnt from * java.util.Calendar*). If the modification is required, the new instance of respective class will be returned.

Let us take a look on key classes and examples of their usages. The first class is **Clock** which provides access to the current instant, date and time using a time-zone. **Clock** can be used instead of **System.currentTimeMillis()** and **TimeZone.getDefault()**.

```
// Get the system clock as UTC offset
final Clock clock = Clock.systemUTC();
System.out.println( clock.instant() );
System.out.println( clock.millis() );
```

The sample output on a console:

```
2014-04-12T15:19:29.282Z

1397315969360
```

Other new classes we are going to look at are **LocaleDate** and **LocalTime**. **LocaleDate** holds only the date part without a time-

zone in the ISO-8601 calendar system. Respectively, **LocaleTime** holds only the time part without time-zone in the ISO-8601 calendar system. Both **LocaleDate** and **LocaleTime** could be created from **Clock**.

```
// Get the local date and local time
final LocalDate date = LocalDate.now();

final LocalDate dateFromClock = LocalDate.now( clock );


System.out.println( date );
System.out.println( dateFromClock );
```

```
final LocalTime timeFromClock = LocalTime.now( clock );


System.out.println( time );
```

The sample output on a console:

```
2014-04-12

2014-04-12

11:25:54.568

15:25:54.568
```

The **LocalDateTime** combines together **LocaleDate** and **LocalTime** and holds a date with time but without a time-zone in the ISO-8601 calendar system. A quick example is shown below.

```
// Get the local date/time

final LocalDateTime datetime = LocalDateTime.now();

final LocalDateTime datetimeFromClock = LocalDateTime.now( clock );
```

The sample output on a console:

```
2014-04-12T11:37:52.309
```

If case you need a date/time for particular timezone, the **ZonedDateTime** is here to help. It holds a date with time and with a time-zone in the ISO-8601 calendar system. Here are a couple of examples for different timezones.

```
// Get the zoned date/time

final ZonedDateTime zonedDatetime = ZonedDateTime.now();

final ZonedDateTime zonedDatetimeFromClock = ZonedDateTime.now( clock );

final ZonedDateTime zonedDatetimeFromZone = ZonedDateTime.now( ZoneId.of( "America/ ↩

    Los_Angeles" ) );
```

The sample output on a console:

```
2014-04-12T11:47:01.017-04:00[America/New_York]

2014-04-12T15:47:01.017Z
```

And finally, let us take a look on **Duration** class: an amount of time in terms of seconds and

nanoseconds. It makes very easy to compute the different between two dates. Let us take a look on that.

```
// Get duration between two dates

final LocalDateTime from = LocalDateTime.of( 2014, Month.APRIL, 16, 0, 0, 0 );

final LocalDateTime to = LocalDateTime.of( 2015, Month.APRIL, 16, 23, 59, 59 );


final Duration duration = Duration.between( from, to );
```

The example above computes the duration (in days and hours) between two dates, **16 April 2014** and **16 April 2015**. Here is the sample output on a console:

```
Duration in days: 365
Duration in hours: 8783
```

he overall impression about Java 8's new date/time API is very, very positive. Partially, because of the battle-proved foundation it is built upon (Joda-Time), partially because this time it was finally tackled seriously and developer voices have been heard. For more details please refer to official documentation.

2.1 Interface's Default and Static Methods

Java 8 extends interface declarations with two new concepts: default and static methods. Default methods make interfaces somewhat similar to traits but serve a bit different goal. They allow adding new methods to existing interfaces without breaking the binary compatibility with the code written for older versions of those interfaces.

The difference between default methods and abstract methods is that abstract methods are required to be implemented. But default methods are not. Instead, each interface must provide so called default implementation and all the implementers will inherit it by default (with a possibility to override this default implementation if needed). Let us take a look on example below.

```
private interface Defaulable {

    // Interfaces now allow default methods, the implementer may or

    // may not implement (override) them.
    default String notRequired() {

        return "Default implementation";

    }
```

```
private static class OverridableImpl implements Defaulable {
    @Override

    public String notRequired() {

        return "Overridden implementation";
```

The interface **Defaulable** declares a default method **notRequired()** using keyword **default** as part of the method definition. One of the classes, **DefaultableImpl**, implements this interface leaving the default method implementation as-is. Another one, **OverridableImpl** , overrides the default implementation and provides its own.

Another interesting feature delivered by Java 8 is that interfaces can declare (and provide implementation) of static methods. Here is an example.

```
private interface DefaulableFactory {

    // Interfaces now allow static methods

    static Defaulable create( Supplier< Defaulable > supplier ) {
        return supplier.get();
```

The small code snippet below glues together the default methods and static methods from the examples above.

```
public static void main( String[] args ) {

    Defaulable defaulable = DefaulableFactory.create( DefaultableImpl::new );
    System.out.println( defaulable.notRequired() );


    defaulable = DefaulableFactory.create( OverridableImpl::new );
    System.out.println( defaulable.notRequired() );
```

The console output of this program looks like that:

```
Default implementation
Overridden implementation
```

Default methods implementation on JVM is very efficient and is supported by the byte code instructions for method invocation. Default methods allowed existing Java interfaces to evolve without breaking the compilation process. The good examples are the plethora of methods added to **java.util.Collection** interface: **stream()**, **parallelStream()**, **forEach()**, **removeIf()**, . . .

Though being powerful, default methods should be used with a caution: before declaring method as default it is better to think twice if it is really needed as it may cause ambiguity and compilation errors in complex hierarchies. For more details please refer to official documentation.

2.2 Method References

Method references provide the useful syntax to refer directly to exiting methods or constructors of Java classes or objects (in- stances). With conjunction of Lambdas expressions, method references make the language constructs look compact and concise,

leaving off boilerplate.

Below, considering the class **Car** as an example of different method definitions, let us distinguish four supported types of method references.

```
public static class Car {

    public static Car create( final Supplier< Car > supplier ) {
        return supplier.get();

    }
```

```
        System.out.println( "Collided " + car.toString() );
    }

    public void follow( final Car another ) {
        System.out.println( "Following the " + another.toString() );
    }

    public void repair() {
        System.out.println( "Repaired " + this.toString() );
    }
}
```

The first type of method references is constructor reference with the syntax **Class::new** or alternatively, for generics, **Class< T >::new**. Please notice that the constructor has no arguments.

```
final Car car = Car.create( Car::new );

final List< Car > cars = Arrays.asList( car );
```

The second type is reference to static method with the syntax **Class::static_method**. Please notice that the method accepts exactly one parameter of type **Car**.

```
cars.forEach( Car::collide );
```

The third type is reference to instance method of arbitrary object of specific type with the syntax **Class::method**. Please notice, no arguments are accepted by the method.

```
cars.forEach( Car::repair );
```

And the last, fourth type is reference to instance method of particular class instance the syntax **instance::method**. Please notice that method accepts exactly one parameter of type **Car**.

```
final Car police = Car.create( Car::new );
cars.forEach( police::follow );
```

Running all those examples as a Java program produces following output on a console (the actual **Car** instances might be different):

```
Collided com.javacodegeeks.java8.method.references.MethodReferences$Car@7a81197d
Repaired com.javacodegeeks.java8.method.references.MethodReferences$Car@7a81197d
Following the com.javacodegeeks.java8.method.references.MethodReferences$Car@7a81197d
```

For more examples and details on method references, please refer to official documentation.

Lecture -21

Java Base64 Encode and Decode

Java provides a class Base64 to deal with encryption. You can encrypt and decrypt your data by using provided methods. You need to import java.util.Base64 in your source file to use its methods.

This class provides three different encoders and decoders to encrypt information at each level. You can use these methods at the following levels.

Basic Encoding and Decoding

It uses the Base64 alphabet specified by Java in RFC 4648 and RFC 2045 for encoding and decoding operations. The encoder does not add any line separator character. The decoder rejects data that contains characters outside the base64 alphabet.

URL and Filename Encoding and Decoding

It uses the Base64 alphabet specified by Java in RFC 4648 for encoding and decoding operations. The encoder does not add any line separator character. The decoder rejects data that contains characters outside the base64 alphabet.

MIME

It uses the Base64 alphabet as specified in RFC 2045 for encoding and decoding operations. The encoded output must be represented in lines of no more than 76 characters each and uses a carriage return '\r' followed immediately by a linefeed '\n' as the line separator. No line separator is added to the end of the encoded output. All line separators or other characters not found in the base64 alphabet table are ignored in decoding operation.

Java Default Methods

Java provides a facility to create default methods inside the interface. Methods which are defined inside the interface and tagged with default are known as default methods. These methods are non-abstract methods.

Java Default Method Example

In the following example, Sayable is a functional interface that contains a default and an abstract method. The concept of default method is used to define a method with default implementation. You can override default method also to provide more specific implementation for the method.

Java Default Method Example

In the following example, Sayable is a functional interface that contains a default and an abstract method. The concept of default method is used to define a method with default implementation. You can override default method also to provide more specific implementation for the method.

Let's see a simple

```
1. interface Sayable{
2. // Default method
3. default void say(){
4. System.out.println("Hello, this is default method");
5. }
6. // Abstract method
7. void sayMore(String msg);
8. }
9. public class DefaultMethods implements Sayable{
10. public void sayMore(String msg){       // implementing abstract method
11. System.out.println(msg);
12. }
13. public static void main(String[] args) {
14. DefaultMethods dm = new DefaultMethods();
15. dm.say();   // calling default method
16. dm.sayMore("Work is worship");  // calling abstract method
17.
18. }
19. }
```

Java forEach loop

Java provides a new method forEach() to iterate the elements. It is defined in Iterable and Stream interface. It is a default method defined in the Iterable interface. Collection classes which extends Iterable interface can use forEach loop to iterate elements.

This method takes a single parameter which is a functional interface. So, you can pass lambda expression as an argument.

forEach() Signature in Iterable Interface

```
default void forEach(Consumer<super T>action)
```

Java 8 forEach() example 1

```
1. import java.util.ArrayList;
2. import java.util.List;
3. public class ForEachExample {
4. public static void main(String[] args) {
5. List<String> gamesList = new ArrayList<String>();
6. gamesList.add("Football");
7. gamesList.add("Cricket");
8. gamesList.add("Chess");
9. gamesList.add("Hocky");
10. System.out.println("------------Iterating by passing lambda expression--------------");
11. gamesList.forEach(games -> System.out.println(games));
12.
13. }
14. }
```

Java 9 Try With Resource Enhancement

Java introduced **try-with-resource** feature in Java 7 that helps to close resource automatically after being used.

In other words, we can say that we don't need to close resources (file, connection, network etc) explicitly, try-with-resource close that automatically by using AutoClosable interface.

In Java 7, try-with-resources has a limitation that requires resource to declare locally within its block.

Example Java 7 Resource Declared within resource block

```
1. import java.io.FileNotFoundException;
2. import java.io.FileOutputStream;
3. public class FinalVariable {
4. public static void main(String[] args) throws FileNotFoundException {
5. try(FileOutputStream fileStream=new FileOutputStream("javatpoint.txt");){
6. String greeting = "Welcome to javaTpoint.";
7. byte b[] = greeting.getBytes();
8. fileStream.write(b);
9. System.out.println("File written");
10.}catch(Exception e) {
11.System.out.println(e);
12.}
13.}
            }
```

Lecture -22

Java Type Annotations

Java 8 has included two new features repeating and type annotations in its prior annotations topic. In early Java versions, you can apply annotations only to declarations. After releasing of Java SE 8 , annotations can be

applied to any type use. It means that annotations can be used anywhere you use a type. For example, if you want to avoid NullPointerException in your code, you can declare a string variable like this:

```
1. @NonNull String str;
```

Java Repeating Annotations

In Java 8 release, Java allows you to repeating annotations in your source code. It is helpful when you want to reuse annotation for the same class. You can repeat an annotation anywhere that you would use a standard annotation.

For compatibility reasons, repeating annotations are stored in a container annotation that is automatically generated by the Java compiler. In order for the compiler to do this, two declarations are required in your code.

1. Declare a repeatable annotation type
2. Declare the containing annotation type

Java 9 Module System

Java Module System is a major change in Java 9 version.

Java added this feature to collect Java packages and code into a single unit called *module*.

In earlier versions of Java, there was no concept of module to create modular Java applications,

that why size of application increased

and difficult to move around.

Even JDK itself was too heavy in size, in Java 8, **rt.jar** file size is around 64MB.

To deal with situation, **Java 9 restructured JDK into set of modules** so that we can use only

required module for our project.

Apart from JDK, Java also allows us to create our own modules so that

we can develop module-based application.

The module system includes various tools and options that are given below.

- Includes various options to the Java tools **javac, jlink and java** where we can specify module paths that locates to the location of module.
- Modular JAR file is introduced. This JAR contains module-info.class file in its root folder.
- JMOD format is introduced, which is a packaging format similar to JAR except it can include native code and configuration files.
- The JDK and JRE both are reconstructed to accommodate modules. It improves performance, security and maintainability.
- Java defines a new URI scheme for naming modules, classes and resources.

Module is a collection of Java programs or softwares. To describe a module,

a Java file **module-info.java** is required. This file also known as module descriptor and

defines the following

- Module name
- What does it export
- What does it require?

Lecture-23

Java 9 Anonymous Inner Classes Improvement

Java 9 introduced a new feature that allows us to use diamond operator with anonymous classes. Using the diamond with anonymous classes was not allowed in Java 7.

In Java 9, as long as the inferred type is denotable, we can use the diamond operator when we create an anonymous inner class.

Data types that can be written in Java program like int, String etc are called denotable types. Java 9 compiler is enough smart and now can infer type.

Note: This feature is included to Java 9, to add type inference in anonymous inner classes.

Let's see an example, in which we are using diamond operator with inner class without specifying type.

Java 9 Anonymous Inner Classes Example

```
1. abstract class ABCD<T>{
2. abstract T show(T a, T b);
3. }
4. public class TypeInferExample {
5. public static void main(String[] args) {
6. ABCD<String> a = new ABCD<>() { // diamond operator is empty, compiler infer type
7. String show(String a, String b) {
8. return a+b;
9. }
10. };
11. String result = a.show("Java","9");
12. System.out.println(result);
13. }
14. }
15.
```

Diamond operator for Anonymous Inner Class with Examples in Java

Diamond Operator: Diamond operator was introduced in Java 7 as a new feature.The main purpose of the diamond operator is to simplify the use of generics when creating an object. It avoids unchecked warnings in a program and makes the program more readable. The diamond operator could not be used with Anonymous inner classes in JDK 7. In JDK 9, it can be used with the anonymous class as well to simplify code and improves readability. Before

JDK 7, we have to create an object with Generic type on both side of the expression like:

With the help of Diamond operator, we can create an object without mentioning the generic type on the right hand side of the expression. But the problem is it will only work with normal classes.

```
abstract class Geeksforgeeks<T> {

abstract T add(T num1, T num2);

}


public class Geeks {

public static void main(String[] args)

{

Geeksforgeeks<Integer> obj = new Geeksforgeeks<>() {

Integer add(Integer n1, Integer n2)

{

return (n1 + n2);

}

};

Integer result = obj.add(10, 20);

System.out.println("Addition of two numbers: " + result);

}

}
```

What is Local Variable type inference?

