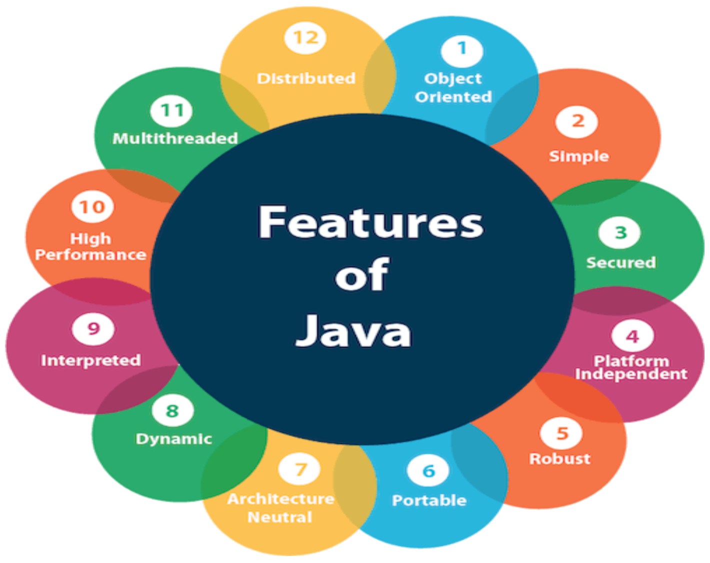
Uday Kumar S

07th June 2024

Features of JAVA.
The primary objective of Java programming language creation was to make it portable, simple and secure programming language. Apart from this, there are also some excellent features which play an important role in the popularity of this language. The features of Java are also known as Java buzzwords.

A list of the most important features of the Java language is given below.

1. Object Oriented
2. Simple
3. Secured
4. Platform Independent
5. Robust
6. Portable
7. Architectural Neutral
8. Dynamic
9. Interpreted
10. High Performance
11. Multithreaded
12. Distributed



Why is JAVA Platform Independent?
Java is known as a platform-independent language because of its ability to run the same compiled bytecode across different operating systems and hardware platforms without modification. This characteristic stems from several key features of the Java ecosystem:

Bytecode: When you compile a Java program, the Java compiler (javac) converts your source code into an intermediate form known as bytecode, stored in .class files. This bytecode is not specific to any type of machine; it's a standardized set of instructions meant to be executed by the Java Virtual Machine (JVM).

Java Virtual Machine (JVM): The JVM is the cornerstone of Java's platform independence. Each operating system (Windows, macOS, Linux, etc.) has its own JVM implementation. When you run a Java program, the JVM interprets the bytecode into the machine code of the underlying platform. Essentially, the JVM acts as an intermediary between the bytecode and the hardware, ensuring that the same Java program can run on any platform that has a compatible JVM.

Standardized Libraries: Java comes with a comprehensive standard library, which provides a wide array of functionalities, from basic data structure handling to network operations. These libraries are written in Java and compiled into platform-neutral bytecode, ensuring that they behave the same way on any platform. This standardization helps in maintaining consistency across different environments.

Garbage Collection: Java manages memory allocation and deallocation in the background through a process called garbage collection, which is handled by the JVM. This feature not only helps in managing memory more efficiently but also ensures that Java applications can run within various environments without modifications for memory management.

How to run a JAVA program?
Writing the Code: Java programs are written in plain text files using any text editor or an Integrated Development Environment (IDE) like IntelliJ IDEA or Eclipse. The code is written in Java, a high-level programming language.

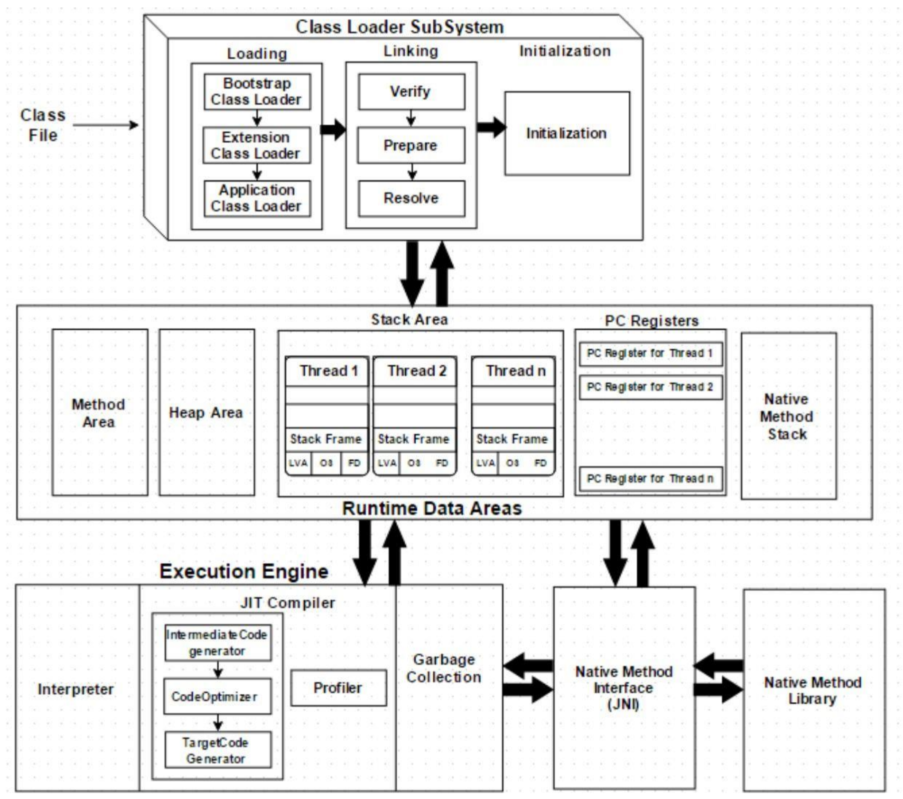
Saving the File: The code is saved with a .java extension, which identifies it as a Java source file. The name of the file should match the public class name contained within the file. For example, if your class is named HelloWorld, the file should be named HelloWorld.java.

Compiling the Code: Before a Java program can be executed, it must be compiled. The compilation is done using the Java Compiler (javac), which converts the source code (.java files) into bytecode. Bytecode is a set of instructions that the Java Virtual Machine (JVM) can understand but is not specific to any one machine's hardware.

Running the Program: To run the compiled Java program, you use the Java Runtime Environment (JRE), which includes the Java Virtual Machine (JVM). The JVM reads and interprets the bytecode and executes the program.

You run the program using the java command followed by the name of the class that contains the main method (without the .class extension):
Java HelloWorld
This command tells the JVM to execute the bytecode in HelloWorld.class, starting with the main method.

JVM Architecture



Class Loader Subsystem

The Class Loader Subsystem is a crucial component of the Java Virtual Machine (JVM) that plays a vital role in the loading, linking, and initialization of Java classes. Let's delve deeper into each stage and its purpose:

1. Loading: In this stage, the Class Loader reads the .class files—compiled Java bytecode—from the filesystem or network sources and converts them into a binary stream. This stream is then used to create instances of the Class class in the JVM, which represents these classes as objects in the JVM memory. Here are the types of class loaders involved:

- ***Bootstrap Class Loader:*** It is the parent of all other class loaders and is responsible for loading key Java classes from the JAVA_HOME/jre/lib directory, such as the java.lang.* classes.
- ***Extension Class Loader:*** This loader handles classes that are extensions of the standard core Java classes, which are available from the JAVA_HOME/jre/lib/ext directory or any other directory specified by the java.ext.dirs system property.
- ***System/Application Class Loader:*** It loads classes from the classpath specified by the user (e.g., via the -cp or -classpath command-line options). This includes your application's classes.

2. Linking: After a class is loaded, it undergoes linking which consists of three main activities:

- ***Verification:*** Ensures the correctness of the loaded class files, checking if the code adheres to the Java Language Specification and does not violate JVM internal constraints. This step is crucial as it prevents things like memory corruption by verifying bytecode before it's run.
- ***Preparation:*** The JVM allocates memory for class variables and initializes the memory to default values. This preparation phase is about setting up the necessary data structures in the JVM's method area.
- ***Resolution:*** This is an optional stage that involves replacing symbolic references from the type with direct references. It's done to support dynamic linking and can happen during the resolution phase or lazily at runtime when a symbolic reference is first used.

3. Initialization

In this final phase, all static variables are assigned their values as defined in the code, and static blocks are executed. This is done from the parent to child classes and from the top to the bottom of a class. This step ensures that the class is set up correctly before its instances are created.

Key Points:

- If class is not found in any it throws ***ClassNotFoundException.***
- ***Hierarchy and Delegation Model:*** Class loaders typically use a delegation model where a class loader delegates class loading to its parent before attempting to load the class itself. This model ensures that classes available in the parent class loader are not reloaded by the child, maintaining consistency across different parts of the application.
- ***Security:*** The class loading mechanism also plays a critical role in Java's security model. By controlling class loading, the JVM can isolate and protect resources from untrusted code sources.
- ***Performance:*** To enhance performance, class loaders cache classes. Once a class is loaded into the JVM, it does not need to be reloaded. This caching mechanism significantly speeds up the performance of Java applications.

The Class Loader Subsystem is fundamental to the JVM's operation, affecting everything from the application's performance to its security. It carefully manages the way Java classes are introduced into the JVM, ensuring that they are integrated in a secure, efficient, and orderly manner.

Memory Area

The Memory Area in the JVM is organized into several key regions, each serving specific purposes in the operation and execution of a Java application:

Method Area: This area stores class-level information such as class structures, method data, method code, and runtime constant pools. It is shared among all threads and is also sometimes referred to as the "Permanent Generation" in older versions of JVM or "Metaspace" in more recent versions of Java.

Heap Area: The JVM Heap is the runtime data area from which memory for all class instances (objects) and arrays are allocated. It is a shared resource accessible to all threads. Garbage Collection, which is the process of automatically freeing memory occupied by objects that are no longer in use, primarily occurs in this area.

Java Stack: Each Java thread has its own stack, known as the Java Stack, created at the same time as the thread. This stack stores frames, which contain local variable arrays, operand stacks, and method invocation details. The Java Stack is used for managing method calls and returns. Each method call creates a new frame that is pushed onto the stack, and each method return pops a frame off the stack.

Program Counter (PC) Register: This is a small memory space where the JVM keeps the address of the Java virtual machine instruction currently being executed. Each thread has its own PC Register.

Native Method Stack: Similar to the Java Stack, the Native Method Stack supports native methods (methods written in languages other than Java) used in the application. It handles the execution of native code, such as C/C++ invoked through the Java Native Interface (JNI).

Execution Engine

The Execution Engine is the component of the JVM that executes the bytecode loaded into the memory area. It includes:

Interpreter: The Interpreter reads bytecode instructions one at a time, interprets them, and executes them directly. While straightforward and simple, interpreting bytecode is slower compared to compiled execution because each instruction is interpreted as and when needed.

Just-In-Time (JIT) Compiler: To overcome the speed limitations of interpretation, the JVM includes a JIT compiler. The JIT compiler compiles bytecode into native machine code that the computer's processor can execute directly, improving performance significantly. This compilation happens at runtime, and the compiled code is used directly for subsequent calls, which reduces the need for interpretation.
Adaptive Optimization: Modern JIT compilers perform optimizations like inlining, dead code elimination, and loop unrolling during the compilation process. These optimizations are based on runtime data, making them highly effective.

Garbage Collector: As part of the execution engine, the Garbage Collector automatically manages memory. It identifies and disposes of objects that are no longer in use by the application, helping to reclaim heap space and prevent memory leaks.

Runtime Constant Pool: A runtime constant pool is a per-class structure that contains several kinds of constants, from numeric literals known at compile-time to method and field references that are resolved at runtime.

These components of the JVM work in concert to perform the complex task of executing Java applications. The Memory Area handles the allocation and management of memory for application data, while the Execution Engine directly handles the interpretation, compilation, and execution of the program code, making Java capable of high performance despite its high level of abstraction and platform independence.

Uday Kumar S

10th June 2024

Access Modifiers in Java

Java access modifiers are keywords used in object-oriented programming to set the access level or scope of classes, constructors, variables, and methods. They play a crucial role in providing encapsulation, one of the fundamental principles of object-oriented programming.

1. **Default Access Modifier (Package-Private)**

Keyword: None (no keyword is used).
Scope: Accessible within the same package only.
Usage: If you do not specify any access modifier, the class, method, or data member is treated as having the default access modifier. This is also referred to as package-private, meaning the accessible scope is limited to other classes within the same package. It's often used when you want to allow class members to be accessed within the same package but hidden from classes in other packages.

2. **Private Access Modifier**

Keyword: private
Scope: Accessible only within the class it is declared.
Usage: Private access is the most restrictive level of access. Variables and methods declared as private can be accessed only within the declared class itself. They are not visible to subclasses and completely hidden from other classes outside the class they are declared in. Private access is typically used for helper methods and fields that are used internally by the class.

3. **Protected Access Modifier**

Keyword: protected
Scope: Accessible within the same package or any subclasses, even if they are in different packages.
Usage: Protected access provides a more permissive access level than private but is more restrictive than public. It is used when you want to hide the member from the world, yet allow subclasses to access it. This is particularly useful in inheritance hierarchies to give subclass methods the ability to access superclass members.

4. **Public Access Modifier**

Keyword: public
Scope: Accessible from any other class.
Usage: The public access modifier is the least restrictive of all access levels. Classes, methods, or data members declared public can be accessed from any other class in the Java universe, assuming the classes are accessible (imported and compiled). Use public access when you need to provide a broad interface to your class.

Uday Kumar S

11th June 2024

Class, Interface, and Object Keywords

- **class**: The class keyword is used to declare a class, which is a blueprint for creating objects. A class can contain fields (variables) and methods to define the behaviour of its objects.
- **interface**: The interface keyword is used to declare an interface. An interface in Java is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces cannot contain instance fields. The methods in interfaces are abstract by default.
- **enum**: The enum keyword defines a fixed set of constants. Enumerations extend the base class java.lang.Enum automatically and provide a type-safe way of defining a set of predefined constants. This means you can have a variable that can be set to one of a predefined set of constants (and nothing else).
- **extends**: The extends keyword is used in class declarations to inherit from a superclass. It indicates that a class is a subclass of another class and can inherit fields and methods from the superclass.
- **implements**: The implements keyword is used in class declarations to specify that a class implements an interface. It is used to inherit abstract methods from the interface.
- **new**: The new keyword is used to create new objects. It initializes a new object and calls the constructor to set up the object.