Local variable type inference is a feature in Java 10 that allows the developer to skip the type declaration associated with local variables (those defined inside method definitions, initialization blocks, for-loops, and other blocks like if-else), and the type is inferred by the JDK. It will, then, be the job of the compiler to figure out the datatype of the variable.

```
class A {
```

```
public static void main(String a[])

{

var x = "Hi there";

System.out.println(x)

}

}
```

Java Switch Statement

The Java *switch statement* executes one statement from multiple conditions. It is like if-else-if ladder statement. The switch statement works with byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long. Since Java 7, you can use strings in the switch statement.

```
1. public class SwitchExample {
2. public static void main(String[] args) {
3. //Declaring a variable for switch expression
4. int number=20;
5. //Switch expression
6. switch(number){
7. //Case statements
8. case 10: System.out.println("10");
9. break;
10. case 20: System.out.println("20");
11. break;
12. case 30: System.out.println("30");
13. break;
14. //Default case statement
15. default:System.out.println("Not in 10, 20 or 30");
16. }
17. }
18. }
```

2. The *yield* Keyword

The *yield* keyword lets us exit a *switch* expression by returning a value that becomes the value of the *switch* expression.

This means we can assign the value of a *switch* expression to a variable.

Lastly, by using *yield* in a *switch* expression, we get an implicit check that we're covering our cases, which makes our code more robust.

Let's look at some examples.

yield with Arrow Operator

To start, let's say we have the following *enum* and *switch* statement:

```
public enum Number {
ONE, TWO, THREE, FOUR;
}

String message;
switch (number) {
case ONE:
message = "Got a 1";
break;
case TWO:
message = "Got a 2";
break;
default:
message = "More than 2";
}
```

Copy

Let's convert this to a *switch* expression and use the *yield* keyword along with the arrow operator:

```
String message = switch (number) {
case ONE -> {
yield "Got a 1";
}
case TWO -> {
yield "Got a 2";
}
default -> {
yield "More than 2";
}
};
```

Copy

yield sets the value of the *switch* expression depending on the value of *number*.

2.2. *yield* with Colon Delimiter

We can also create a *switch* expression using *yield* with the colon delimiter:

```
String message = switch (number) {
case ONE:
yield "Got a 1";
case TWO:
```

```
yield "Got a 2";
default:
yield "More than 2";
};
```

This code behaves the same as in the previous section. But the arrow operator is clearer and also less prone to forgetting *yield* (or *break*) statements.

We should note that we can't mix colon and arrow delimiters within the same *switch* expression.

TEXT BLOCK

Text blocks start with a *“””* (three double-quote marks) followed by optional whitespaces and a newline. The most simple example looks like this:

```
String example = """

Example text""";
```

Note that the result type of a text block is still a *String.* Text blocks just provide us with another way to write *String* literals in our source code.

A text block is an alternative form of Java string representation that can be used anywhere a traditional double-quoted string literal can be used. Text blocks begin with a “”” (3 double-quote marks) observed through non-obligatory whitespaces and a newline. For example:

```
// Using a literal string

String text1 = "Geeks For Geeks";



// Using a text block

String text2 = """

Geeks For Geeks""";
```

RECORD

As developers and software engineers, our aim is to always design ways to obtain maximum efficiency and if we need to write less code for it, then that's a blessing.

In Java, a record is a special type of class declaration aimed at reducing the boilerplate code. Java records were introduced with the intention to be used as a fast way to create data carrier classes, i.e. the classes whose objective is to simply contain data and carry it between modules, also known as POJOs (Plain Old Java Objects) and DTOs (Data Transfer Objects). Record was introduced in Java SE 14 as a preview feature, which is a feature whose design, implementation, and specification are complete but it is not a permanent addition to the language, which means that the feature may or may not exist in the future versions of the language. Java SE 15 extends the preview feature with additional capabilities such as local record classes.

In Java, we have the concept of abstract classes. It is mandatory to inherit from these classes since objects of these classes cannot be instantiated. On the other hand, there is a concept of a final class in Java, which cannot be inherited or extended by any other class. What if we want to restrict the number of classes that can inherit from a particular class? The answer is sealed class. So, a sealed class is a technique that limits the number of classes that can inherit the given class. This means that only the classes designated by the programmer can inherit from that particular class, thereby restricting access to it. when a class is declared sealed, the programmer must specify the list of classes that can inherit it. The concept of sealed classes in Java was introduced in Java 15.

Steps to Create a Sealed Class

- Define the class that you want to make a seal.
- Add the "sealed" keyword to the class and specify which classes are permitted to inherit it by using the "permits" keyword.

Example

```
sealed class Human permits Manish, Vartika, Anjali
{
   public void printName()
   {
       System.out.println("Default");
```

```
   }
}
non-sealed class Manish extends Human
{
   public void printName()
   {
       System.out.println("Manish Sharma");
   }
}
sealed class Vartika extends Human
{
   public void printName()
   {
       System.out.println("Vartika Dadheech");
   }
}
final class Anjali extends Human
{
   public void printName()
   {


System.out.println("Anjali Sharma");
   }
}
```

Explanation of the above Example:

- *Human* is the parent class of *Manish*, *Vartika,* and *Anjali*. It is a sealed class so; other classes cannot inherit it.
- *Manish*, *Vartika*, and *Anjali* are child classes of the *Human* class, and it is necessary to make them either *sealed*, *non-sealed*, or *final*. Child classes of a sealed class must be sealed, non-sealed, or final.
- If any class other than *Manish*, *Vartika*, and *Anjali* tries to inherit from the *Human* class, it will cause a compiler error.

Lecture -24

Switch Expressions

Switch expressions were introduced as a preview feature in Java 12 and became a standard feature in Java 14. They provide a more concise and expressive way to use the switch statement,

allowing it to be used as either a statement or an expression. Here's a summary of the key points about switch expressions in Java:

Introduction to Switch Expressions:

Switch expressions in Java allow the switch statement to be used as an expression, resulting in a single value.

They can use either traditional case : labels with fall through or new case ... -> labels without fall through.

Syntax and Usage:

Switch expressions evaluate to a single value and can be used in statements.

They may contain case L -> labels that eliminate the need for break statements to prevent fall through.

The yield statement can be used to specify the value of a switch expression.

Example:

Here's an example of a switch expression in Java:

```
int numLetters = 0;

Day day = Day.WEDNESDAY;

numLetters = switch (day) {

    case MONDAY, FRIDAY, SUNDAY -> 6;

    case TUESDAY -> 7;

    case THURSDAY, SATURDAY -> 8;

    case WEDNESDAY -> 9;

    default -> throw new IllegalStateException("Invalid day: " + day);

};

System.out.println(numLetters);
```

Benefits:

Switch expressions provide a more concise and readable alternative to traditional switch statements, especially when the goal is to produce a result.

They eliminate the need for break statements and provide a more functional programming style.

Evolution of Switch Expressions:

Switch expressions were introduced as a preview feature in Java 12 and became a standard feature in Java 14.

The feature was designed to simplify everyday coding and prepare the way for the use of pattern matching

in switch statements.

Switch expressions in Java provide a more concise and expressive way to use the switch statement, allowing it to be used as either a statement or an expression. This feature simplifies everyday coding and prepares the way for the use of pattern matching in switch statements.

Yield Keyword

The yield keyword is used in various programming languages, including Python, JavaScript, C#, and Java, with slightly different functionalities in each language. Here's a summary of the yield keyword in different programming languages based on the provided search results:

in Java, the yield keyword is used within switch expressions to exit the switch and return a value that becomes the value of the switch expression. It allows the value of a switch expression to be assigned to a variable and provides an implicit check for covering all cases in the switch expression. The yield keyword is used in different programming languages to create generator functions, pause and resume generator functions, provide values in iterations, and exit switch expressions with a return value. Each language has its own specific use case and syntax for the yield keyword.

Lecture-25

Text Blocks

Text blocks in Java are a feature introduced in Java 13 as a preview and became a permanent feature in Java 15. They provide a more concise and readable way to declare multi-line strings in Java. Here's a summary

of the key points about text blocks in Java based on the provided search results:

Purpose of Text Blocks:

The principal purpose of text blocks is to provide clarity by minimizing the Java syntax required to render

a string that spans multiple lines.

Text blocks eliminate the need for explicit line terminators, string concatenations, and delimiters, allowing for the embedding of code snippets and text sequences more or less as-is.

Usage:

Text blocks start with """ followed by optional whitespaces and a newline. For example:

String example = """

Example text""";

Inside the text blocks, newlines and quotes can be used without the need for escaping line breaks. This allows for the inclusion of literal fragments of HTML, JSON, SQL, or other content in a more elegant and readable way.

Benefits:

Text blocks provide a more efficient and readable way to declare multi-line strings in Java, especially when dealing with large or complex text content.

They enhance code readability and maintainability by eliminating the need for explicit escape characters and concatenations.

Evolution of Text Blocks:

Text blocks were introduced as a preview feature in Java 13 and refined in a second preview before becoming a permanent feature in Java 15.

The feature was aimed at reducing the complexity of declaring and using multi-line string literals in Java.

Text blocks in Java provide a more efficient and readable way to declare multi-line strings, enhancing code readability and maintainability. They eliminate the need for explicit escape characters and concatenations, making it easier to include literal fragments of various content types in a more elegant and readable way.

Records

In Java, a record is a new type of class introduced in Java 14 that is designed to be a simple and concise way to declare classes that are transparent holders for shallowly immutable data. Here's a summary of the key points about records in Java:

Purpose of Records:

Records are designed to provide a compact syntax for declaring classes that are meant to be used primarily for storing data.

They are immutable by default, meaning that their state cannot be changed after they are constructed.

Syntax and Usage:

Records are declared using the record keyword followed by the name of the record and a list of components, which are the data fields of the record.

Records automatically provide implementations for methods such as equals(), hashCode(), and toString(), based on the components of the record.

Example:

Here's an example of a record declaration in Java:

```
public record Point(int x, int y) {}
```

In this example, the Point record declares two components, x and y, and automatically provides implementations for equals(), hashCode(), and toString().

Benefits:

Records provide a more concise and readable way to declare classes that are intended for data storage, reducing the amount of boilerplate code that needs to be written.

They promote immutability and are well-suited for use in functional programming and concurrent environments.

Evolution of Records:

Records were introduced as a preview feature in Java 14 and became a standard feature in Java 16.

The feature was aimed at simplifying the development of classes meant for data storage and promoting best practices for immutability and transparency.

Records in Java provide a concise and immutable way to declare classes that are primarily used for storing data. They automatically generate implementations for common methods, reducing the need for boilerplate code and promoting best practices for immutability and transparency.

Lecture-26

Sealed Classe

In Java, sealed classes were introduced as a preview feature in Java 15 and became a standard
feature in Java 17.
Sealed classes provide a way to control the inheritance hierarchy of a class or interface by specifying
Which classes can extend or implement it.
Here's a summary of the key points about sealed classes in Java based on the provided search results:

1. **Syntax and Usage**:
    - Sealed classes are declared using the `sealed` modifier in their declaration.
    - After any `extends` and `implements` clauses, the `permits` clause specifies the classes that are permitted to extend the sealed class or implement the sealed interface.
2. **Purpose and Benefits**:
    - Sealed classes provide a more fine-grained control over inheritance in Java, allowing classes and interfaces to define their permitted subtypes.
    - This feature is useful for domain modeling and increasing the security of libraries, as it enables the specification of which classes can implement or extend a particular class or interface.
3. **Evolution and Status**:
    - Sealed classes were proposed by JEP 360 and delivered in JDK 15 as a preview feature. They were refined and delivered in JDK 16 as a preview feature with no changes from JDK 16 to JDK 17.
    - With the release of Java 17, sealed classes have become a permanent feature, providing more control over the inheritance hierarchy.

Sealed classes in Java offer a mechanism for controlling class hierarchies, which helps prevent unauthorized extensions and provides a more secure and maintainable codebase. They enable fine-grained control over which classes can extend or implement a particular class or interface, enhancing the security and flexibility of Java applications.

Unit-4

Lecture Delivery Plan:

Lecture-27

27.1 Collections in java
27.2 Hierarchy of collection framework
27.3 Iterable, collector and list interface

Lecture-28

28.1 ArrayList

28.2 LinkedList

28.3 Vector

Lecture-29

29.1 Stack

29.2 Queue Interface

29.2.1 Priority Interface

29.2.2 Dqueue Interface

Lecture-30

30.1 Set Interface

30.2 HashSet

30.3 LinkedHashSet

Lecture-31

31.1 SortedSet

31.2 TreeSet

31.3 Map Interface

31.4 Tree Map

31.5 HashMap class

Lecture-32

32.1 HashTable class

32.2 Sorting

32.3 Comparable Interface

32.4 Comparator Interface

32.5 Properties class in java

Lecture-27

Collections in Java

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

What is Collection in Java

A Collection represents a single unit of objects, i.e., a group.

What is a framework in Java

- It provides readymade architecture.
- It represents a set of classes and interfaces.
- It is optional.

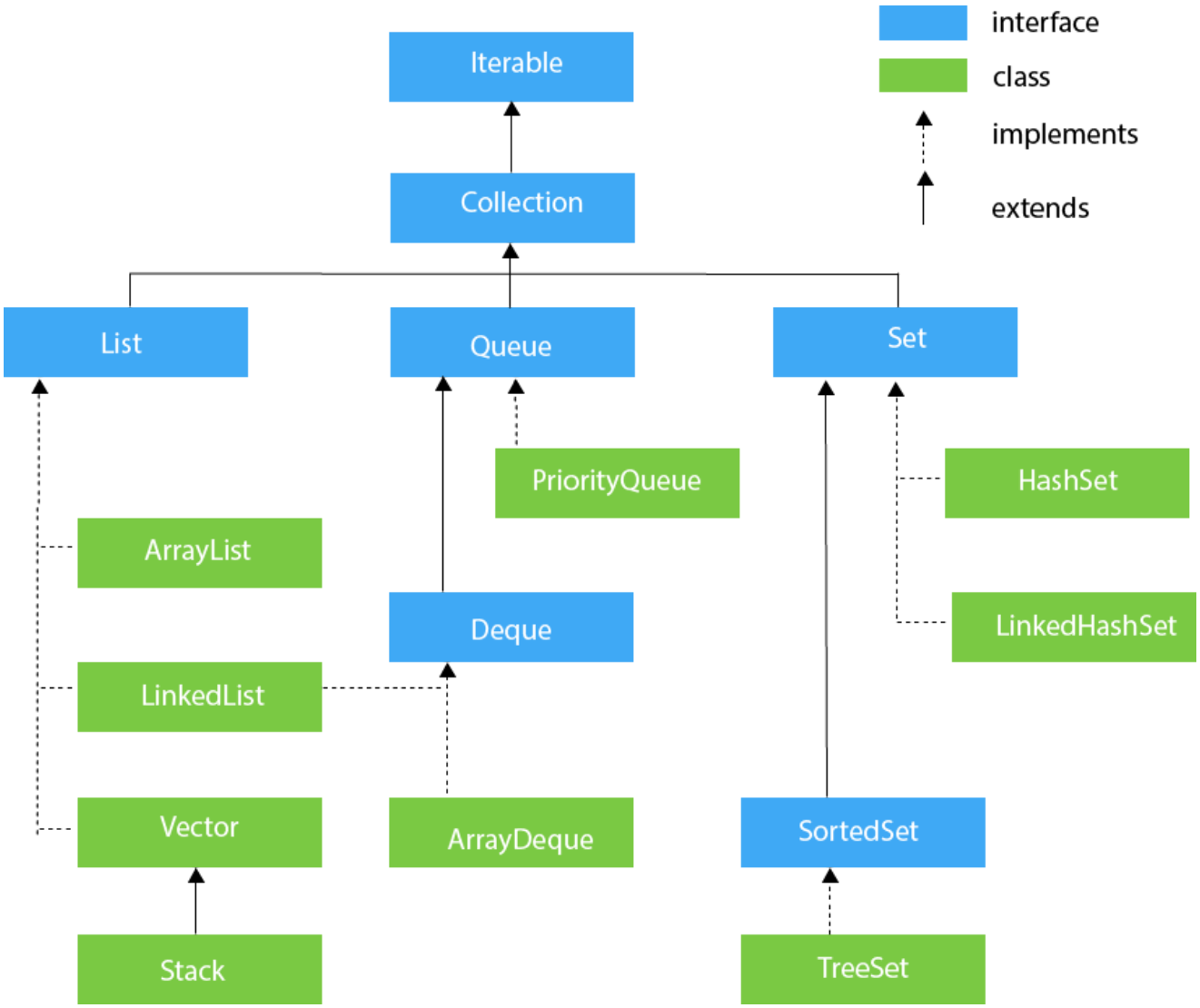
What is Collection framework

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes
2. Algorithm

Hierarchy of Collection Framework

Let us see the hierarchy of Collection framework. The **java.util** package contains all the classes and interfaces for the Collection framework.