Control Flow Statements

- **if**

Syntax: if (condition) { /* statements */ }
Use Case: Executes a block of code only if the specified condition evaluates to true.
Ex.

```
if (age > 18) {
   System.out.println("Adult");
}
```

- **else**

Syntax: Accompanies an if statement and executes if the if condition is false.
Use Case: Provides an alternative execution path when the if condition is not met.
Ex.

```
if (age > 18) {
   System.out.println("Adult");
} else {
   System.out.println("Minor");
}
```

- **switch**

Uday Kumar S

Syntax: switch (expression) { case value: /* statements */ break; /* more cases */ }
Use Case: Selects the execution path based on the value of an expression. More efficient than multiple if-else statements when checking the same variable for different values.
Ex.

```
switch (day) {
case 1:
System.out.println("Monday");
break;
case 2:
System.out.println("Tuesday");
break;
default:
System.out.println("Weekend");
}
```

- **case**

Syntax: Used within a switch statement to specify a block to execute when the expression matches the case.
Use Case: Defines specific actions for distinct values of the switch expression.

- **default**

Syntax: Used in a switch statement as the fallback when no other case matches.
Use Case: Ensures a default action is taken when no specified cases match the switch expression.

- **while**

Syntax: while (condition) { /* statements */ }
Use Case: Repeats a block of code as long as the condition remains true.
Ex.

```
while (count < 5) {
   System.out.println("Count is " + count);
   count++;
}
```

- **do**

Syntax: do { /* statements */ } while (condition);
Use Case: Similar to the while loop, but guarantees that the block is executed at least once before the condition is checked.
Ex.

```
do {
   count++;
   System.out.println("Count is " + count);
} while (count < 5);
```

- **for**

Syntax: for (initialization; condition; increment) { /* statements */ }

Use Case: Iterates over a range or through elements of a collection, handling initialization, condition checking, and incrementing in one line.
Ex.
```
for (int i = 0; i < 10; i++) {
   System.out.println("i is " + i);
}
```

- **break**

Syntax: break;
Use Case: Exits a loop or a switch statement prematurely.
Ex. Use within a loop or switch to stop execution and exit the structure.

- **continue**

Syntax:: continue;
Use Case: Skips the remaining statements in the current loop iteration and proceeds to the next iteration.
Ex. Use to skip specific conditions in a loop.

- **return**

Syntax: return; or return value;
Use Case: Exits from the current method and optionally returns a value to the method caller.
Ex.
```
int doubleValue(int x) {
   return 2 * x;
}
```

- **try**

Syntax: try { /* code */ } catch (ExceptionType e) { /* handler */ }
Use Case: Begins a block of code that will be tested for exceptions.

- **catch**

Syntax: Follows a try block and catches exceptions thrown within the try.
Use Case: Handles specific types of exceptions that may have occurred in the try block.

- **finally**

Syntax: Can follow try or catch blocks and executes regardless of whether an exception was thrown.
Use Case: For code that must be executed after a try block, regardless of whether an exception was thrown or caught.

- **throw**

Syntax: throw new ExceptionType("message");
Use Case: Used to explicitly throw an exception, usually based on a custom condition.
Ex.

```
if (x < 0) {
   throw new IllegalArgumentException("x must be non-negative");
}
```

- **throws**

Syntax: Used in a method signature to indicate that a method might throw one or more exceptions.
Use Case: Declares exceptions that a method can throw, informing callers of the method that they should handle or propagate these exceptions.
Ex.

```
public void readFile(String path) throws IOException {
   // code to read a file
}
```

Primitive Data Types

Type	Size	Range	Default
boolean	1 bit	true or false	false
byte	8 bits	[-128, 127]	0
short	16 bits	[-32,768, 32,767]	0
char	16 bits	['\u0000', '\uffff'] or [0, 65535]	'\u0000'
int	32 bits	[-2,147,483,648 to 2,147,483,647]	0
long	64 bits	[-2^{63}, $2^{63}-1$]	0
float	32 bits	32-bit IEEE 754 floating-point	0.0
double	64 bits	64-bit IEEE 754 floating-point	0.0

Non-Access Modifiers

- **Abstract**

Usage: Applied to classes and methods.
Behaviour:
Abstract Class: Cannot be instantiated directly. Must be subclassed by other classes for implementation.
Abstract Method: Does not have a body and must be implemented by subclasses.
Ex.

```
public abstract class Animal {
   public abstract void makeSound();
}
```

- **final**

Usage: Can be applied to classes, methods, or variables.
Behaviour:
Final Class: Cannot be subclassed.
Final Method: Cannot be overridden by subclasses.
Final Variable: Its value cannot be changed once assigned.

Ex.

```
public final class Constants {
   public static final int MAX_SIZE = 100;
}
```

- **static**

Usage: Used with variables and methods.
Behaviour:
Static Variable: Shared among all instances of a class, belonging to the class itself rather than any object.
Static Method: Can be invoked without creating an instance of the class.
Ex.

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

- **synchronized**

Usage: Applied to methods and code blocks.
Behaviour: Ensures that only one thread can access the method or block at a time, which is crucial for thread safety in multi-threaded applications
Ex.

```
public synchronized void increment() {
   this.count++;
}
```

- **volatile**

Usage: Used with variables.
Behaviour: Indicates that a variable's value will be modified by different threads and ensures that the value of the variable is always read from the main memory and not from the thread's cache
Ex.

```
public volatile boolean running = true;
```

- **transient**

Usage: Used with class fields.
Behaviour: Specifies that a field should not be serialized. This is useful when you don't want sensitive data to be part of the serialized state or when a field is derived from other fields and need not be serialized.
Ex.

```
public class UserSession implements Serializable {
   private transient String passwordHash;
}
```

More Java keywords

- **this**

Usage: Refers to the current instance of a class.

Behaviour: Used within an instance method or a constructor to refer to the current object.
Ex.
```
public class Car {
   private int modelYear;

   public Car(int modelYear) {
      this.modelYear = modelYear; // Differentiates the instance variable from the parameter
   }
}
```

- **super**

Usage: Refers to the superclass (parent class) of the current object.
Behaviour: Used to access methods and constructors of the superclass.
Ex.
```
public class ElectricCar extends Car {
   public ElectricCar(int modelYear) {
      super(modelYear); // Calls the constructor of the superclass Car
   }
}
```

- **void**

Usage: Specifies that a method does not return a value.
Behaviour: Used in the method declaration.
Ex.
```
public void display() {
   System.out.println("This method returns no value.");
}
```

- **const** and **goto**

Usage: Both are reserved keywords in Java but are not used.
Behaviour: goto and const are reserved for not breaking older code but they don't serve any function in current Java versions

- **instanceof**

Usage: Tests whether an object is an instance of a specific class or an interface.
Behaviour: Returns true if the specified object is an instance of the specified type (class or interface).
Ex.
```
if (car instanceof ElectricCar) {
   System.out.println("It's an electric car!");
}
```