Iterable Interface

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method. i.e.,

1. Iterator<T> iterator()

It returns the iterator over the elements of type T.

Collection Interface

The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other

words, we can say that the Collection interface builds the foundation on which the collection framework depends.

Some of the methods of Collection interface are Boolean add (Object obj), Boolean addAll (Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

```
1. List <data-type> list1= new ArrayList();
2. List <data-type> list2 = new LinkedList();
3. List <data-type> list3 = new Vector();
4. List <data-type> list4 = new Stack();
```

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

The classes that implement the List interface are given below.

Lecture-28

The classes that implement the List interface are given below.

ArrayList

The ArrayList class is a resizable array, which can be found in the java.util package. The difference between a built-in array and an ArrayList in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one).

How the ArrayList works

The `ArrayList` class has a regular array inside it. When an element is added, it is placed into the array. If the array is not big enough, a new, larger array is created to replace the old one and the old one is removed.

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

```
1. import java.util.*;
2. class TestJavaCollection1{
3. public static void main(String args[]){
4. ArrayList<String> list=new ArrayList<String>();//Creating arraylist
5. list.add("Ravi");//Adding object in arraylist
6. list.add("Vijay");
7. list.add("Ravi");
8. list.add("Ajay");
9. //Traversing list through Iterator
10. Iterator itr=list.iterator();
11. while(itr.hasNext()){
12. System.out.println(itr.next());
13. }
14. }
15. }
```

Output:

```
Ravi
Vijay
Ravi
Ajay
```

LinkedList

The `LinkedList` class is a collection which can contain many objects of the same type, just like the `ArrayList`.

The `LinkedList` class has all of the same methods as the `ArrayList` class because they both implement the `List` interface. This means that you can add items, change items, remove items and clear the list in the same way.

However, while the `ArrayList` class and the `LinkedList` class can be used in the same way, they are built very differently.

How the LinkedList works

The `LinkedList` stores its items in "containers." The list has a link to the first container and each container has a link to the next container in the list. To add an element to the list, the element is placed into a new container and that container is linked to one of the other containers in the list.

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

Consider the following example.

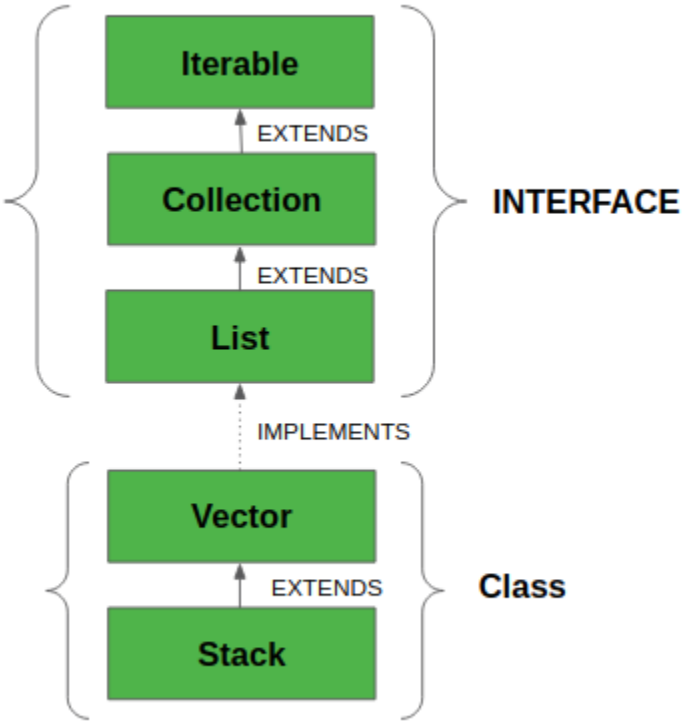
```
1. import java.util.*;
2. public class TestJavaCollection2{
3. public static void main(String args[]){
4. LinkedList<String> al=new LinkedList<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ravi");
8. al.add("Ajay");
9. Iterator<String> itr=al.iterator();
10. while(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }
```

Output:

```
Ravi
Vijay
Ravi
Ajay
```

Vector

The Vector class implements a growable array of objects. Vectors fall in legacy classes, but now it is fully compatible with collections. It is found in java.util package and implement the List interface, so we can use all the methods of the List interface as shown below as follows:



- Vector implements a dynamic array which means it can grow or shrink as required. Like an array, it contains components that can be accessed using an integer index.
- They are very similar to ArrayList, but Vector is synchronized and has some legacy methods that the collection framework does not contain.
- It also maintains an insertion order like an ArrayList. Still, it is rarely used in a non-thread environment as it is **synchronized**, and due to this, it gives a poor performance in adding, searching, deleting, and updating its elements.
- The Iterators returned by the Vector class are fail-fast. In the case of concurrent modification, it fails and throws the **ConcurrentModificationException.**

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection3{
3. public static void main(String args[]){
4. Vector<String> v=new Vector<String>();
5. v.add("Ayush");
6. v.add("Amit");
7. v.add("Ashish");
8. v.add("Garima");
9. Iterator<String> itr=v.iterator();
10. while(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }
```

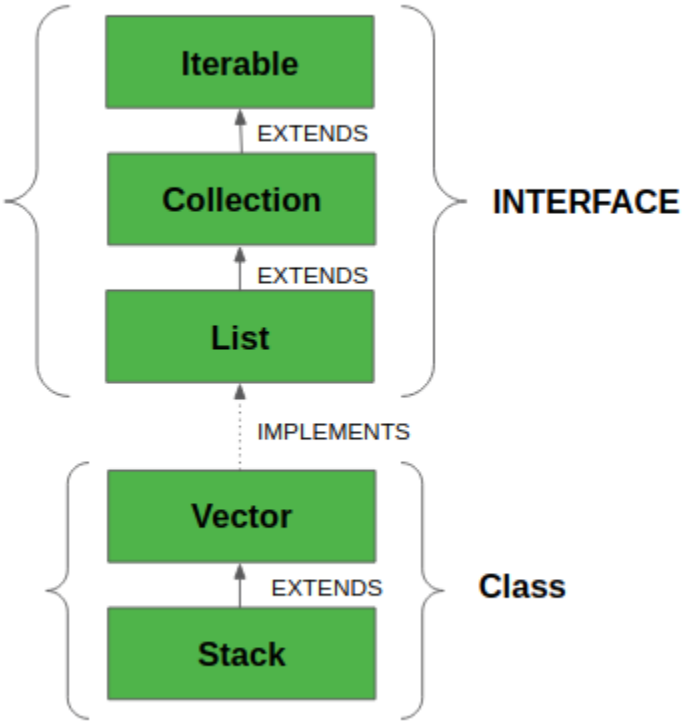
Output:

```
Ayush
Amit
Ashish
Garima
```

Lecture-29

Stack

Java Collection framework provides a Stack class that models and implements a **Stack data structure**. The class is based on the basic principle of last-in-first-out. In addition to the basic push and pop operations, the class provides three more functions of empty, search, and peek. The class can also be said to extend Vector and treats the class as a stack with the five mentioned functions. The class can also be referred to as the subclass of Vector.
The below diagram shows the **hierarchy of the Stack class**:



The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection4{
3. public static void main(String args[]){
4. Stack<String> stack = new Stack<String>();
5. stack.push("Ayush");
6. stack.push("Garvit");
```

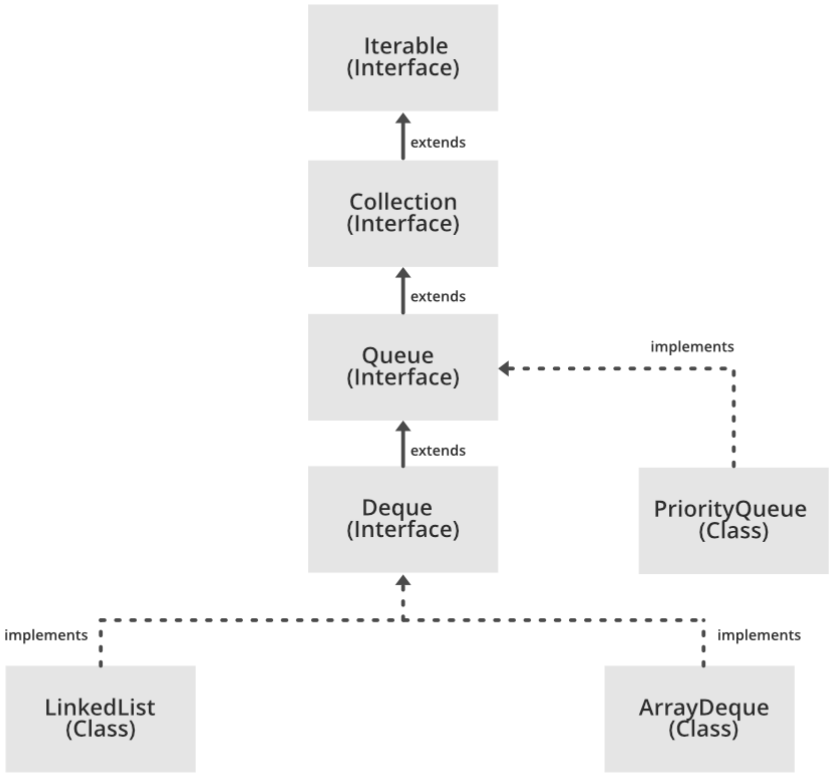
```
7.  stack.push("Amit");
8.  stack.push("Ashish");
9.  stack.push("Garima");
10. stack.pop();
11. Iterator<String> itr=stack.iterator();
12. while(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }
```

Output:

```
Ayush
Garvit
Amit
Ashish
```

Queue Interface

The Queue interface is present in java.util package and extends the Collection interface is used to hold the elements about to be processed in FIFO(First In First Out) order. It is an ordered list of objects with its use limited to inserting elements at the end of the list and deleting elements from the start of the list, (i.e.), it follows the **FIFO** or the First-In-First-Out principle.



Being an interface the queue needs a concrete class for the declaration and the most common classes are the PriorityQueue and LinkedList in Java. Note that neither of these implementations is thread-safe. PriorityBlockingQueue is one alternative implementation if the thread-safe implementation is needed.

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated as:

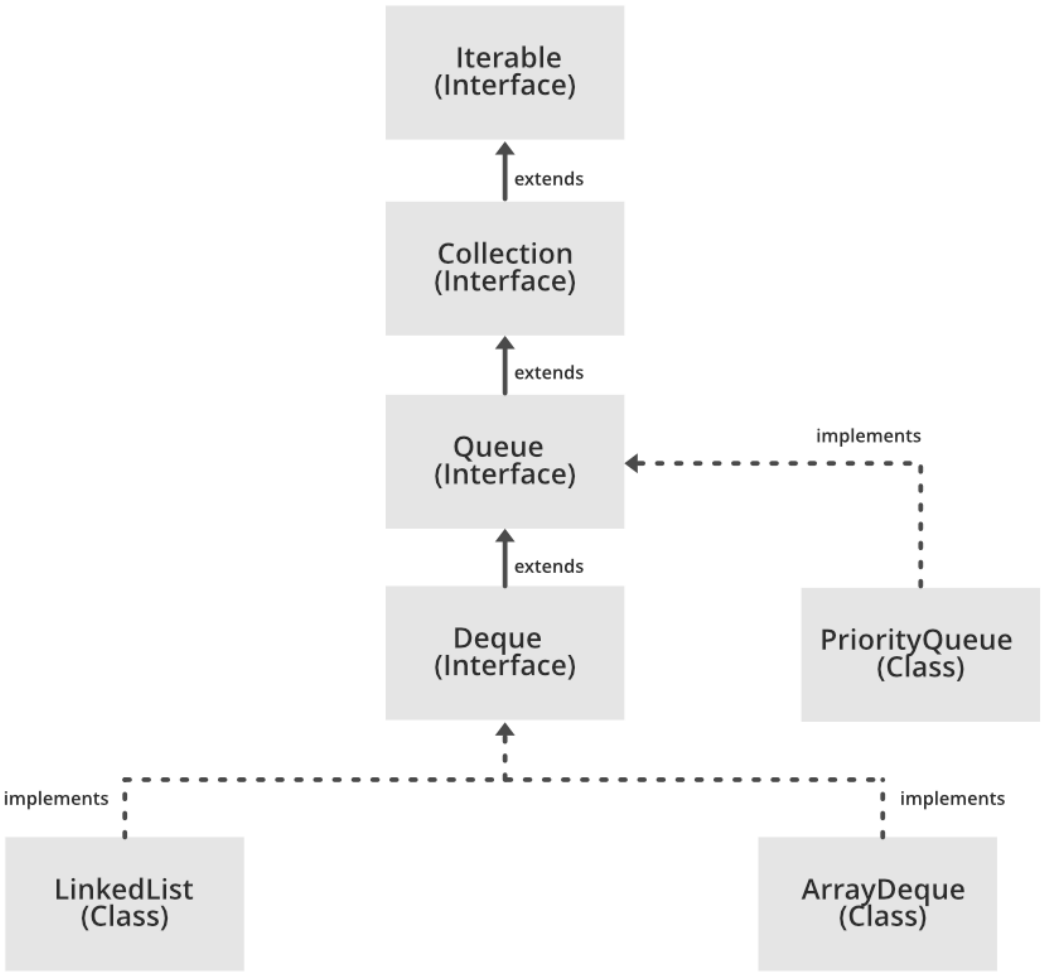
```
1. Queue<String> q1 = new PriorityQueue();
2. Queue<String> q2 = new ArrayDeque();
```

There are various classes that implement the Queue interface, some of them are given below.

PriorityQueue

A PriorityQueue is used when the objects are supposed to be processed based on the priority. It is known that a Queue follows the First-In-First-Out algorithm, but sometimes

the elements of the queue are needed to be processed according to the priority, that's when the PriorityQueue comes into play.
The PriorityQueue is based on the priority heap. The elements of the priority queue are ordered according to the natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.



In the below priority queue, an element with a maximum ASCII value will have the highest priority.

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection5{
3. public static void main(String args[]){
4. PriorityQueue<String> queue=new PriorityQueue<String>();
5. queue.add("Amit Sharma");
```

```
6. queue.add("Vijay Raj");
7. queue.add("JaiShankar");
8. queue.add("Raj");
9. System.out.println("head:"+queue.element());
10. System.out.println("head:"+queue.peek());
11. System.out.println("iterating the queue elements:");
12. Iterator itr=queue.iterator();
13. while(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. queue.remove();
17. queue.poll();
18. System.out.println("after removing two elements:");
19. Iterator<String> itr2=queue.iterator();
20. while(itr2.hasNext()){
21. System.out.println(itr2.next());
22. }
23. }
24. }
```

Output:

```
head:Amit Sharma
head:Amit Sharma
iterating the queue elements:
Amit Sharma
Raj
JaiShankar
Vijay Raj
after removing two elements:
Raj
Vijay Raj
```

Deque Interface

The Deque (double-ended queue) interface in Java is a subinterface of the Queue interface and extends it to provide a double-ended queue, which is a queue that allows elements to be added and removed from both ends. The Deque interface is part of the Java Collections Framework and is used to provide a generic and flexible data structure that can be used to implement a variety of algorithms and data structures.

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

Deque can be instantiated as:

```
1. Deque d = new ArrayDeque();
```

ArrayDeque

The ArrayDeque class in Java is an implementation of the Deque interface that uses a resizable array to store its elements. This class provides a more efficient alternative to the traditional Stack class, which was previously used for double-ended operations. The ArrayDeque class provides constant-time performance for inserting and removing elements from both ends of the queue, making it a good choice for scenarios where you need to perform many add and remove operations.

ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection6{
3. public static void main(String[] args) {
4. //Creating Deque and adding elements
5. Deque<String> deque = new ArrayDeque<String>();
6. deque.add("Gautam");
7. deque.add("Karan");
8. deque.add("Ajay");
9. //Traversing elements
10. for (String str : deque) {
11. System.out.println(str);
12. }
13. }
14. }
```

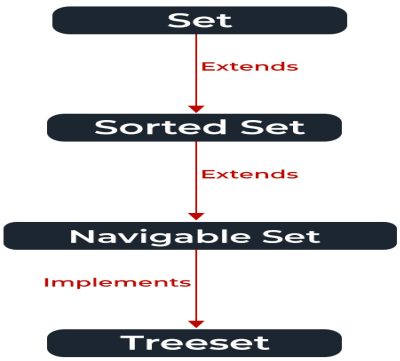
Output:

```
Gautam
Karan
Ajay
```

Lecture-30

Set Interface

The set interface is present in java.util package and extends the Collection interface. It is an unordered collection of objects in which duplicate values cannot be stored. It is an interface that implements the mathematical set. This interface contains the methods inherited from the Collection interface and adds a feature that restricts the insertion of the duplicate elements. There are two interfaces that extend the set implementation



namely SortedSet and NavigableSet.
In the above image, the navigable set extends the sorted set interface. Since a set doesn't retain the insertion order, the navigable set interface provides the implementation to navigate through the Set. The class which implements the navigable set is a TreeSet which is an implementation of a self-balancing tree. Therefore, this interface provides us with a way to navigate through this tree.

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

1. Set<data-type> s1 = new HashSet<data-type>();
2. Set<data-type> s2 = new LinkedHashSet<data-type>();
   Set<data-type> s3 = new TreeSet<data-type>();

HashSet

Java HashSet class implements the Set interface, backed by a hash table which is actually a HashMap instance. No guarantee is made as to the iteration order of the hash sets which means that the class does not guarantee the constant order of elements over time. This class permits the null element. The class also offers constant time performance for the basic operations like add, remove, contains, and size assuming the hash function disperses the elements properly among the buckets, which we shall see further in the article.

Java HashSet Features

A few important features of HashSet are mentioned below:

- Implements Set Interface.
- The underlying data structure for HashSet is Hashtable.
- As it implements the Set Interface, duplicate values are not allowed.
- Objects that you insert in HashSet are not guaranteed to be inserted in the same order. Objects are inserted based on their hash code.
- NULL elements are allowed in HashSet.
- HashSet also implements **Serializable** and **Cloneable** interfaces.

Declaration of HashSet

```
public class HashSet<E> extends AbstractSet<E> implements Set<E>,
Cloneable, Serializable
```

where **E** is the type of elements stored in a HashSet.

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection7{
3. public static void main(String args[]){
4. //Creating HashSet and adding elements
5. HashSet<String> set=new HashSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //Traversing elements
```

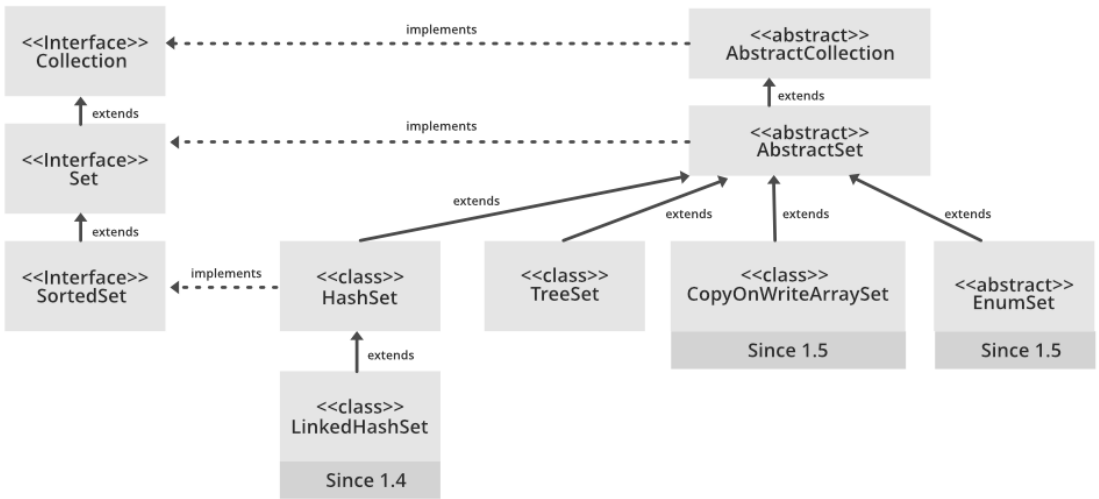
```
11. Iterator<String> itr=set.iterator();
12. while(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }
```

Output:

```
Vijay
Ravi
Ajay
```

LinkedHashSet

The **LinkedHashSet** is an ordered version of HashSet that maintains a doubly-linked List across all elements. When the iteration order is needed to be maintained this class is used. When iterating through a HashSet the order is unpredictable, while a LinkedHashSet lets us iterate through the elements in the order in which they were inserted. When cycling through LinkedHashSet using an iterator, the elements will be returned in the order in which they were inserted.
The Hierarchy of LinkedHashSet is as follows:



Parameters: The type of elements maintained by this set

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection8{
3. public static void main(String args[]){
4. LinkedHashSet<String> set=new LinkedHashSet<String>();
5. set.add("Ravi");
6. set.add("Vijay");
7. set.add("Ravi");
8. set.add("Ajay");
9. Iterator<String> itr=set.iterator();
10. while(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }
```

Output:

```
Ravi
Vijay
```

SortedSet Interface

The SortedSet interface present in java.util package extends the Set interface present in the collection framework. It is an interface that implements the mathematical set. This interface contains the methods inherited from the Set interface and adds a feature that stores all the elements in this interface to be stored in a sorted manner.

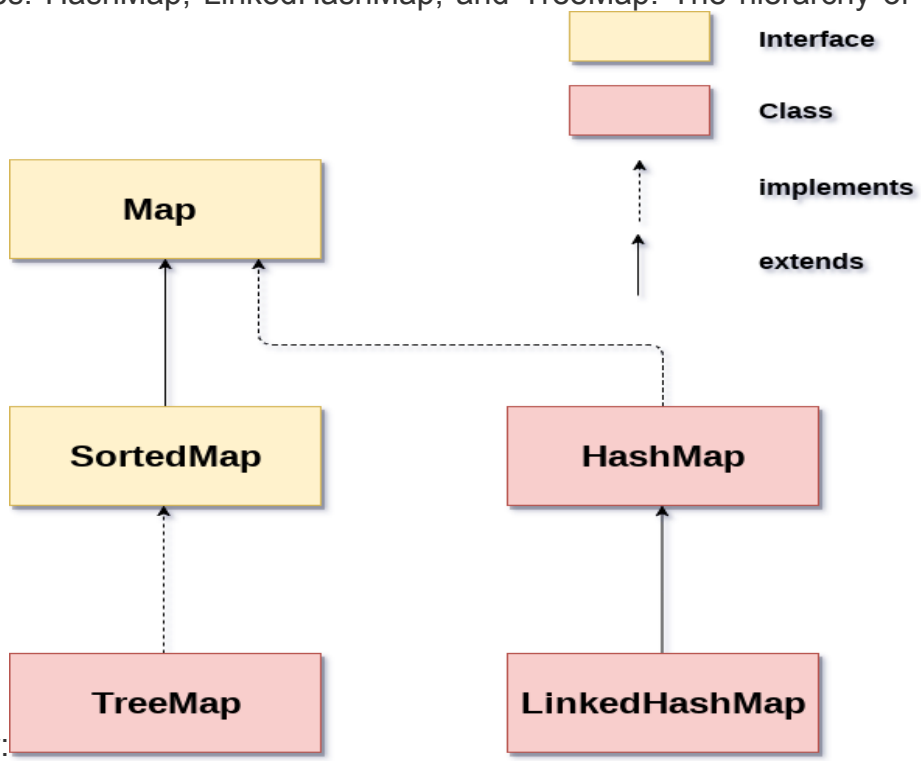
Java Map Interface

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.

A Map is useful if you have to search, update or delete elements on the basis of a key.

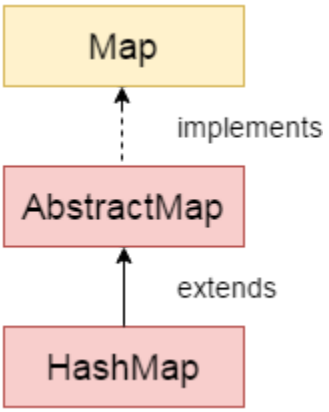
Java Map Hierarchy

There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap. The hierarchy of Java Map is given below:



A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.

Java HashMap



Java **HashMap** class implements the Map interface which allows us *to store key and value pair*, where keys should be unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the java.util package.

HashMap in Java is like the legacy Hashtable class, but it is not synchronized. It allows us to store the null elements as well, but there should be only one null key. Since Java 5, it is denoted as HashMap<K,V>, where K stands for key and V for value. It inherits the AbstractMap class and implements the Map interface.

Points to remember

- Java HashMap contains values based on the key.
- Java HashMap contains only unique keys.
- Java HashMap may have one null key and multiple null values.
- Java HashMap is non synchronized.
- Java HashMap maintains no order.
- The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order. The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

The SortedSet can be instantiated as:

```
1. SortedSet<data-type> set = new TreeSet();
```

TreeSet

TreeSet is one of the most important implementations of the SortedSet interface in Java that uses a Tree for storage. The ordering of the elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided. This must be consistent with equals if it is to correctly implement the Set interface.

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

Consider the following example:

```
1. import java.util.*;
2. public class TestJavaCollection9{
3. public static void main(String args[]){
4. //Creating and adding elements
5. TreeSet<String> set=new TreeSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //traversing elements
11. Iterator<String> itr=set.iterator();
12. while(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }
```

Output:

```
Ajay
Ravi
Vijay
```

LinkedHashMap in Java

The **LinkedHashMap Class** is just like HashMap with an additional feature of maintaining an order of elements inserted into it. HashMap provided the advantage of quick insertion, search, and deletion but it never maintained the track and order of insertion, which the LinkedHashMap provides where the elements can be accessed in their insertion order.

Important Features of a LinkedHashMap are listed as follows:

- A LinkedHashMap contains values based on the key. It implements the Map interface and extends the HashMap class.
- It contains only unique elements.
- It may have one null key and multiple null values.
- It is non-synchronized.
- It is the same as HashMap with an additional feature that it maintains insertion order. For example, when we run the code with a HashMap, we get a different order of elements.

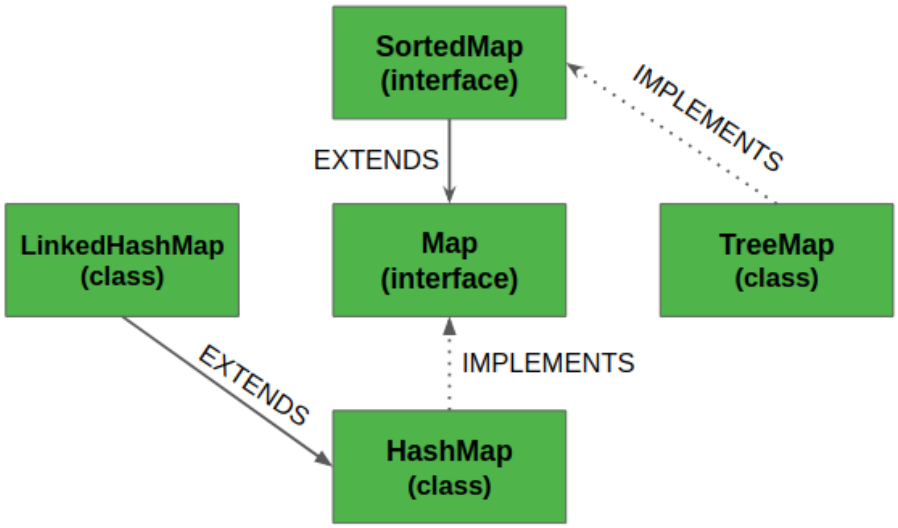
Declaration:

```
public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V>
```

Here, **K** is the key Object type and **V** is the value Object type

- **K** – The type of the keys in the map.
- **V** – The type of values mapped in the map.

It implements Map<K, V> interface, and extends HashMap<K, V> class. Though the Hierarchy of LinkedHashMap is as depicted in below media as follows**:**



MAP Hierarchy in Java

How LinkedHashMap Work Internally?

A LinkedHashMap is an extension of the **HashMap** class and it implements the **Map** interface. Therefore, the class is declared as:

```
public class LinkedHashMap
extends HashMap
implements Map
```

In this class, the data is stored in the form of nodes. The implementation of the LinkedHashMap is very similar to a doubly-linked list. Therefore, each node of the LinkedHashMap is represented as:

- **Hash:** All the input keys are converted into a hash which is a shorter form of the key so that the search and insertion are faster.
- **Key:** Since this class extends HashMap, the data is stored in the form of a key-value pair. Therefore, this parameter is the key to the data.
- **Value:** For every key, there is a value associated with it. This parameter stores the value of the keys. Due to generics, this value can be of any form.
- **Next:** Since the LinkedHashMap stores the insertion order, this contains the address to the next node of the LinkedHashMap.
- **Previous:** This parameter contains the address to the previous node of the LinkedHashMap.

Lecture-32

Hash table in Java

The **Hashtable** class implements a hash table, which maps keys to values. Any non-null object can be used as a key or as a value. To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method.
The java.util.Hashtable class is a class in Java that provides a key-value data structure, similar to the Map interface. It was part of the original Java Collections framework and was introduced in Java 1.0.