- **package**

Usage: Declares a package.
Behaviour: Used at the beginning of a source file to specify the package that the file belongs to.
Ex.
```
package com.example.mycar;
```

Uday Kumar S

- **import**

Usage: Imports other Java packages or classes.
Behaviour: Makes external classes and packages available to the code.
Ex.

```
import java.util.List;
```

- **native**

Usage: Indicates that a method is implemented in native code using the Java Native Interface (JNI).
Behaviour: Used when linking Java code to code written in another programming language like C or C++
Ex.

```
public native void performFastCalculation();
```

- **strictfp**

Usage: Restricts floating-point calculations to ensure portability.
Behaviour: Ensures that you get the same results from your floating-point calculations on all platforms.
Ex.

```
public strictfp class Calculator {
   // Ensures consistent floating-point results
}
```

- **null**:

Usage: Represents a null reference, i.e., one that points to no object.
Behaviour: The default value for object reference variables.
Ex.

```
Car myCar = null; // myCar does not reference any object
```

- **assert**

Usage: Used for debugging purposes to make an assertion.
Behaviour: Tests for factual assumptions made by the programmer. If an assertion evaluates to false, an AssertionError is thrown.
Ex.

```
assert x > 0 : "x must be positive"; // Throws AssertionError if x is not positive
```

Uday Kumar S

12th June 2024

Variables
A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location. It is a combination of "vary + able" which means its value can be changed.

There are three types of variables in Java:

- ***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.

- ***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.

- ***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.

1. **Objects**

- It is also called a blueprint of a class or virtual entity.
- Software development groups can use a modular, object-oriented design-and-implementation approach to be much more productive than with earlier popular techniques like "structured programming"—object-oriented programs are often easier to understand, correct and modify.
- Objects, or more precisely, the classes objects, are essentially reusable software components.
- Almost any noun can be reasonably represented as a software object in terms of attributes (e.g., name, color and size) and behaviours (e.g., calculating, moving and communicating).

2. **Method**

- Performing a task in a program requires a method.
- The method houses the program statements that actually perform its tasks.
- Hides these statements from its user, just as the accelerator pedal of a car hides from the driver the mechanisms of making the car go faster.

3. Class

- In Java, we create a program unit called a class to house the set of methods that perform the class's tasks.
- A class is similar in concept to a car's engineering drawings, which house the design of an accelerator pedal, steering wheel, and so on.

4. Java Inheritance

- Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class to inherit properties (methods and fields) from another class. In Java, inheritance facilitates code reuse, improves readability, and establishes a natural hierarchy between classes.
- The different types of inheritance which are supported by Java.
  - ***Single Inheritance***
    - Definition: A subclass inherits from only one superclass.
    - Example: Class Dog inherits from class Animal

```
class Animal {
   void eat() {
      System.out.println("This animal eats food.");
   }
}

class Dog extends Animal {
   void bark() {
      System.out.println("The dog barks.");
   }
}
```

  - ***Multilevel Inheritance***
    - Definition: A class inherits from a subclass making it a multilevel hierarchy.
    - Example: Class BabyDog inherits from class Dog which inherits from class Animal.

```
class BabyDog extends Dog {
   void weep() {
      System.out.println("The baby dog weeps.");
   }
}
```

  - ***Hierarchical Inheritance***
    - Definition: Multiple classes inherit from a single superclass.
    - Example: Classes Dog and Cat both inherit from class Animal.

```
class Cat extends Animal {
   void meow() {
      System.out.println("The cat meows.");
   }
}
```

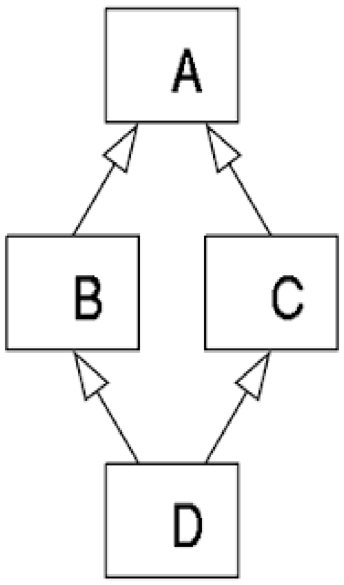
  - ***Multiple Inheritance***
    - Definition: A class can implement multiple interfaces.
    - Note: Java does not support multiple inheritance with classes due to ambiguity issues (like the diamond problem), but it allows a class to implement multiple interfaces.
    - Example: Class SwimmingDog can implement interfaces Swimmer and Dog

```
interface Swimmer {
   void swim();
}

interface Barker {
   void bark();
}

class SwimmingDog implements Swimmer, Barker {
   public void swim() {
      System.out.println("The dog swims.");
   }
   public void bark() {
      System.out.println("The dog barks.");
   }
}
```

- **Hybrid Inheritance**
  - Definition: A mix of two or more of the above types of inheritance.
  - Example: This typically involves a combination of multilevel and hierarchical inheritance.
  - Note: While Java does not support hybrid inheritance involving multiple class inheritance, it can be partially achieved through the use of interfaces.

The Diamond Problem:

The diamond problem occurs in languages that allow multiple inheritance of classes. It refers to an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C. If B and C have overridden an inherited method and D does not override it, there is an ambiguity about which version of the method D should inherit.
Java's Solution: Java avoids the diamond problem by not allowing multiple class inheritance. Instead, it uses interfaces to achieve similar functionality without ambiguity, as methods in interfaces do not have an implementation by default (until Java 8 introduced default methods, which must be carefully managed to avoid similar issues).



Java's approach to inheritance through interfaces and single class inheritance helps maintain clear and unambiguous implementation pathways, ensuring that Java programs remain robust, maintainable, and free from the complexities that multiple class inheritance might introduce.

5. **Polymorphism**

- Polymorphism allows methods to do different things based on the object that is acting upon them. This is fundamental to OOP because it helps make the program more scalable and easy to maintain.
- Compile-Time Polymorphism (Static Polymorphism): Achieved through method overloading.

```
class MathOperation {
   int multiply(int a, int b) {
      return a * b;
   }
```

```
    double multiply(double a, double b) {
        return a * b;
    }
}
```

- Runtime Polymorphism (Dynamic Polymorphism): Achieved through method overriding.

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}
```

6. **Method Overloading in Java**

- Method overloading occurs when multiple methods in the same class have the same name but different parameters (different type, number, or both).
- Allows a class to have more than one method handling different data types or different numbers of input parameters.
- You can have a method add(int, int) and another method add(double, double) in the same class.

```
class Display {
    void show(int a) {
        System.out.println(a);
    }
    void show(String a) {
        System.out.println(a);
    }
}
```

7. **Method Overriding in Java**

- Method overriding happens when a subclass has a method with the same name, return type, and parameters as a method in its superclass.
- Allows a subclass to provide a specific implementation of a method that is already provided by its parent class.
- Overriding the toString() method of the superclass in a subclass to provide more specific string representation for the object.

```
class Vehicle {
    void run() {
        System.out.println("Vehicle is running");
    }
}
class Bike extends Vehicle {
    @Override
    void run() {
        System.out.println("Bike is running safely");
    }
}
```

8. Java Abstraction

- Abstraction is a process of hiding the implementation details and showing only functionality to the user.
- Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.
- Abstraction lets you focus on what the object does instead of how it does it.