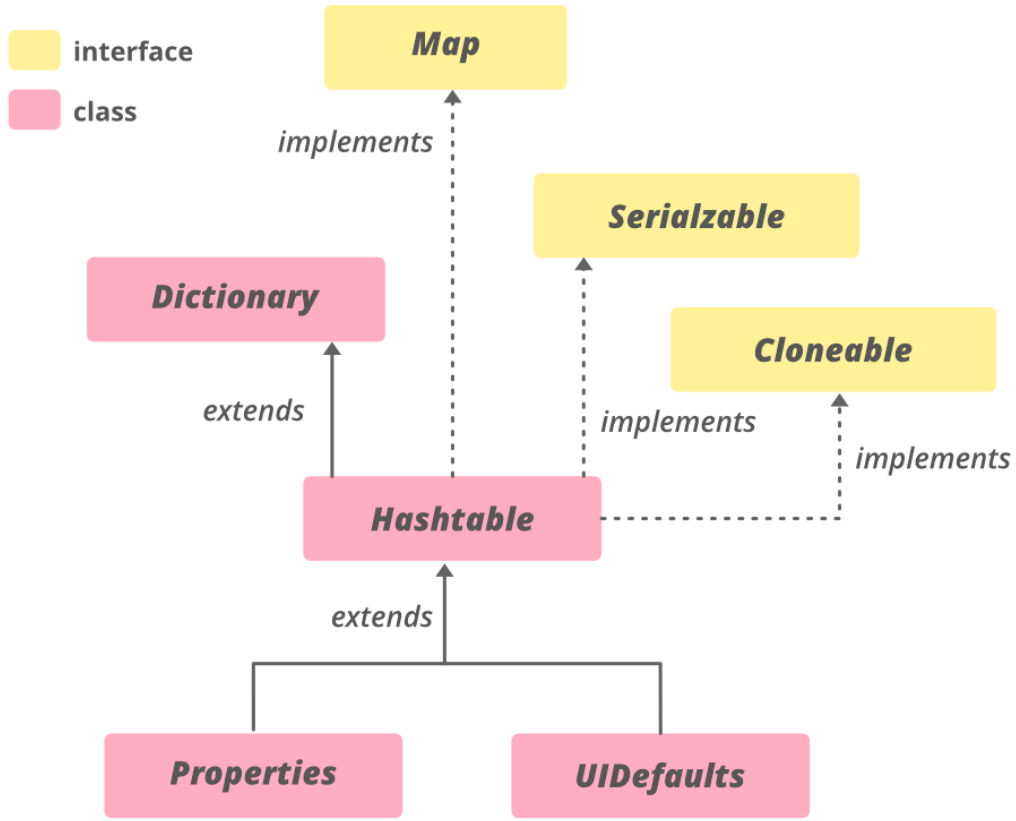
However, the Hashtable class has since been considered obsolete and its use is generally discouraged. This is because it was designed prior to the introduction of the Collections framework and does not implement the Map interface, which makes it difficult to use in conjunction with other parts of the framework. In addition, the Hashtable class is synchronized, which can result in slower performance compared to other implementations of the Map interface.

In general, it's recommended to use the Map interface or one of its implementations (such as HashMap or ConcurrentHashMap) instead of the Hashtable class.

Features of Hashtable

- It is similar to HashMap, but is synchronized.
- Hashtable stores key/value pair in hash table.
- In Hashtable we specify an object that is used as a key, and the value we want to associate to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.
- The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.
- HashMap doesn't provide any Enumeration, while Hashtable provides not fail-fast Enumeration.

The Hierarchy of Hashtable



Sorting in Java

Whenever we do hear sorting algorithms come into play such as selection sort, bubble sort, insertion sort, radix sort, bucket sort, etc but if we look closer here we are not asked to use any kind of algorithms. It is as simple sorting with the help of linear and non-linear data structures present within java. So there is sorting done with the help of brute force in java with the help of loops and there are two in-built methods to sort in Java.

Ways of sorting in Java

1. Using loops
1. Using sort() method of Arrays class
1. Using sort method of Collections class
1. Sorting on a subarray

Let us discuss all four of them and propose a code for each one of them.

Java Comparable interface

Java Comparable interface is used to order the objects of the user-defined class. This interface is found in java.lang package and contains only one method named compareTo(Object). It provides a single sorting sequence only, i.e., you can sort the

elements on the basis of single data member only. For example, it may be rollno, name, age or anything else.

compareTo(Object obj) method

public int compareTo(Object obj): It is used to compare the current object with the specified object. It returns

- positive integer, if the current object is greater than the specified object.
- negative integer, if the current object is less than the specified object.
- zero, if the current object is equal to the specified object.

We can sort the elements of:

1. String objects
2. Wrapper class objects
3. User-defined class objects

Collections class

Collections class provides static methods for sorting the elements of collections. If collection elements are of Set or Map, we can use TreeSet or TreeMap. However, we cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements.

Method of Collections class for sorting List elements

public void sort(List list): It is used to sort the elements of List. List elements must be of the Comparable type.

> *Note: String class and Wrapper classes implement the Comparable interface by default. So if you store the objects of string or wrapper classes in a list, set or map, it will be Comparable by default.*

Comparator Interface in Java with Examples

- A comparator interface is used to order the objects of user-defined classes. A comparator object is capable of comparing two objects of the same class**.** Following function compare obj1 with obj2.
Syntax:

```
public int compare(Object obj1, Object obj2):
```

Suppose we have an Array/ArrayList of our own class type, containing fields like roll no, name, address, DOB, etc, and we need to sort the array based on Roll no or name?

Method 1: One obvious approach is to write our own sort() function using one of the standard algorithms. This solution requires rewriting the whole sorting code for different criteria like Roll No. and Name.

Method 2: Using comparator interface- Comparator interface is used to order the objects of a user-defined class. This interface is present in java.util package and contains 2 methods compare(Object obj1, Object obj2) and equals(Object element). Using a comparator, we can sort the elements based on data members. For instance, it may be on roll no, name, age, or anything else.

Method of Collections class for sorting List elements is used to sort the elements of List by the given comparator.

```
public void sort(List list, ComparatorClass c)
```

To sort a given List, ComparatorClass must implement a Comparator interface.

How do the sort() method of Collections class work?

Internally the Sort method does call Compare method of the classes it is sorting. To compare two elements, it asks “Which is greater?” Compare method returns -1, 0, or 1 to say if it is less than, equal, or greater to the other. It uses this result to then determine if they should be swapped for their sort.

Properties Class in Java

-

The Properties class represents a persistent set of properties. The Properties can be saved to a stream or loaded from a stream. It belongs to **java.util** package. **Properties** define the following instance variable. This variable holds a default property list associated with a **Properties** object.
Properties defaults: *This variable holds a default property list associated with a Properties object.*

Features of Properties class:

- **Properties** is a subclass of Hashtable.
- It is used to maintain a list of values in which the key is a string and the value is also a string i.e; it can be used to store and retrieve string type data from the properties file.

- Properties class can specify other properties list as it's the default. If a particular key property is not present in the original Properties list, the default properties will be searched.
- Properties object does not require external synchronization and Multiple threads can share a single Properties object.
- Also, it can be used to retrieve the properties of the system.

Advantage of a Properties file

In the event that any data is changed from the properties record, you don't have to recompile the java class. It is utilized to store data that is to be changed habitually.

Note: The Properties class does not inherit the concept of a load factor from its superclass, **Hashtable**.

Declaration

```
public class Properties extends Hashtable<Object, Object>
```

Constructors of Properties

1. Properties(): This creates a **Properties** object that has no default values.
Properties p = new Properties();

2. Properties(Properties propDefault): The second creates an object that uses **propDefault** for its default value.
Properties p = new Properties(Properties propDefault);

UNIT-5

Lecture Delivery Plan:

Lecture-33

33.1 Spring Core Basics

33.2 Spring Dependency Injection concepts

33.3 Spring Inversion of control

Lecture-34

34.1 AOP

34.2 Bean Scopes-Singleton, Prototype, Request,Session,

Application and Web socket

Lecture-35

35.1 Autowiring

35.2 Annotations

35.3 Life cycle call backs

35.4 Bean configuration Styles

Lecture-36

36.1 Spring Boot build Systems

36.2 Spring Boot code Structure

36.3 Spring Boot Runners

4.4 Logger

Lecture-37

38.1 Building Restful Web Services

38.2 Rest Controller

Lecture-38

38.1 Get, Put, Post, Delete APIs

Lecture-39

39.1 Request Mapping

39.2 Request Body

39.3 Path Variable

39.4 Request Parameter

Lecture-40

40.1 Build web application

LECTURE-33

Java Framework is the body or platform of pre-written codes used by Java developers to develop Java applications or web applications. In other words, **Java Framework** is a collection of predefined classes and functions that is used to process input, manage

hardware devices interacts with system software. It acts like a skeleton that helps the developer to develop an application by writing their own code.

It was **developed by Rod Johnson in 2003**. Spring framework makes the easy development of JavaEE application.

It is helpful for beginners and experienced persons.

Spring Framework

Spring is a *lightweight* framework. It can be thought of as a *framework of frameworks* because it provides support to various frameworks such as Struts, Hibernate, Tapestry, EJB, JSF, etc. The framework, in broader sense, can be defined as a structure where we find solution of the various technical problems.

The Spring framework comprises several modules such as IOC, AOP, DAO, Context, ORM, WEB MVC etc. We will learn these modules in next page. Let's understand the IOC and Dependency Injection first.

Advantages of Spring Framework

There are many advantages of Spring Framework. They are as follows:

1) Predefined Templates

Spring framework provides templates for JDBC, Hibernate, JPA etc. technologies. So there is no need to write too much code. It hides the basic steps of these technologies.

Let's take the example of JdbcTemplate, you don't need to write the code for exception handling, creating connection, creating statement, committing transaction, closing connection etc. You need to write the code of executing query only. Thus, it save a lot of JDBC code.

2) Loose Coupling

The Spring applications are loosely coupled because of dependency injection.

3) Easy to test

The Dependency Injection makes easier to test the application. The EJB or Struts application require server to run the application but Spring framework doesn't require server.

4) Lightweight

Spring framework is lightweight because of its POJO implementation. The Spring Framework doesn't force the programmer to inherit any class or implement any interface. That is why it is said non-invasive.

5) Fast Development

The Dependency Injection feature of Spring Framework and it support to various frameworks makes the easy development of JavaEE application.

6) Powerful abstraction

It provides powerful abstraction to JavaEE specifications such as JMS, JDBC, JPA and JTA.

7) Declarative support

It provides declarative support for caching, validation, transactions and formatting.

Spring - Dependency Injection

Every Java-based application has a few objects that work together to present what the end-user sees as a working application. When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while unit testing. Dependency Injection (or sometime called wiring) helps in gluing these classes together and at the same time keeping them independent.

Consider you have an application which has a text editor component and you want to provide a spell check. Your standard code would look something like this −

```
public class TextEditor {
   private SpellChecker spellChecker;

   public TextEditor() {
      spellChecker = new SpellChecker();
   }
```

What we've done here is, create a dependency between the TextEditor and the SpellChecker. In an inversion of control scenario, we would instead do something like this −

```
public class TextEditor {
   private SpellChecker spellChecker;

   public TextEditor(SpellChecker spellChecker) {
      this.spellChecker = spellChecker;
   }
```

```
}
```

Here, the TextEditor should not worry about SpellChecker implementation. The SpellChecker will be implemented independently and will be provided to the TextEditor at the time of TextEditor instantiation. This entire procedure is controlled by the Spring Framework.

Here, we have removed total control from the TextEditor and kept it somewhere else (i.e. XML configuration file) and the dependency (i.e. class SpellChecker) is being injected into the class TextEditor through a **Class Constructor**. Thus the flow of control has been "inverted" by Dependency Injection (DI) because you have effectively delegated dependances to some external system.

The second method of injecting dependency is through **Setter Methods** of the TextEditor class where we will create a SpellChecker instance. This instance will be used to call setter methods to initialize TextEditor's properties.

Dependency Injection is the main functionality provided by Spring IOC(Inversion of Control). The Spring-Core module is responsible for injecting dependencies through either Constructor or Setter methods. The design principle of Inversion of Control emphasizes keeping the Java classes independent of each other and the container frees them from object creation and maintenance. These classes, managed by Spring, must adhere to the standard definition of Java-Bean. Dependency Injection in Spring also ensures loose coupling between the classes. There are two types of Spring Dependency Injection.

1. **Constructor-based dependency injection**
   Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on the other class.

2. **Setter-based dependency injection**
   Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

You can mix both, Constructor-based and Setter-based DI but it is a good rule of thumb to use constructor arguments for mandatory dependencies and setters for optional dependencies.

The code is cleaner with the DI principle and decoupling is more effective when objects are provided with their dependencies. The object does not look up its dependencies and does not know the location or class of the dependencies, rather everything is taken care by the Spring Framework.

Spring IoC (Inversion of Control)

Inversion of Control is a principle in software engineering which transfers the control of objects or portions of a program to a container or framework. We most often use it in the context of object-oriented programming.

Spring IoC (Inversion of Control) Container is the core of Spring Framework. It creates the objects, configures and assembles their dependencies, manages their entire life cycle. The Container uses Dependency Injection(DI) to manage the components that make up the application. It gets the information about the objects from a configuration file(XML) or Java Code or Java Annotations and Java POJO class. These objects are called Beans. Since the Controlling of Java objects and their lifecycle is not done by the developers, hence the name Inversion Of Control. The followings are some of the main features of Spring IoC,

- Creating Object for us,
- Managing our objects,
- Helping our application to be configurable,
- Managing dependencies

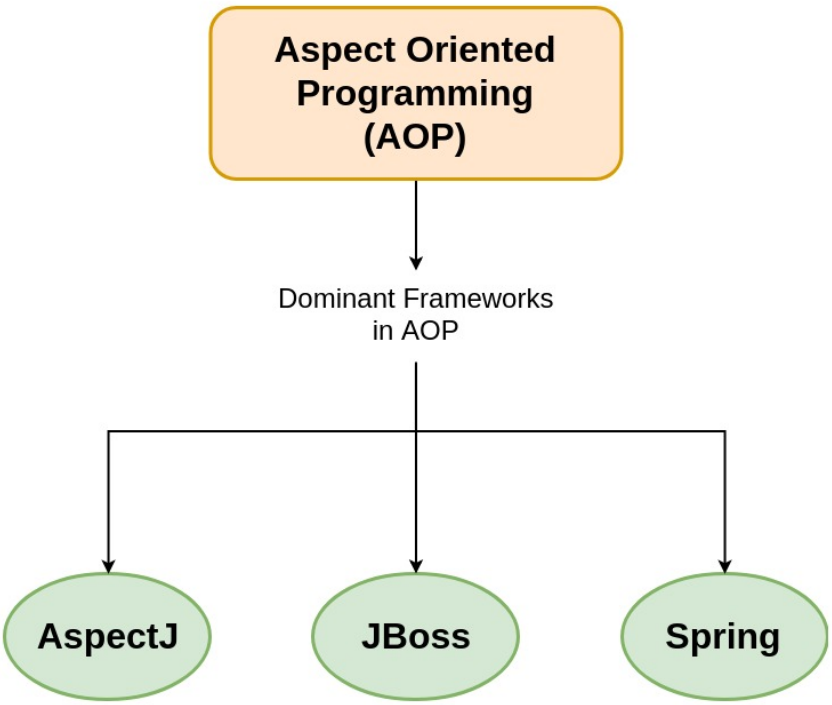
LECTURE-34

AOP in Spring Framework

Aspect oriented programming(AOP) as the name suggests uses aspects in programming. It can be defined as the breaking of code into different modules, also known as modularisation, where the aspect is the key unit of modularity. Aspects enable the implementation of crosscutting concerns such as- transaction, logging not central to business logic without cluttering the code core to its functionality. It does so by adding additional behaviour that is the advice to the existing code. For example- Security is a crosscutting concern, in many methods in an application security rules can be applied, therefore repeating the code at every method, define the functionality in a common class and control were to apply that functionality in the whole application.