```
abstract class Device {
   abstract void turnOn();
}
class TV extends Device {
   @Override
   void turnOn() {
      System.out.println("TV turned on");
   }
}
```

9. Java Encapsulation

- Classes (and their objects) encapsulate, i.e., encase, their attributes and methods.
- Objects may communicate with one another, but they’re normally not allowed to know how other objects are implemented—implementation details are hidden within the objects themselves.
- Information hiding is crucial to good software engineering.

```
class Account {
   private double balance;

   public void deposit(double amount) {
      if (amount > 0) {
         balance += amount;
      }
   }

   public double getBalance() {
      return balance;
   }
}
```

10. Rules for Java Method Overriding

- Method Signature: The overriding method must have the same name, return type, and parameter list as the method in the superclass.
- Access Level: The access level cannot be more restrictive than the overridden method.
- Superclass Method: The method being overridden must be marked with public, protected, or no access modifier (default access) to be visible to the subclass.

11. Constructors in Java

- A constructor is a block of code used to initialize an object. It has the same name as the class and does not have a return type.
- Constructors are used to set up initial conditions or perform setup operations when an instance of a class is created.

```
class Book {
   String title;
   Book(String t) {
      title = t;
   }
}
```

12. Types of Java Constructors

- No-Arg Constructor: A constructor with no parameters.
- Parameterized Constructor: A constructor that takes one or more parameters to set initial properties of the object.

13. Constructor Overloading in Java

- Constructor overloading involves having more than one constructor in a class, each having a different parameter list.
- Allows the creation of objects in different ways depending on the information available at the time of instantiation.

```
class Point {
   int x, y;
   Point() {
      this(0, 0);
   }
   Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
}
```

14. Interface in Java

- An interface in Java is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces cannot contain instance fields, and all methods are abstract by default.
- Interfaces are used to specify a set of methods that a class must implement. They are used for achieving abstraction and multiple inheritance in Java.
- A class implements an interface using the implements keyword, and all methods defined in the interface must be implemented by the class.

```
interface Animal {
   void eat();
}
class Dog implements Animal {
   public void eat() {
      System.out.println("Dog eats");
   }
}
```

13th June 2024

Exception Handling

Uday Kumar S

Exception Handling in Java is one of the effective means to handle runtime errors so that the regular flow of the application can be preserved. Java Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

What are Java Exceptions?

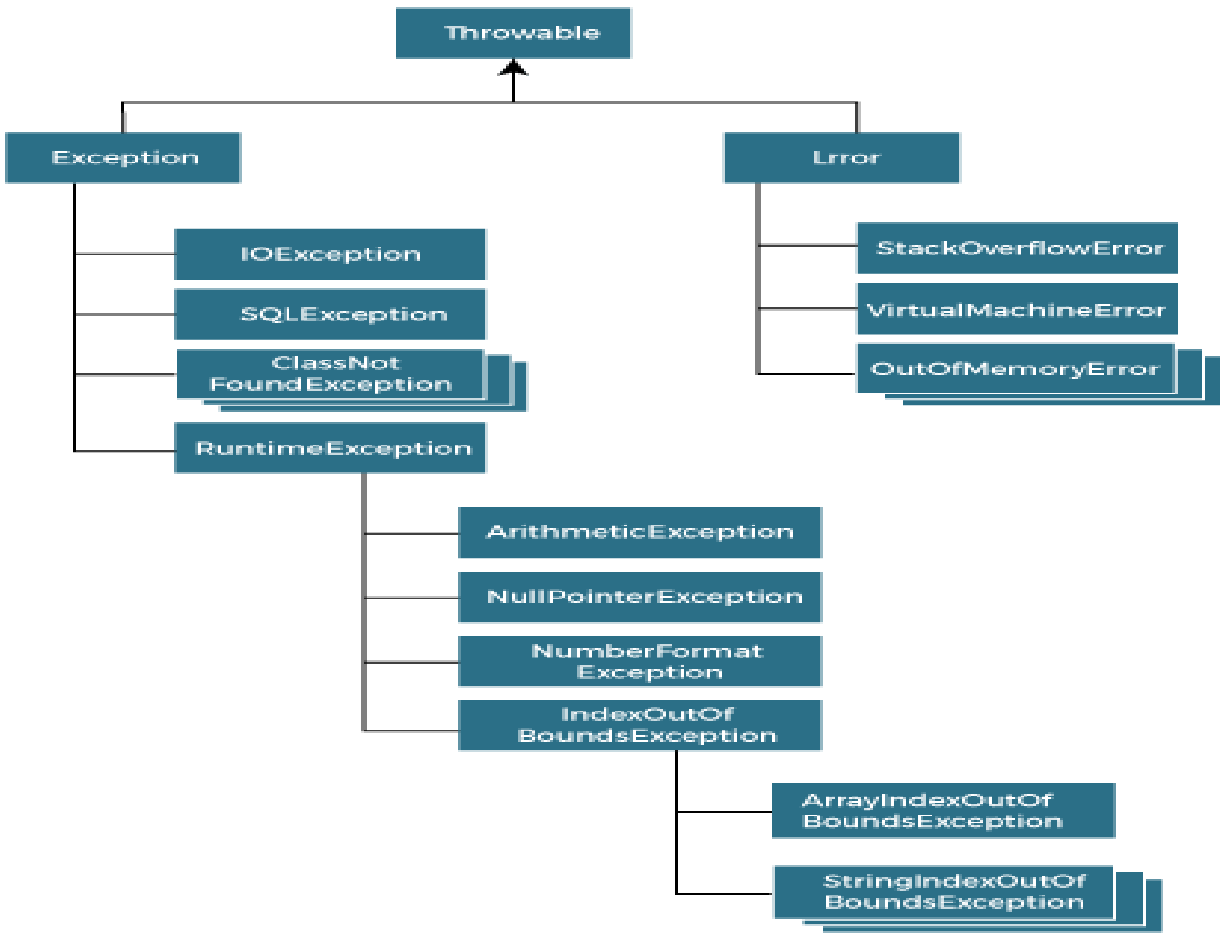
In Java, Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions. Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This 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.

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.

Throwable is the superclass of all Exceptions in Java.

Difference between Error and Exception

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

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.

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

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'.

How Does JVM Handle an Exception?

Default Exception Handling: Whenever inside a method, if an exception has occurred, the method creates an Object known as an Exception Object and hands it off to the run-time system(JVM). The exception object contains the name and description of the exception and the current state of the program where the exception has occurred. Creating the Exception Object and handling it in the run-time system is called throwing an Exception. There might be a list of the methods that had been called to get to the method where an exception occurred. This ordered list of methods is called Call Stack. Now the following procedure will happen.

The run-time system searches the call stack to find the method that contains a block of code that can handle the occurred exception. The block of the code is called an Exception handler.
The run-time system starts searching from the method in which the exception occurred and proceeds through the call stack in the reverse order in which methods were called.

If it finds an appropriate handler, then it passes the occurred exception to it. An appropriate handler means the type of exception object thrown matches the type of exception object it can handle.
If the run-time system searches all the methods on the call stack and couldn't have found the appropriate handler, then the run-time system handover the Exception Object to the default exception handler, which is part of the run-time system. This handler prints the exception information in the following format and terminates the program abnormally.

How Programmer Handle an Exception?

Customized Exception Handling: Java exception handling is managed via five keywords: try, catch, throw, throws, and finally. Briefly, here is how they work. Program statements that you think can raise exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch block) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed after a try block completes is put in a finally block.

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 try-catch Syntax

```
try{
//code that may throw an exception
}catch(Exception_class_Name ref){}
```

Uday Kumar S

Java try-finally Syntax

```
try{
//code that may throw an exception
}finally{}
```

Java Multi-catch block

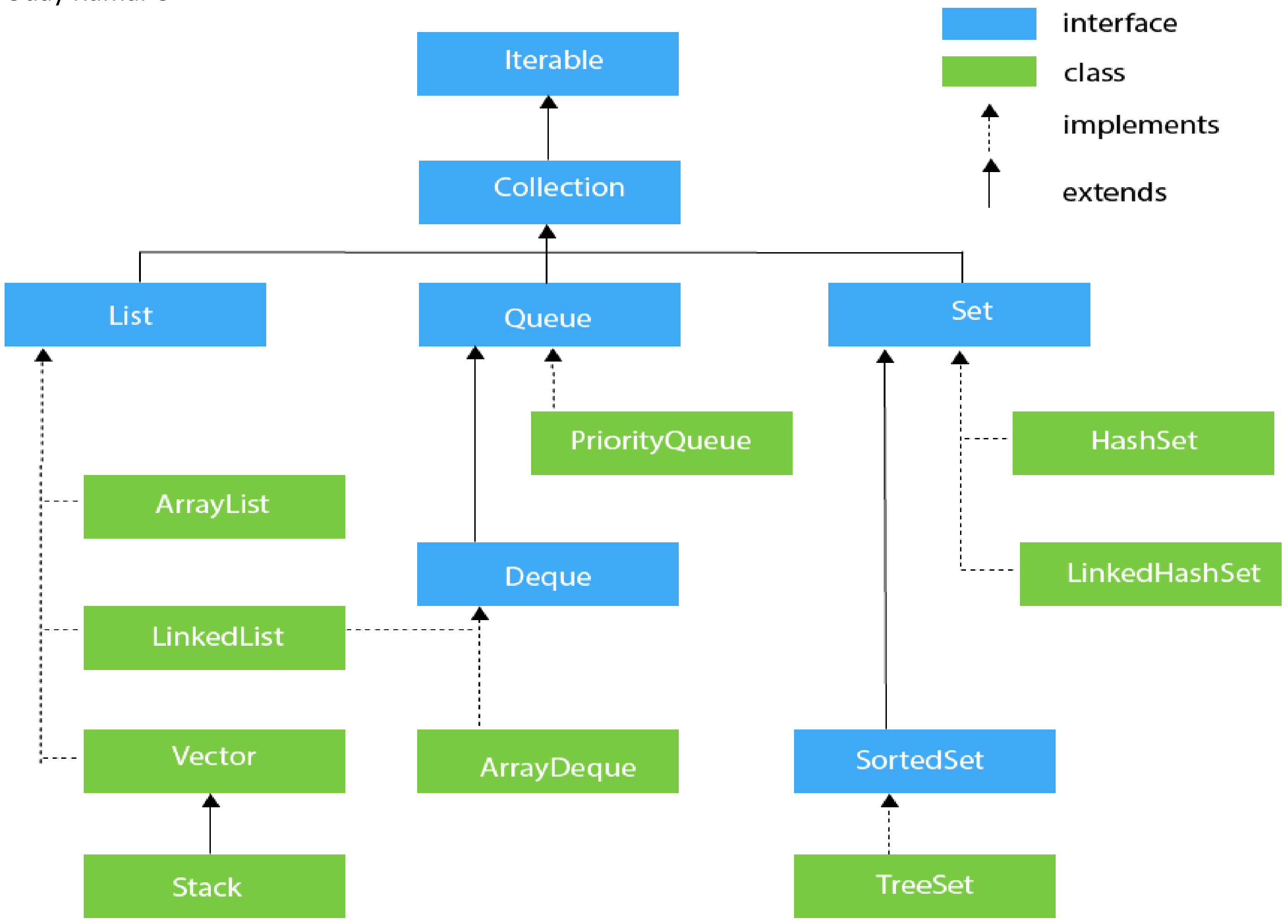
A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

- 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.

```
public class MultipleCatchBlock1 {

    public static void main(String[] args) {

           try{
                int a[]=new int[5];
                a[5]=30/0;
               }
               catch(ArithmeticException e)
                  {
                   System.out.println("Arithmetic Exception occurs");
                  }
               catch(ArrayIndexOutOfBoundsException e)
                  {
                   System.out.println("ArrayIndexOutOfBounds Exception occurs");
                  }
               catch(Exception e)
                  {
                   System.out.println("Parent Exception occurs");
                  }
               System.out.println("rest of the code");
    }
}
```

Java Collection Framework

Uday Kumar S



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).

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.,

```
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.

Uday Kumar S
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 :

```
List <data-type> list1= new ArrayList();
List <data-type> list2 = new LinkedList();
List <data-type> list3 = new Vector();
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.

ArrayList

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.

```
import java.util.*;
class TestJavaCollection1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();//Creating arraylist
list.add("Ravi");//Adding object in arraylist
list.add("Vijay");
list.add("Ravi");
list.add("Ajay");
//Traversing list through Iterator
Iterator itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ravi
Vijay
Ravi
Ajay
```

LinkedList

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.

```
import java.util.*;
public class TestJavaCollection2{
public static void main(String args[]){
LinkedList<String> al=new LinkedList<String>();
al.add("Ravi");
al.add("Vijay");
al.add("Ravi");
al.add("Ajay");
Iterator<String> itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ravi
Vijay
Ravi
Ajay
```

Vector

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.

```
import java.util.*;
public class TestJavaCollection3{
public static void main(String args[]){
Vector<String> v=new Vector<String>();
v.add("Ayush");
v.add("Amit");
v.add("Ashish");
v.add("Garima");
Iterator<String> itr=v.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ayush
Amit
Ashish
Garima
```

Stack

Uday Kumar S
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.

```
import java.util.*;
public class TestJavaCollection4{
public static void main(String args[]){
Stack<String> stack = new Stack<String>();
stack.push("Ayush");
stack.push("Garvit");
stack.push("Amit");
stack.push("Ashish");
stack.push("Garima");
stack.pop();
Iterator<String> itr=stack.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ayush
Garvit
Amit
Ashish
```

Queue Interface

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:

```
Queue<String> q1 = new PriorityQueue();
Queue<String> q2 = new ArrayDeque();
```

There are various classes that implement the Queue interface, some of them are given below.

PriorityQueue

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.