Dominant Frameworks in AOP:

AOP includes programming methods and frameworks on which modularisation of code is supported and implemented. Let's have a look at



the three **dominant frameworks in AOP**:

- **AspectJ:** It is an extension for Java programming created at **PARC research centre**. It uses Java like syntax and included IDE integrations for displaying crosscutting structure. It has its own compiler and weaver, on using it enables the use of full AspectJ language.
- **JBoss:** It is an open source Java application server developed by JBoss, used for Java development.
- Spring**:** It uses XML based configuration for implementing AOP, also it uses annotations which are interpreted by using a library supplied by AspectJ for parsing and matching.

Currently, **AspectJ libraries with Spring framework** are dominant in the market, therefore let's have an understanding of how Aspect-oriented programming works with Spring.

How Aspect-Oriented Programming works with Spring:
One may think that invoking a method will automatically implement cross-cutting concerns but that is not the case. Just invocation of the method does not invoke the advice(the job which is meant to be done). Spring uses **proxy based mechanism** i.e. it creates a proxy Object which will wrap around the original object and will take up the advice which is relevant to the method call. Proxy objects can be created either manually through proxy factory bean or through auto proxy configuration in the XML file and get destroyed when the execution completes. Proxy objects are used to enrich the Original behaviour of the real object.

A **cross-cutting concern** is a concern that can affect the whole application and should be centralized in one location in code as possible, such as transaction management, authentication, logging, security etc.

Common terminologies in AOP:

1. **Aspect:** The class which implements the JEE application cross-cutting concerns(transaction, logger etc) is known as the aspect. It can be normal class configured through XML configuration or through regular classes annotated with @Aspect.
2. **Weaving:** The process of linking Aspects with an Advised Object. It can be done at load time, compile time or at runtime time. Spring AOP does weaving at runtime.
   Let's write our first aspect class but before that have a look at the jars required and the Bean configuration file for AOP.

BEAN SCOPE

Bean Definition*: In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.*

Bean Scopes refers to the lifecycle of Bean that means when the object of Bean will be instantiated, how long does that object live, and how many objects will be created for that bean throughout. Basically, it controls the instance creation of the bean and it is managed by the spring container.

The following are the different scopes provided for a bean:

1. **Singleton:** Only one instance will be created for a single bean definition per Spring IoC container and the same object will be shared for each request made for that bean.
2. **Prototype:** A new instance will be created for a single bean definition every time a request is made for that bean.

3. **Request:** A new instance will be created for a single bean definition every time an HTTP request is made for that bean. But Only valid in the context of a web-aware Spring ApplicationContext.
4. **Session:** Scopes a single bean definition to the lifecycle of an HTTP Session. But Only valid in the context of a web-aware Spring ApplicationContext.
5. **Global-Session:** Scopes a single bean definition to the lifecycle of a global HTTP Session. It is also only valid in the context of a web-aware Spring ApplicationContext.

Singleton Scope:

If the scope is a singleton, then only one instance of that bean will be instantiated per Spring IoC container and the same instance will be shared for each request. That is when the scope of a bean is declared singleton, then whenever a new request is made for that bean, spring IOC container first checks whether an instance of that bean is already created or not. If it is already created, then the IOC container returns the same instance otherwise it creates a new instance of that bean only at the first request. By default, the scope of a bean is a singleton.

Prototype Scope:

If the scope is declared **prototype**, then spring IOC container will create a new instance of that bean every time a request is made for that specific bean. A request can be made to the bean instance either programmatically using **getBean()** method or by XML for Dependency Injection of secondary type. Generally, we use the prototype scope for all beans that are stateful, while the singleton scope is used for the stateless beans.

Request Scope

In request scope, container creates a new instance for each and every HTTP request. So, if the server is currently handling 50 requests, then the container can have at most 50 individual instances of the bean class. Any state change to one instance, will not be visible to other instances. **A bean instance is destructed as soon as the request is completed**.

Session Scope

In session scope, the application context creates a new instance for each and every HTTP session. So, if the server has 20 active sessions, then the container can have at most 20 individual instances of the bean class. All HTTP

requests within a single session lifetime will have access to the same single bean instance in that session scope.

Any state change to one instance will not be visible to other instances. **An instance is destructed as soon as the session ends.**

Application Scope

In application scope, the container creates one instance per web application runtime. It is almost similar to singleton scope with only two differences i.e.

1. The application scoped bean is singleton per ServletContext, whereas singleton scoped bean is singleton per ApplicationContext. Please note that **there can be multiple application contexts within a single application**.
2. The application scoped bean is visible as a ServletContext attribute.

WebSocket Scope

The WebSocket Protocol enables two-way communication between a client and a remote host that has opted-in to communicate with the client. WebSocket Protocol provides a single TCP connection for traffic in both directions. This is especially useful for multi-user applications with simultaneous editing and multi-user games.

In this type of web application, HTTP is used only for the initial handshake. The server can respond with HTTP status 101 (switching protocols) if it agrees – to the handshake request. If the handshake succeeds, the TCP socket remains open, and both the client and server can use it to send messages to each other.

When first accessed, *WebSocket* scoped beans are stored in the *WebSocket* session attributes. The same bean instance is then returned during the entire *WebSocket* session.

Please note that websocket scoped beans are typically singleton and live longer than any individual WebSocket session.

LECTURE-35

Autowiring in Spring

Autowiring in the Spring framework can inject dependencies automatically. The Spring container detects those dependencies specified in the configuration file and the relationship between the beans. This is referred to as **Autowiring in Spring**. To enable Autowiring in the Spring application we should use @Autowired annotation. Autowiring in Spring internally uses constructor injection. An autowired application requires fewer lines of code comparatively but at the same time, it provides very little flexibility to the programmer.

Modes of Autowiring

Modes	Description
No	This mode tells the framework that auto wiring is not supposed to be done. It is the default mode used by Spring.
byname	It uses the name of the bean for injecting dependencies.
byType	It injects the dependency according to the type of bean.
Constructor	It injects the required dependencies by invoking the constructor.
Autodetect	The autodetect mode uses two other

Advantage of Autowiring

It requires the **less code** because we don't need to write the code to inject the dependency explicitly.

Disadvantage of Autowiring

No control of programmer.

Annotations

Annotations are a form of metadata that provides data about a program. Annotations are used to provide supplemental information about a program. It does not have a direct effect on the operation of the code they annotate. It does not change the action of the compiled program. So in this article, we are going to discuss what are the main types of annotation that are available in the spring framework with some examples.

Use of java annotations

Java annotations are mainly used for the following:

- Compiler instructions
- Build-time instructions
- Runtime instructions

Compiler instructions: Java provides the 3 in built annotations which are used to give certain instructions to the compiler. Java in built annotation are @Deprecated, @Override & @SuppressWarnings.

Build-time instructions: Java annotations can be used for build time or compile time instructions. These instructions can be used by the build tools for generating source code, compiling the source, generating XML files, packaging the compiled code and files into a JAR file etc.

Runtime instructions: Normally, Java annotations are not present in your Java code after compilation. However, we can define our own annotations that can be available at runtime. These annotations can be accessed using Java Reflection.

Java annotations basics:

A java annotation always starts with the symbol @ and followed by the annotation name. The @symbol signals the compiler that this is an annotation.

Syntax:

```
@AnnotationName
```

Example:

```
@Entity
```

Here @ symbol signals the compiler that this is an annotation and the Entity is the name of this annotation.An annotation can contain zero, one or multiple elements. We have to set values for these elements. **Example:**

```
@Entity(tableName = "USERS")
```

Where we can use annotations?

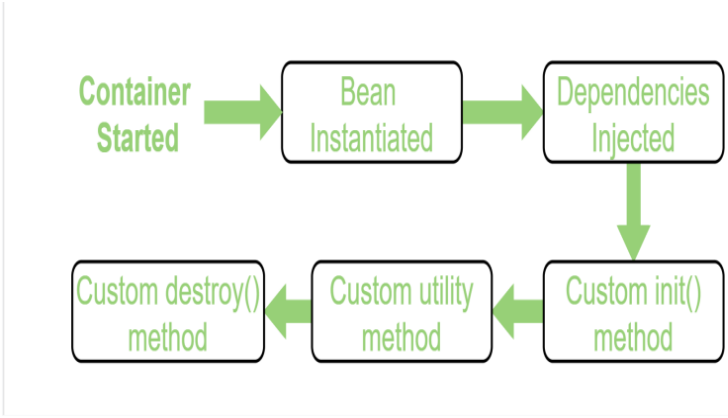
We can use java annotations above classes, interfaces, methods, fields and local variables. Here is an example annotation added above a class definition:

```
@Entity
 public class Users {
}
```

Spring Bean Life Cycle and Callbacks

The lifecycle of any object means when & how it is born, how it behaves throughout its life, and when & how it dies. Similarly, the bean life cycle refers to when & how the bean is instantiated, what action it performs until it lives, and when & how it is destroyed. In this article, we will discuss the life cycle of the bean.

Bean life cycle is managed by the spring container. When we run the program then, first of all, the spring container gets started. After that, the container creates the instance of a bean as per the request, and then dependencies are injected. And finally, the bean is destroyed when the spring container is closed. Therefore, if we want to execute some code on the bean instantiation and just after closing the spring container, then we can write that code inside the custom **init()** method and the **destroy()** method.
The following image shows the process flow of the bean life cycle.



Note: We can choose a custom method name instead of **init()** and **destroy()**. Here, we will use init() method to execute all its code as the spring container starts up and the bean is instantiated, and destroy() method to execute all its code on closing the container.

1. Spring bean life cycle involves initialization and destruction callbacks and Spring bean aware classes.
   2. Initialization callback methods execute after dependency injection is completed. Their purposes are to check the values that have been set in bean properties, perform any custom initialization or provide a wrapper on original bean etc. Once the initialization callbacks are completed, bean is ready to be used.
   3. When IoC container is about to remove bean, destruction callback methods execute. Their purposes are to release the resources held by bean or to perform any other finalization tasks.
   4. When more than one initialization and destructions callback methods have been implemented by bean, then those methods execute in certain order.

SPRING CONFIGURATION STYLE

Spring Framework provides three ways to configure beans to be used in the application.

1. **Annotation Based Configuration** - By using @Service or @Component annotations. Scope details can be provided with @Scope annotation.
2. **XML Based Configuration** - By creating Spring Configuration XML file to configure the beans. If you are using Spring MVC framework, the xml based configuration can be loaded automatically by writing some boiler plate code in web.xml file.
3. **Java Based Configuration** - Starting from Spring 3.0, we can configure Spring beans using java programs. Some important annotations used for java based configuration are @Configuration, @ComponentScan and @Bean.

LECTURE-36
Spring Boot

Spring Boot is most commonly used and contains all Spring Framework features. Nowadays Spring Boot is becoming the best choice for Java developers to build rapid Spring applications. When Java Developers use Spring Boot for developing applications then they will

focus on the logic instead of struggling with the configuration setup environment of the application.

In Spring Boot, choosing a build system is an important task. We recommend Maven or Gradle as they provide a good support for dependency management. Spring does not support well other build systems.

Dependency Management

Spring Boot team provides a list of dependencies to support the Spring Boot version for its every release. You do not need to provide a version for dependencies in the build configuration file. Spring Boot automatically configures the dependencies version based on the release. Remember that when you upgrade the Spring Boot version, dependencies also will upgrade automatically.

Note – If you want to specify the version for dependency, you can specify it in your configuration file. However, the Spring Boot team highly recommends that it is not needed to specify the version for dependency.

Maven Dependency

For Maven configuration, we should inherit the Spring Boot Starter parent project to manage the Spring Boot Starters dependencies.

We should specify the version number for Spring Boot Parent Starter dependency. Then for other starter dependencies, we do not need to specify the Spring Boot version number.

Gradle Dependency

We can import the Spring Boot Starters dependencies directly into **build.gradle** file. We do not need Spring Boot start Parent dependency like Maven for Gradle.

SPRING BOOT CODE STRUCTURE

Spring Boot does not have any code layout to work with. However, there are some best practices that will help us. This chapter talks about them in detail.

Default package

A class that does not have any package declaration is considered as a **default package**. Note that generally a default package declaration is not recommended. Spring Boot will cause issues such as malfunctioning of Auto Configuration or Component Scan, when you use default package.

Note – Java's recommended naming convention for package declaration is reversed domain name.

Spring Boot - Runners

Application Runner and Command Line Runner interfaces lets you to execute the code after the Spring Boot application is started. You can use these interfaces to perform any actions immediately after the application has started. This chapter talks about them in detail.

Application Runner

Application Runner is an interface used to execute the code after the Spring Boot application started.

Command Line Runner

Command Line Runner is an interface. It is used to execute the code after the Spring Boot application started.

LOGGER

Logging in Spring Boot plays a vital role in Spring Boot applications for recording information, actions, and events within the app. It is also used for monitoring the performance of an application, understanding the behavior of the application, and recognizing the issues within the application. Spring Boot offers flexible logging capabilities by providing various logging frameworks and also provides ways to manage and configure the logs.

Why to use Spring Boot – Logging?

A good logging infrastructure is necessary for any software project as it not only helps in understanding what's going on with the application but also to trace any unusual incident

or error present in the project. This article covers several ways in which logging can be enabled in a spring boot project through easy and simple configurations. Let's first do the initial setup to explore each option in more depth.

Elements of Logging Framework

- Logger: It captures the messages.
- Formatter: It formats the messages which are captured by loggers.
- Handler: It prints messages on the console, stores them in a file or sends an email, etc.

Java provides several logging frameworks, some of which are:

1. *Logback Configuration logging*
1. *Log4j2 Configuration logging*

Lecture-37

Introduction to RESTful Web Services

REST stands for **REpresentational State Transfer**. It is developed by **Roy Thomas Fielding**, who also developed HTTP. The main goal of RESTful web services is to make web services **more effective**. RESTful web services try to define services using the different concepts that are already present in HTTP. REST is an **architectural approach**, not a protocol.

It does not define the standard message exchange format. We can build REST services with both XML and JSON. JSON is more popular format with REST. The **key abstraction** is a resource in REST. A resource can be anything. It can be accessed through a **Uniform Resource Identifier (URI)**. For example:

The resource has representations like XML, HTML, and JSON. The current state capture by representational resource. When we request a resource, we provide the representation of the resource. The important methods of HTTP are:

- **GET:** It reads a resource.
- **PUT:** It updates an existing resource.
- **POST:** It creates a new resource.
- **DELETE:** It deletes the resource.

For example, if we want to perform the following actions in the social media application, we get the corresponding results.

POST /users: It creates a user.

GET /users/{id}: It retrieves the detail of a user.

GET /users: It retrieves the detail of all users.

DELETE /users: It deletes all users.

DELETE /users/{id}: It deletes a user.

GET /users/{id}/posts/post_id: It retrieve the detail of a specific post.

POST / users/{id}/ posts: It creates a post of the user.

Further, we will implement these URI in our project.

HTTP also defines the following standard status code:

- **404:** RESOURCE NOT FOUND
- **200:** SUCCESS
- **201:** CREATED
- **401:** UNAUTHORIZED
- **500:** SERVER ERROR

RESTful Service Constraints

- There must be a service producer and service consumer.
- The service is stateless.
- The service result must be cacheable.
- The interface is uniform and exposing resources.
- The service should assume a layered architecture.

Advantages of RESTful web services

- RESTful web services are **platform-independent**.
- It can be written in any programming language and can be executed on any platform.
- It provides different data format like **JSON, text, HTML,** and **XML**.
- It is fast in comparison to SOAP because there is no strict specification like SOAP.
- These are **reusable**.
- They are **language neutral**.