```
import java.util.*;
public class TestJavaCollection5{
public static void main(String args[]){
PriorityQueue<String> queue=new PriorityQueue<String>();
queue.add("Amit Sharma");
queue.add("Vijay Raj");
queue.add("JaiShankar");
queue.add("Raj");
System.out.println("head:"+queue.element());
System.out.println("head:"+queue.peek());
System.out.println("iterating the queue elements:");
Iterator itr=queue.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
queue.remove();
queue.poll();
System.out.println("after removing two elements:");
Iterator<String> itr2=queue.iterator();
while(itr2.hasNext()){
System.out.println(itr2.next());
}
}
}
```

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

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:

```
Deque d = new ArrayDeque();
```

ArrayDeque

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.

Uday Kumar S

```
import java.util.*;
public class TestJavaCollection6{
public static void main(String[] args) {
//Creating Deque and adding elements
Deque<String> deque = new ArrayDeque<String>();
deque.add("Gautam");
deque.add("Karan");
deque.add("Ajay");
//Traversing elements
for (String str : deque) {
System.out.println(str);
}
}
}
```

Output:

```
Gautam
Karan
Ajay
```

Set Interface

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:

```
Set<data-type> s1 = new HashSet<data-type>();
Set<data-type> s2 = new LinkedHashSet<data-type>();
Set<data-type> s3 = new TreeSet<data-type>();
```

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.

```
import java.util.*;
public class TestJavaCollection7{
public static void main(String args[]){
//Creating HashSet and adding elements
HashSet<String> set=new HashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//Traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Vijay
Ravi
Ajay
```

LinkedHashSet

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.

```
import java.util.*;
public class TestJavaCollection8{
public static void main(String args[]){
LinkedHashSet<String> set=new LinkedHashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ravi
Vijay
Ajay
```

SortedSet Interface

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:

```
SortedSet<data-type> set = new TreeSet();
```

TreeSet

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.

Uday Kumar S
Consider the following example:

```
import java.util.*;
public class TestJavaCollection9{
public static void main(String args[]){
//Creating and adding elements
TreeSet<String> set=new TreeSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ajay
Ravi
Vijay
```

Java HashMap

Java HashMap class hierarchy
Java HashHashtable in programs that rely on its thread safety but not on its synchronization 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.

Java ConcurrentHashMap class

A hash table supporting full concurrency of retrievals and high expected concurrency for updates. This class obeys the same functional specification as Hashtable and includes versions of methods corresponding to each method of Hashtable. However, even though all operations are thread-safe, retrieval operations do not entail locking, and there is not any support for locking the entire table in a way that prevents all access. This class is fully interoperable with Hashtable in programs that rely on its thread safety but not on its synchronization details.

Difference between ArrayList and LinkedList

	ArrayList	**LinkedList**
1.	This class uses a dynamic array to store the elements in it. With the introduction	This class uses a doubly linked list to store the elements in it. Similar to the ArrayList, this class also supports the storage of all types of objects.

	of generics, this class supports the storage of all types of objects.	
2.	Manipulating ArrayList takes more time due to the internal implementation. Whenever we remove an element, internally, the array is traversed and the memory bits are shifted.	Manipulating LinkedList takes less time compared to ArrayList because, in a doubly-linked list, there is no concept of shifting the memory bits. The list is traversed and the reference link is changed.
3.	Inefficient memory utilization.	Good memory utilization.
4.	It can be one, two or multi-dimensional.	It can either be single, double or circular LinkedList.
5.	Insertion operation is slow.	Insertion operation is fast.
6.	This class implements a List interface. Therefore, this acts as a list.	This class implements both the List interface and the Deque interface. Therefore, it can act as a list and a deque.
7.	This class works better when the application demands storing the data and accessing it.	This class works better when the application demands manipulation of the stored data.
8.	Data access and storage is very efficient as it stores the elements according to the indexes.	Data access and storage is slow in LinkedList.
9.	Deletion operation is not very efficient.	Deletion operation is very efficient.
10.	It is used to store only similar types of data.	It is used to store any types of data.
11.	Less memory is used.	More memory is used.
12.	This is known as static memory allocation.	This is known as dynamic memory allocation.

14th June 2024

Java 8 SE Features

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.

Uday Kumar S
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.

```
@FunctionalInterface
interface sayable{
   void say(String msg);
}
public class FunctionalInterfaceExample implements sayable{
   public void say(String msg){
      System.out.println(msg);
   }
   public static void main(String[] args) {
      FunctionalInterfaceExample fie = new FunctionalInterfaceExample();
      fie.say("Hello there");
   }
}
```

Output:
```
Hello there
```

Java Optional Class

Java introduced a new class Optional in jdk8. It is a public final class and used to deal with NullPointerException in Java application. You must import java.util package to use this class. It provides methods which are used to check the presence of value for particular variable.

```
import java.util.Optional;
public class OptionalExample {
   public static void main(String[] args) {
      String[] str = new String[10];
      str[5] = "JAVA OPTIONAL CLASS EXAMPLE";  // Setting value for 5th index
      Optional<String> checkNull = Optional.ofNullable(str[5]);
      checkNull.ifPresent(System.out::println);   // printing value by using method reference
      System.out.println(checkNull.get());    // printing value by using get method
      System.out.println(str[5].toLowerCase());
   }
}
```

Output:
```
JAVA OPTIONAL CLASS EXAMPLE
JAVA OPTIONAL CLASS EXAMPLE
java optional class example
```

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.

```
interface Sayable{
   // Default method
   default void say(){
      System.out.println("Hello, this is default method");
   }
   // Abstract method
   void sayMore(String msg);
}
public class DefaultMethods implements Sayable{
   public void sayMore(String msg){       // implementing abstract method
      System.out.println(msg);
   }
   public static void main(String[] args) {
      DefaultMethods dm = new DefaultMethods();
      dm.say();   // calling default method
      dm.sayMore("Work is worship");  // calling abstract method

   }
}
```

Output:

```
Hello, this is default method
Work is worship
```

Java 8 Stream

Java provides a new additional package in Java 8 called java.util.stream. This package consists of classes, 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.

```
import java.util.*;
class Product{
   int id;
   String name;
   float price;
   public Product(int id, String name, float price) {
      this.id = id;
      this.name = name;
      this.price = price;
   }
}
public class JavaStreamExample {
   public static void main(String[] args) {
      List<Product> productsList = new ArrayList<Product>();
      //Adding Products
      productsList.add(new Product(1,"HP Laptop",25000f));
      productsList.add(new Product(2,"Dell Laptop",30000f));
```

```
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));
        List<Float> productPriceList = new ArrayList<Float>();
        for(Product product: productsList){

            // filtering data of list
            if(product.price<30000){
                productPriceList.add(product.price);    // adding price to a productPriceList
            }
        }
        System.out.println(productPriceList);   // displaying data
    }
}
```

Output:

```
[25000.0, 28000.0, 28000.0]
```

Java Date and Time

The java.time, java.util, java.sql and java.text packages contains classes for representing date and time.

Java 8 Date/Time API

Java has introduced a new Date and Time API since Java 8. The java.time package contains Java 8 Date and Time classes.

- java.time.LocalDate class
- java.time.LocalTime class
- java.time.LocalDateTime class
- java.time.MonthDay class
- java.time.OffsetTime class
- java.time.OffsetDateTime class
- java.time.Clock class
- java.time.ZonedDateTime class
- java.time.ZoneId class
- java.time.ZoneOffset class
- java.time.Year class
- java.time.YearMonth class
- java.time.Period class
- java.time.Duration class
- java.time.Instant class
- java.time.DayOfWeek enum
- java.time.Month enum

Classical Date/Time API

But classical or old Java Date API is also useful. Let's see the list of classical Date and Time classes.

- java.util.Date class
- java.sql.Date class
- java.util.Calendar class
- java.util.GregorianCalendar class
- java.util.TimeZone class

- java.sql.Time class
- java.sql.Timestamp class

Formatting Date and Time

We can format date and time in Java by using the following classes:

- java.text.DateFormat class
- java.text.SimpleDateFormat class

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.

Syntax:

```
(argument-list) -> {body}
```

No Parameter Syntax:

```
() -> {
//Body of no parameter lambda
}
```

One Parameter Syntax:

```
(p1) -> {
//Body of single parameter lambda
}
```

Two Parameter Syntax:

```
(p1,p2) -> {
//Body of multiple parameter lambda
}
```

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. In this tutorial, we are explaining method reference concept in detail.

Types of Method References

- **Reference to a static method:**

You can refer to static method defined in the class. Following is the syntax and example which describe the process of referring static method in Java.

Syntax

```
ContainingClass::staticMethodName
```

- **Reference to an instance method:**

like static methods, you can refer instance methods also. In the following example, we are describing the process of referring the instance method.

Syntax

```
containingObject::instanceMethodName
```

- **Reference to a constructor:**

You can refer a constructor by using the new keyword. Here, we are referring constructor with the help of functional interface.

Syntax

```
ClassName::new
```

Java Meta Space

In Java 8, Metaspace is a new memory space that replaced the older "PermGen" (Permanent Generation) space used in previous versions of Java. Metaspace is where the Java Virtual Machine (JVM) stores class metadata. This change was introduced primarily to improve performance and to eliminate the need for Java applications to worry about the often problematic PermGen space limitations.

- Location: Metaspace is a part of the native memory, which includes the operating system's memory, as opposed to the Java heap where application data is stored. This means Metaspace is allocated out of the same system memory as the heap, but it is not part of the heap.
- Function: Metaspace stores:
  - The metadata of classes (information about class declarations such as the number of methods or fields a class has).
  - Interned strings.
  - Static class fields.
  - Other artifacts related to the JVM that are not part of the application data per se.

- Management: Unlike PermGen, which had a fixed size and could cause the infamous OutOfMemoryError if the space was exhausted, Metaspace dynamically resizes based on demand. This flexibility helps prevent applications from crashing due to fixed memory limits. However, it also means that Metaspace could potentially use an excessive amount of system memory if not monitored and limited, which could lead to system issues.
- Configuration: While Metaspace grows dynamically, it can still be capped using JVM options to prevent it from growing excessively. The primary configuration parameters are:
  - `-XX:MetaspaceSize`: This setting defines the initial size of Metaspace. If the space is exhausted, the JVM will resize Metaspace.
  - `-XX:MaxMetaspaceSize`: This sets the maximum size that Metaspace can grow to. Unlike the old PermGen space, this setting is not mandatory in Java 8 and later, and if left unspecified, Metaspace will grow dynamically (limited by the system's available memory).

Multithreading

Multithreading in Java is a core feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Let's dive into each of these concepts related to multithreading to provide a clearer understanding:

Uday Kumar S

Lifecycle of a Thread

A thread in Java can be in one of several states:

New: The thread is created but not yet started.

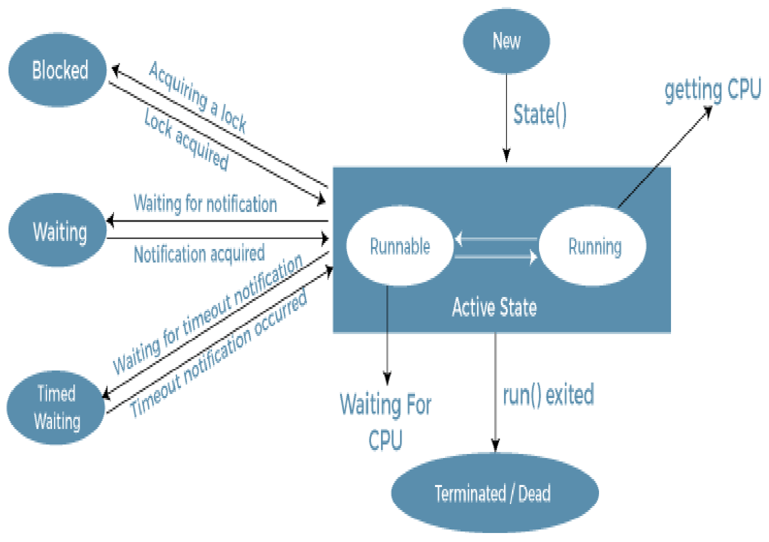
Runnable: The thread is ready to run and is waiting for CPU time.

Blocked: The thread is blocked and waiting for a monitor lock.

Waiting: The thread is waiting indefinitely for another thread to perform a particular action.

Timed Waiting: The thread is waiting for another thread to perform an action up to a specified waiting time.

Terminated: The thread has exited its run method and is complete.



Life Cycle of a Thread

Thread Priority in Multithreading

Each thread in Java is given a priority that helps the operating system determine the order in which threads are scheduled.

Java thread priorities are integers ranging from 1 (least priority) to 10 (highest priority). The default priority is 5.

Runnable Interface in Java

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread.

The class must define a method of no arguments called run.

```
public class MyRunnable implements Runnable {
  public void run() {
    System.out.println("MyRunnable running");
  }
}
```

start() Function in Multithreading

The start() method of the Thread class is used to begin the execution of a thread.

Calling start() causes the JVM to call the run() method of the Thread object on a separate call stack.

Thread.sleep() Method in Java

Thread.sleep(long millis) causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.

Thread.run() in Java

The run() method in a Thread or Runnable object is called to start the execution of the thread’s task. Calling run() directly doesn't start a new call stack but instead runs the method in the current stack.

Deadlock in Java

Deadlock in Java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.

Uday Kumar S

How to Avoid Deadlock in Java?

Deadlocks cannot be completely resolved. But we can avoid them by following basic rules mentioned below:

- Avoid Nested Locks: We must avoid giving locks to multiple threads, this is the main reason for a deadlock condition. It normally happens when you give locks to multiple threads.
- Avoid Unnecessary Locks: The locks should be given to the important threads. Giving locks to the unnecessary threads that cause the deadlock condition.
- Using Thread Join: A deadlock usually happens when one thread is waiting for the other to finish. In this case, we can use join with a maximum time that a thread will take.

Synchronization in Java

Synchronization in Java is the capability to control the access of multiple threads to shared resources. Without synchronization, it is possible for one thread to modify a shared object while another thread is in the process of using or updating that object's value.

Method Level Lock

Synchronized methods allow a method to be accessed by only one thread at a time.

```
public synchronized void increment() {
   count++;
}
```

Block Level Lock

Synchronized blocks allow critical sections of a method to be synchronized.

```
public void add(int value) {
   synchronized(this) {
      count += value;
   }
}
```

Executor Framework in Java

The Executor Framework is a framework provided by the JDK which simplifies the execution of tasks in asynchronous mode.

It provides a pool of threads and API for assigning tasks to the executor.

Callable Interface in Java

The Callable interface is similar to Runnable, except that its call() method can return a result and throw a checked exception.

Callable tasks are submitted to an ExecutorService and return results via Future objects.

```
Callable<Integer> task = () -> {
   return 123;
};
```