Spring RestController annotation is used to create RESTful web services using Spring MVC. Spring RestController takes care of mapping request data to the defined request handler method. Once response body is generated from the handler method, it converts it to JSON or XML response.3 Aug 2022

RestController: *RestController is used for making restful web services with the help of the @RestController annotation. This annotation is used at the class level and allows the class to handle the requests made by the client. Let's understand @RestController annotation using an example. The RestController allows to handle all REST APIs such as GET, POST, Delete, PUT requests.*

RestController is very useful when we are working on a real-time REST API Spring Application. This Rest Controller class gives us JSON(JavaScript Object Notation) response as the output. The normal class is annotated with “@RestController” then that class is treated as Rest Controller class in the class.

Difference between @Controller and @RestController:

```
@Controller
@ResponseBody
class TestController{
----------------
----------------
}@RestController
class TestController{
----------------
----------------
}
```

A normal class annotated with “@Controller” then that class is treated as a Controller class in Spring MVC.

Spring 2.5 introduced the @Controller annotation used for web application return as a view in Spring MVC. This is the specialization of the “@Component” annotation.

In @Controller we need to use @ReponseBody in every handler method for response output without a view.

Spring 4.0 introduced the “@RestController” for the creation of the Restful web services in the Spring Framework in a simple manner. It is a specialization of the “@Controller” annotation.

In the “@RestController” annotation, we don’t need to use the “@ResponseBody” annotation in the handler method. In this “@RestController” we cannot return a view as output.

In this Project, we can use “@RestController” and “@GetMapping” annotations in the POJO class.

@RestController: This combination of these annotations “@Controller” and “@ResponseBody”.This will be used when we are making Rest Apis.

@GetMapping: It will handle requests from HTTP Get Method (Client).

Development Process:
1. Keep eclipse IDE ready
2. Create the Spring Boot Starter Project for this example of the RestController in the Spring Boot(Select Spring Web dependency)
3. Create RestController class
4. Run the Project

Lecture-38

When building robust APIs, understanding and appropriately utilizing the HTTP methods GET, POST, PUT, and DELETE is essential. Each method serves a specific purpose and has its own limitations. In this article, we will explore these HTTP methods, their characteristics, and discuss their limitations in the context of building robust APIs. Additionally, we will provide Java code examples to demonstrate their usage.

What Are HTTP Methods and Why Are They Important?

HTTP methods, also known as HTTP verbs, are a set of standardized actions that can be performed on a resource using the Hypertext Transfer Protocol (HTTP). These methods define the intended operation to be performed on the resource and provide a uniform way of interacting with web servers. The most commonly used HTTP methods are GET, POST, PUT, and DELETE, but there are other methods as well, such as PATCH, HEAD, and OPTIONS.

HTTP methods are important for several reasons:

1. **Resource Manipulation**: HTTP methods allow clients to perform various operations on resources, such as retrieving data, submitting data, updating data, or deleting data. Each method represents a specific action that can be taken on a resource, enabling a wide range of interactions between clients and servers.
2. **Uniform Interface**: HTTP methods provide a uniform interface for interacting with web resources. By adhering to the standard set of methods, clients and servers can communicate effectively, regardless of the underlying technologies or platforms being used. This promotes interoperability and simplifies the development and integration of web applications.
3. **Intent and Semantics**: Each HTTP method carries a specific intent and semantic meaning, making it easier for developers to understand the purpose of an API endpoint by looking at the method used. For example, a GET request is used to retrieve data, while a POST request is used to submit data for processing or storage. By selecting the appropriate method, developers can convey the intended operation more accurately.
4. **Idempotence and Safety**: HTTP methods have different characteristics regarding idempotence and safety. Idempotence means that making multiple identical requests should have the same outcome. Safe methods, such as GET, should not cause any modifications or side effects on the server. Understanding these characteristics helps developers design APIs that adhere to the expected behavior and minimize unintended side effects or data inconsistencies.
5. **RESTful Architecture**: HTTP methods play a fundamental role in building RESTful APIs (Representational State Transfer). REST is an architectural style that leverages HTTP methods to provide a scalable and flexible approach to designing web services. Each resource in a RESTful API is typically associated with a specific URL, and the appropriate HTTP method is used to interact with that resource.

By understanding and utilizing the appropriate HTTP methods, developers can build robust and well-designed APIs that adhere to the principles of HTTP and REST. This ensures consistency, interoperability, and efficiency in the communication between clients and servers.

Below we will elaborate more on the most commonly used HTTP methods.

GET Method:

The GET method is one of the fundamental HTTP methods used for retrieving data from a server. It is designed to be safe, meaning it should not modify any data on the server. When using the GET method, the client requests a representation of a resource from the server.

Here are some key points to consider when working with the GET method in API development:

1. **Retrieving Data**: The primary purpose of the GET method is to retrieve data from the server. It is commonly used to fetch resources such as user profiles, product information, blog articles, and more. The client sends a GET request to a specific URL, and the server responds with the requested data.
2. **Request Parameters**: GET requests can include parameters in the URL to provide additional information to the server. These parameters are appended to the URL as query parameters, typically in the form of key-value pairs. For example, a GET request to retrieve user information for a specific ID could look like: `GET /users?id=123`. The server can use these parameters to filter or modify the returned data accordingly.
3. **Idempotence:** The GET method is considered idempotent, meaning that multiple identical GET requests should have the same outcome. It implies that making multiple GET requests for the same resource should not result in any unintended side effects or modifications on the server. It allows caching mechanisms to be employed for efficient retrieval and reduces the risk of unintentional changes or inconsistencies.
4. **Response Codes:** The server responds to a GET request with an appropriate HTTP response code to indicate the success or failure of the request. The most common response code is 200 (OK), indicating that the request was successful, and the requested resource is returned in the response body. Other possible response codes include 404 (Not Found) if the requested resource does not exist, or 500 (Internal Server Error) if there was an error on the server while processing the request.
5. **Limitations**: While the GET method is crucial for retrieving data, it has certain limitations to consider:
   - **Data Length**: GET requests have limitations on the length of the URL. Excessive data in the URL can result in truncation, errors, or security vulnerabilities. It is recommended to keep the data size within reasonable limits and consider alternative methods (e.g., POST) for large payloads.
   - **Security Considerations**: Since GET requests expose the data in the URL, it is important to avoid including sensitive information like passwords or authentication tokens in the query parameters. Instead, consider using other secure methods such as headers or request bodies for transmitting sensitive data.

Java Code Example:

```
01  import java.net.HttpURLConnection;
02  import java.net.URL;
03  import java.io.BufferedReader;
04  import java.io.InputStreamReader;
05
06  public class GetExample {
07      public static void main(String[] args) {
```

```
        try {
            URL url = new URL("https://api.example.com/resource");
            HttpURLConnection connection = (HttpURLConnection)
url.openConnection();
            connection.setRequestMethod("GET");

            int responseCode = connection.getResponseCode();
            if (responseCode == HttpURLConnection.HTTP_OK) {
                BufferedReader reader =
                        new BufferedReader(new
InputStreamReader(connection.getInputStream()));
                String line;
                StringBuilder response = new StringBuilder();
                while ((line = reader.readLine()) != null) {
                    response.append(line);
                }
                reader.close();
                System.out.println("Response: " + response.toString());
            } else {
                System.out.println("Error: " + responseCode);
            }
            connection.disconnect();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

In the given Java code example, the GET request is sent to `https://api.example.com/resource`. The response from the server is then read and stored in a StringBuilder for further processing. If the response code is 200 (HTTP_OK), the response is printed to the console.

while the GET method is efficient for retrieving data, it should not be used for operations that modify the server state. For such operations, other HTTP methods like POST, PUT, or DELETE should be employed.

POST Method:

The POST method is one of the HTTP methods used for submitting data to be processed by the server. Unlike the GET method, which is used for retrieving data, the POST method is intended for data submission and can cause modifications on the server. Here are some important points to consider when working with the POST method:

1. **Submitting Data**: The primary purpose of the POST method is to submit data to the server for processing or storage. This data can be in various formats such as form data, JSON, XML, or binary data. The POST request typically includes a request body that contains the data being sent to the server.
2. **Idempotence**: Unlike the GET method, the POST method is generally considered non-idempotent. This means that multiple identical POST requests may have different outcomes or effects on the server. For example, submitting the same POST request multiple times may result in the creation of multiple resources or duplicate entries.
3. **Request Headers**: POST requests often include specific headers to provide additional information about the request or the format of the data being sent. For example, the "Content-Type" header specifies the format of the data in the request body, such as "application/json" or "application/x-www-form-urlencoded".
4. **Response Codes**: Similar to other HTTP methods, the server responds to a POST request with an appropriate HTTP response code to indicate the success or failure of the request. The common response code for a successful POST request is 201 (Created), indicating that the resource has been successfully created on the server. Other possible response codes include 200 (OK) for general success or 400 (Bad Request) for invalid or malformed requests.
5. **Limitations**: While the POST method is commonly used for data submission, it also has some limitations to consider:
   - **Lack of Idempotence**: As mentioned earlier, the non-idempotent nature of the POST method means that repeated identical requests may lead to unintended side effects. It is important to design APIs in a way that handles duplicate submissions appropriately and ensures data integrity.
   - **Lack of Caching:** By default, POST requests are typically not cacheable. Caching is important for improving performance and reducing the load on the server. If caching is required for POST requests, additional measures such as Cache-Control headers or server-side caching strategies need to be implemented.

Java Code Example:

```
01 import java.net.HttpURLConnection;
02 import java.net.URL;
03 import java.io.DataOutputStream;
04
05 public class PostExample {
06   public static void main(String[] args) {
07       try {
08     URL url = new URL("https://api.example.com/resource");
09           HttpURLConnection connection = (HttpURLConnection)
10    url.openConnection();
11           connection.setRequestMethod("POST");
12           connection.setDoOutput(true);
13
14           String postData = "data=example";
15           DataOutputStream outputStream = new
16    DataOutputStream(connection.getOutputStream());
17           outputStream.writeBytes(postData);
18           outputStream.flush();
19           outputStream.close();
20
           int responseCode = connection.getResponseCode();
           if (responseCode == HttpURLConnection.HTTP_OK) {
```

```
            // Process response
        } else {
            // Handle error
        }
        connection.disconnect();
    } catch (Exception e) {
        e.printStackTrace();
    }
  }
}
```

In the provided Java code example, a POST request is sent to `https://api.example.com/resource` with a request body containing the data to be submitted. The data is written to the request output stream and the response code is checked to handle the response accordingly.
When using the POST method, it is crucial to ensure proper authentication, authorization, and input validation to prevent security vulnerabilities and protect the integrity of the server and data.

Remember to use the appropriate HTTP method based on the intended operation. While the GET method is used for retrieving data, the POST method is suitable for submitting data for processing or creating new resources on the server.

PUT Method:

The PUT method is an HTTP method used for updating or replacing a resource on the server. It is idempotent, meaning that multiple identical PUT requests should have the same outcome. Here are some important points to consider when working with the PUT method:

1. **Updating Resources**: The primary purpose of the PUT method is to update an existing resource on the server. It replaces the entire resource with the new representation provided in the request. The PUT request typically includes a request body containing the updated data for the resource.
2. **Idempotence**: As mentioned earlier, the PUT method is idempotent. It means that making multiple identical PUT requests should have the same outcome. This property allows for safe retries of failed requests without causing unintended side effects or data inconsistencies.
3. **Resource Identification**: To update a specific resource, the client must provide the unique identifier or URL of the resource in the PUT request. The server uses this information to locate the resource and perform the update operation. It is crucial to ensure the accuracy and integrity of the resource identification mechanism to prevent unintended modifications to unrelated resources.
4. **Partial Updates**: By default, the PUT method replaces the entire resource with the new representation provided in the request body. However, in some cases, it may be desirable to perform partial updates, modifying only specific fields or properties of the resource. While the HTTP specification does not directly support partial updates with the PUT

method, some APIs implement custom conventions or use additional operations (e.g., PATCH) to achieve partial updates.

5. **Response Codes**: Similar to other HTTP methods, the server responds to a PUT request with an appropriate HTTP response code to indicate the success or failure of the request. The common response code for a successful PUT request is 200 (OK), indicating that the resource has been successfully updated. Alternatively, if the resource does not exist and a new resource is created, the response code may be 201 (Created).
6. **Limitations**: While the PUT method is commonly used for resource updates, it also has some limitations to consider:
   - **Lack of Partial Updates**: As mentioned earlier, the PUT method typically replaces the entire resource rather than allowing partial updates to specific fields. If partial updates are required, alternative approaches like PATCH or custom conventions can be considered.
   - **Security Considerations**: It is essential to implement proper authentication, authorization, and validation mechanisms to ensure that only authorized clients can update the resources. Additionally, input validation should be performed to prevent invalid or malicious data from being stored.

Java Code Example:

```
01
02  import java.net.HttpURLConnection;
03  import java.net.URL;
04  import java.io.DataOutputStream;
05
06  public class PutExample {
07      public static void main(String[] args) {
08          try {
09              URL url = new URL("https://api.example.com/resource");
10              HttpURLConnection connection = (HttpURLConnection)
    url.openConnection();
11              connection.setRequestMethod("PUT");
12              connection.setDoOutput(true);
13
14              String putData = "data=updated";
15              DataOutputStream outputStream = new
    DataOutputStream(connection.getOutputStream());
16              outputStream.writeBytes(putData);
17              outputStream.flush();
18              outputStream.close();
19
20              int responseCode = connection.getResponseCode();
21              if (responseCode == HttpURLConnection.HTTP_OK) {
22                  // Process response
23              } else {
24                  // Handle error
25              }
26              connection.disconnect();
27          } catch (Exception e) {
28              e.printStackTrace();
29          }
30      }
    }
```

In the provided Java code example, a PUT request is sent to `https://api.example.com/resource` with a request body containing the updated data. The data is written to the request output stream, and the response code is checked to handle the response accordingly.
When using the PUT method, it is important to handle concurrency and data consistency issues appropriately. For example, you may use optimistic locking mechanisms or versioning to ensure that updates do not conflict with other concurrent modifications to the resource.

Remember to use the appropriate HTTP method based on the intended operation. While the GET method is used for retrieving data and the POST method is used for submitting data, the PUT method is suitable for updating existing resources on the server.

DELETE Method:

The DELETE method is an HTTP method used for deleting a specified resource on the server. It is used to remove a resource permanently from the server. Here are some important points to consider when working with the DELETE method:

1. **Deleting Resources**: The primary purpose of the DELETE method is to delete a specific resource on the server. The client sends a DELETE request to the server, specifying the URL or identifier of the resource to be deleted.
2. **Idempotence:** Similar to the PUT method, the DELETE method is also idempotent. Multiple identical DELETE requests should have the same outcome. Making repeated DELETE requests for the same resource should not result in any unintended side effects or modifications on the server.
3. **Resource Identification:** To delete a specific resource, the client must provide the unique identifier or URL of the resource in the DELETE request. The server uses this information to locate and remove the corresponding resource. It is crucial to ensure the accuracy and integrity of the resource identification mechanism to prevent accidental deletions of unrelated resources.
4. **Response Codes**: The server responds to a DELETE request with an appropriate HTTP response code to indicate the success or failure of the request. The common response code for a successful DELETE request is 204 (No Content), indicating that the resource has been successfully deleted. Alternatively, if the resource does not exist, the response code may be 404 (Not Found).
5. **Limitations**: While the DELETE method is commonly used for resource deletion, it is important to consider the following limitations:
    - **Lack of Safety**: Unlike the GET method, which is considered safe and should not modify any data on the server, the DELETE method performs irreversible actions. Once a resource is deleted, it cannot be easily recovered. Therefore, it is crucial to implement appropriate authorization and authentication mechanisms to ensure that only authorized clients can initiate DELETE requests.
    - **Cascading Deletions**: In some cases, deleting a resource may have cascading effects on related resources. For example, deleting a user may require deleting associated

records such as posts or comments. It is essential to define the behavior and potential cascading actions in your API's design and documentation.

Java Code Example:

```
import java.net.HttpURLConnection;
import java.net.URL;

public class DeleteExample {
    public static void main(String[] args) {
        try {
            URL url = new URL("https://api.example.com/resource/123");
            HttpURLConnection connection = (HttpURLConnection)
url.openConnection();
            connection.setRequestMethod("DELETE");

            int responseCode = connection.getResponseCode();
            if (responseCode == HttpURLConnection.HTTP_NO_CONTENT) {
                // Resource deleted successfully
            } else if (responseCode == HttpURLConnection.HTTP_NOT_FOUND)
{
                // Resource not found
            } else {
                // Handle other errors
            }
            connection.disconnect();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

In the provided Java code example, a DELETE request is sent to `https://api.example.com/resource/123`, where "123" represents the identifier of the resource to be deleted. The response code is checked to handle the success or failure of the deletion operation.

When using the DELETE method, it is important to implement proper authorization and authentication mechanisms to prevent unauthorized deletions. Additionally, consider providing proper error handling and feedback to the client in case of failures or errors during the deletion process.

Remember to use the appropriate HTTP method based on the intended operation. While the GET method is used for retrieving data, the POST method is used for submitting data, the PUT method is used for updating data, the DELETE method is specifically designed for resource deletion.

Use Cases and Examples of HTTP Methods

Here are some common use cases and examples of the HTTP methods GET, POST, PUT, and DELETE:

GET:

- Use Case: Retrieving Data
- Description: The GET method is used to retrieve data from a specified resource on the server.
- Example: Fetching a user's profile information from an API endpoint:

```
URL url = new URL("https://api.example.com/users/123");
HttpURLConnection connection = (HttpURLConnection) url.openConnection();
connection.setRequestMethod("GET");

// Process the response
```

POST:

- Use Case: Submitting Data
- Description: The POST method is used to submit data to be processed or stored by the server.
- Example: Creating a new user by sending data in the request body:

```
URL url = new URL("https://api.example.com/users");
HttpURLConnection connection = (HttpURLConnection)
url.openConnection();
connection.setRequestMethod("POST");
connection.setDoOutput(true);

// Set the request body with user data
DataOutputStream outputStream = new
DataOutputStream(connection.getOutputStream());
String userData = "name=John&email=john@example.com";
outputStream.writeBytes(userData);
outputStream.flush();
outputStream.close();

// Process the response
```

PUT:

- Use Case: Updating Data
- Description: The PUT method is used to update or replace a specified resource on the server.
- Example: Updating a user's information by sending the updated data in the request body:

```
URL url = new URL("https://api.example.com/users/123");
HttpURLConnection connection = (HttpURLConnection) url.openConnection();
connection.setRequestMethod("PUT");
connection.setDoOutput(true);

// Set the request body with updated user data
DataOutputStream outputStream = new
DataOutputStream(connection.getOutputStream());
String updatedUserData = "name=John Smith&email=john.smith@example.com";
outputStream.writeBytes(updatedUserData);
outputStream.flush();
outputStream.close();
```

```
11
12    // Process the response
13
```

DELETE:

- Use Case: Deleting Data
- Description: The DELETE method is used to delete a specified resource from the server.
- Example: Deleting a user by sending a DELETE request to the corresponding API endpoint:

```
1   URL url = new URL("https://api.example.com/users/123");
2   HttpURLConnection connection = (HttpURLConnection) url.openConnection();
3   connection.setRequestMethod("DELETE");
4
5   // Process the response
```

These examples demonstrate how the HTTP methods can be used in different scenarios to perform common operations on resources. It's important to note that these are just simplified examples, and in real-world scenarios, you would typically handle error handling, authentication, and other considerations to build robust and secure APIs.

Understanding the characteristics and limitations of the HTTP methods GET, POST, PUT, and DELETE is crucial for building robust APIs. Each method serves a specific purpose and should be used appropriately to ensure data integrity and consistent behavior. By leveraging these HTTP methods effectively, developers can design APIs that are efficient, secure, and reliable.

Lecture-39

RequestMapping

One of the most important annotations in spring is the **@RequestMapping Annotation** which is used to map HTTP requests to handler methods of MVC and REST controllers. In Spring MVC applications, the DispatcherServlet (Front Controller) is responsible for routing incoming HTTP requests to handler methods of controllers. When configuring Spring MVC, you need to specify the mappings between the requests and handler methods. To configure the mapping of web requests, we use the **@RequestMapping** annotation. The @RequestMapping annotation can be applied to class-level and/or method-level in a controller. The class-level annotation maps a specific request path or pattern onto a controller. You can then apply additional method-level annotations to make mappings more specific to handler methods. So let's understand @RequestMapping Annotation at Method-level and Class level by examples.

Requirements:

- *Eclipse (EE version)/STS IDE*
- *Spring JAR Files*
- *Tomcat Apache latest version*

@RequestMapping Annotation at Method-Level

Spring Boot is the most popular framework of Java for building enterprise-level web applications and back-ends. Spring Boot has a handful of features that support quicker and more efficient web app development. Some of them are Auto-configuration, Embedded Server, opinionated defaults, and Annotation Support. In this article, we'll be exploring the core annotation of **Spring Boot – @RequestMapping** which is part of the set of annotations that Spring Boot employs for defining URL endpoints and REST APIs.

@RequestMapping

This annotation is a versatile and flexible annotation that can be used with a controller (class) as well as the methods to map specific web requests with the handler methods and controllers. This annotation is part of a larger set of annotations provided by Spring Framework to define URL endpoints and simplify the development of Spring Boot applications.

It has the following features:

- Define several different endpoints to access a specific resource.
- Build REST APIs to serve web requests.
- Simplify the web development process by simply defining an annotation that offers a set of functionalities for handling requests.
- Define multiple endpoints in a single @RequestMapping annotation.

What is Request Body?

Data sent over the request body can be of any format like json, XML, PDF, Http Forms, and many more. The **Content-Type** header indicates the server's understanding type of request.

- Spring provides **@RequestBody** annotation to deserialize the incoming payload into java object or map.
- The request body goes along with **content-type** header for handler method to understand and deserialize the payload.

How to Get the Body of Request in Spring Boot?

Java language is one of the most popular languages among all programming languages. There are several advantages of using the Java programming language, whether for security purposes or building large distribution projects. One of the advantages of using Java is that Java tries to connect every concept in the language to the real world with the help of the concepts of classes, inheritance, polymorphism, etc.
There are several other concepts present in Java that increase the user-friendly interaction between the Java code and the programmer such as generic, Access specifiers, Annotations, etc. These features add an extra property to the class as well as the method of the Java program. In this article, we will discuss how to get the body of the incoming request in the spring boot.
@RequestBody: Annotation is used to get the request body in the incoming request.

Spring Initializr is a web-based tool using which we can easily generate the structure of the Spring Boot project. It also provides various features for the projects expressed in a metadata model. This model allows us to configure the list of dependencies that are supported by JVM. Here, we will create the structure of an application using a spring initializer and then use an IDE to create a sample GET route. Therefore, to do this, the following steps are followed sequentially as follows:

@RequestBody Annotation in Spring and Spring Boot

The `@RequestBody` annotation is responsible for binding the HTTPRequest body to the body of the web request. Depending on the content type of the request, the body of the request is given through a `HttpMessageConverter`, which resolves the method argument.

We can also use the `@Valid` annotation to automatically validate the input.
In brief, the `@RequestBody` annotation is responsible for retrieving the request body and automatically converting it to the Java object.

path variable in spring boot

The @PathVariable annotation is used to retrieve data from the URL path. By defining placeholders in the request mapping URL, you can bind those

placeholders to method parameters annotated with @PathVariable. This allows you to access dynamic values from the URL and use them in your code.

The @PathVariable annotation is used to retrieve data from the URL path. By defining placeholders in the request mapping URL, you can bind those placeholders to method parameters annotated with @PathVariable. This allows you to access dynamic values from the URL and use them in your code.

Using @PathVariable

The @PathVariable annotation is used to extract data from the URL path. It allows you to define placeholders in your request mapping URL and bind those placeholders to method parameters. Let's consider an example where you have a REST API endpoint for retrieving a user's details by their ID:

- **RestController:** *RestController is used for making restful web services with the help of the @RestController annotation. This annotation is used at the class level and allows the class to handle the requests made by the client. Let's understand @RestController annotation using an example. The RestController allows to handle all REST APIs such as GET, POST, Delete, PUT requests.*
- Spring Initializr is a web-based tool using which we can easily generate the structure of the Spring Boot project. It also provides various different features for the projects expressed in a metadata model. This model allows us to configure the list of dependencies that are supported by JVM. Here, we will create the structure of an application using a spring initializer and then use an IDE to create a sample GET route.

Lecture-40

How to build a Web Application Using Java

Java is one of the most used programming languages for developing dynamic web applications. A web application is computer software that utilizes the web browser and technologies to perform tasks over the internet. A web application is deployed on a web server.

Java provides some technologies like Servlet and JSP that allow us to develop and deploy a web application on a server easily. It also provides some frameworks such as Spring, Spring Boot that simplify the work and provide an efficient way to develop a web application. They reduce the effort of the developer.

We can create a website using static HTML pages and style them using CSS, but we need server-side technology when we want to create a dynamic website.

In this section, we will see how to create a website using Java Servlets and HTML. Further, we will see how these technologies are useful for developing a web application.

What is a Web Application

A web application is computer software that can be accessed using any web browser. Usually, the frontend of a web application is created using the scripting languages such as HTML, CSS, and JavaScript, supported by almost all web browsers. In contrast, the backend is created by any of the programming languages such as Java, Python, Php, etc., and databases. Unlike the mobile application, there is no specific tool for developing web applications; we can use any of the supported IDE for developing the web application.

Web Server and Client

The web server is a process that handles the client's request and responds. It processes the request made by the client by using the related protocols. The main function of the webserver is to store the request and respond to them with web pages. It is a medium between client and server. For example, Apache is a leading webserver.

A client is a software that allows users to request and assist them in communicating with the server. The web browsers are the clients in a web application; some leading clients are Google Chrome, Firefox, Safari, Internet Explorer, etc.

HTML and HTTP

The HTML stands for HyperText Markup Language; it is a common language for Web Server and Web Client communication. Since both the web server and web client are two different software components of the web, we need a language that communicates between them.

The HTTP stands for HyperText Transfer Protocol; it is a communication protocol between the client and the server. It runs on top of the TCP/IP protocol.

Some of the integral components of an HTTP Request are as following:

HTTP Method: The HTTP method defines an action to be performed; usually, they are GET, POST, PUT, etc.

URL: URL is a web address that is defined while developing a web application. It is used to access a webpage.

Form Parameters: The form parameter is just like an argument in a Java method. It is passed to provide the details such as user, password details on a login page.

What is URL

URL stands for Universal Resource Locator used to locate the server and resource. It is an address of a web page. Every web page on a project must have a unique name.

A URL looks like as follows:

1. http://localhost:8080/SimpleWebApplication/

Where,

http or https: It is the starting point of the URL that specifies the protocol to be used for communication.

Localhost: The localhost is the address of the server. When we run our application locally, it is called localhost; if we deployed our project over the web, then it is accessed by using the domain name like "javatpoint.com". The domain name maps the server to IP addresses.

8080: This is the port number for the local server; it is optional and may differ in different machines. If we do not manually type the port number in the URL, then by default, the request goes to the default port of the protocol. Usually, the port no between 0 to 1023 are reserved for some well-known services such as HTTP, HTTPS, FTP, etc.

We have discussed all the major components of a web application. Let's move towards our main motive How to build a web application in Java.

First, understand servlet:

What is Servlet

A Servlet is a Java program that runs within a web server; it receives the requests and responds to them using related protocols (Usually HTTP). The Servlets are capable enough to respond to any type of request; they are commonly used to make the application functional.

We can create a static website using only HTML and CSS, but when it comes to dynamic, we need a server-side programming language. For these applications, Java provides Servlet technology, which contains HTTP-specific servlet classes.

The **javax.servlet** and **javax.servlet.http** packages contain interfaces and classes for creating servlets. All servlets should implement the Servlet interface, which defines life-cycle methods. To implement a generic service, we can use the GenericServlet class by extending it. It provides **doGet** and **doPost** methods to handle HTTP-specific services.

Why are the Servlets Useful?

Web servers are capable enough to serve static HTML requests, but they don't know how to deal with dynamic requests and databases. So, we need a language for dynamic content; these languages are PHP, Python, Java, Ruby on Rails, etc. In Java, there are two technologies Servlet and JSPs, that deals with dynamic content and database. Java also provides frameworks such as Spring, Spring Boot, Hibernate, and Struts to use the servlet and JSP easily.

The Servlets and JSPs are server-side technologies that extend the functionality of a web server. They support dynamic response and data persistence. We can easily create a web application using these technologies.

Let's create our first web applications:

First Web Application Using Java Servlet

To create a web application, we need the following tools:

Java

IDE (Eclipse or Netbeans)

Database (Oracle or Mysql)

Server (Tomcat)

Before Creating any web application, ensure that all of the above tools are properly installed on your system.

Now, follow the below steps to develop a web application:

Step1: Open Eclipse Create a Dynamic Web Project

Open the Eclipse IDE, navigate to **File-> New-> Dynamic Web Project.**

If the dynamic web project is not listed in your IDE, then go to the other option and search for it. Click on it to continue.

Step2: Provide Project Name

Now, enter the project name and click **Next** to continue.

Follow the prompt and tick the generate **web.xml** deployment descriptor.

Now, our project is ready; the project structure will look as follows:

Step3: Create a Servlet

Now, create a servlet by right-clicking on the **Java Resources/src** folder. To create a servlet right click on the **src** folder and navigate to **the New-> Servlet** menu. Here, provide the Servlet name:

Click on the **Finish** button. It will create a TestServlet as specified. You can choose any of your Servlet names.

Step4: Add the Servlet Jar file

We can see our Servlet is displaying lots of errors it is because we have not added the **servlet-api** jar file yet. To add the jar file, right-click on the project and select the configuration option by navigating to **Build Path-> Configure Build Path** option. Now, click on the **Add External JARs** option.

Navigate to the directory where you have installed your server and select the **servlet-api.jar** file.

Click **Open** to continue.

Now select **Apply and Close** option. It will add the jar file to our project.

Step5: Create a HTML or JSP file

Now, our first web application is almost ready. We can create HTML pages that we want to display on our website.

To create an HTML page, right-click on the **WebContent** folder and select the New HTML file option from the **New-> HTML File** menu with the name **index.html**.

Map the File

Now, map this file in the web.xml file. The web.xml is a deployment descriptor for the Servlet applications. Since, Servlet 3.0, we can use annotations instead of the deployment descriptor.

We can also define our welcome file; a welcome file is the first file of the project that initiates the project, also known as Home. We can define multiple welcome files.

Consider the below code:

From the above code, we can see by default the servlet defines several welcome files. If you want to use any file other than the listed files, you can define that here.

Now, our first web application is ready.

Step7: Run the Application

To run the application, right-click on the project and run it on the server by selecting **Run-> Run on Server** option.

It will take some time to load the application.

We can also test it on other browsers by entering the URL.

Now, we can design this by adding more web pages and styles.

In the above screen, we have updated our index.html file as follows:

Add the image file into **WebContent** folder.