Java Notes

Java programming language:

The Java programming language was created by James Gosling and his team at Sun Microsystems in the early 1990s. The language was initially called "Oak," but was later renamed "Java" due to trademark issues.

Java was designed to be a portable, platform-independent language that could be used to develop applications for a wide range of devices and operating systems. It was also intended to be a more secure language than other programming languages, such as C and C++, by using a "sandbox" approach to executing code.

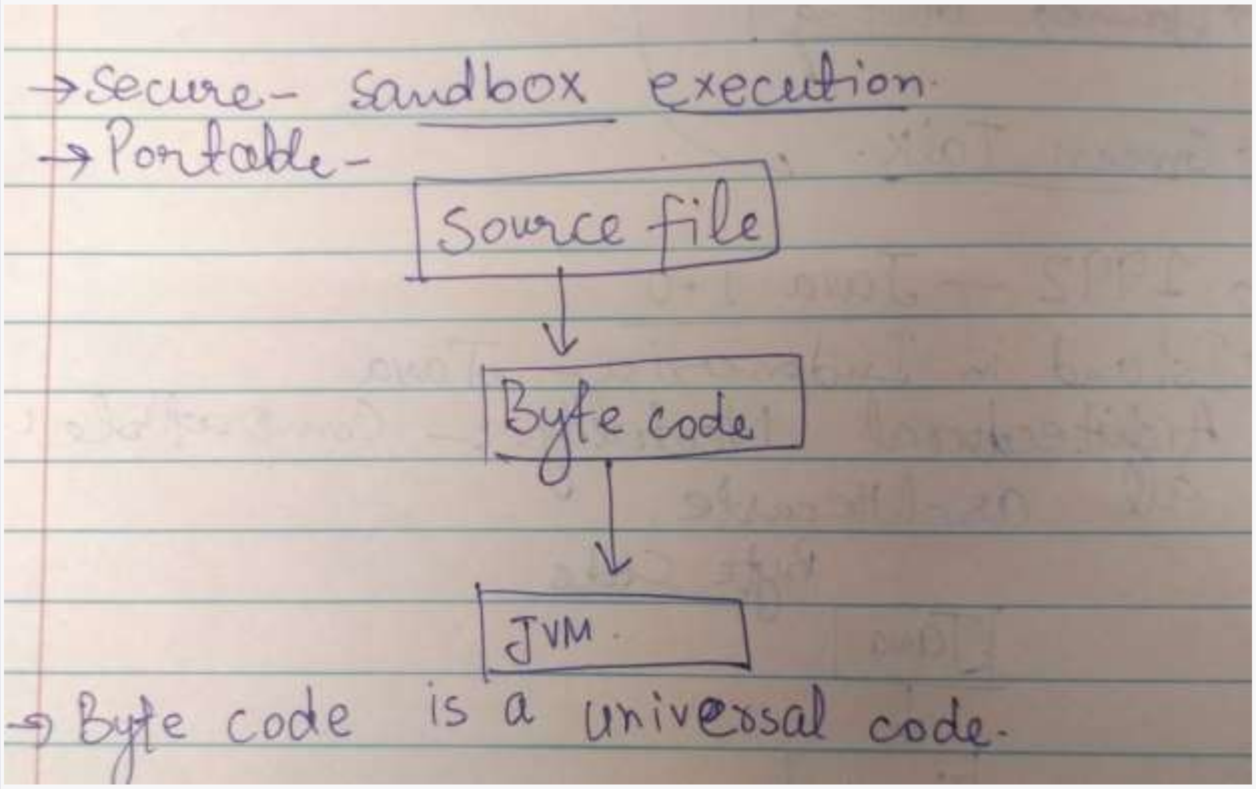
One of the key features of Java is its "write once, run anywhere" capability, which allows Java code to be compiled into bytecode that can be executed on any platform that has a Java Virtual Machine (JVM) installed. This has made Java a popular choice for developing applications that need to run on multiple platforms, such as web applications and mobile apps.

Some of the key developments in the history of Java include the introduction of the Java Development Kit (JDK) in 1996, the release of Java 2 in 1998, the introduction of the Java Platform, Enterprise Edition (Java EE) in 1999, and the release of Java SE 8 in 2014.

Today, Java remains one of the most widely used programming languages in the world, and is used for everything from developing desktop applications to building large-scale enterprise systems. Despite competition from newer languages such as Python and JavaScript, Java's robustness and versatility have ensured that it remains a popular choice for developers around the globe.

Compilation:

In Java, the source code is first compiled into bytecode using a Java compiler (javac). This bytecode is saved in a file with a .class extension, which can be run on any system that has a Java Virtual Machine (JVM) installed. When the .class file is executed, the JVM loads the bytecode into memory and executes it, following the control flow specified by the program. The JVM manages memory allocation, garbage collection, and other low-level details of program execution. The use of bytecode and the JVM allows Java programs to be platform-independent, meaning they can be run on any system that has a compatible JVM installed, without the need to recompile the source code for each platform.



→ Secure- sandbox execution

→ Portable-

$$\begin{tikzcd}
\boxed{\text{Source file}} \arrow[d] \\
\boxed{\text{Byte code}} \arrow[d] \\
\boxed{\text{JVM}}
\end{tikzcd}$$

→ Byte code is a universal code.

Features of Java :

1. Object-Oriented: Java is a fully object-oriented programming language. It supports the principles of encapsulation, inheritance, and polymorphism, allowing developers to create modular, reusable, and maintainable code.

2. Platform Independence: Java programs can run on any operating system or platform that has a Java Virtual Machine (JVM) installed. The "Write once, run anywhere" (WORA) principle enables portability and makes Java highly versatile.

3. Robust and Secure: Java has features like strong type-checking at compile time, automatic memory management (garbage collection), and exception handling, which contribute to creating robust and reliable software. Additionally, Java includes built-in security mechanisms to protect against common vulnerabilities.

4. Rich Standard Library: Java provides a vast collection of classes and APIs in its standard library, offering ready-to-use functionality for tasks such as file I/O, networking, GUI development, data structures, and more. This extensive library saves development time and effort.

5. Multithreading: Java supports multithreading, allowing the execution of multiple threads concurrently within a single program. This feature is crucial for developing efficient, responsive, and concurrent applications

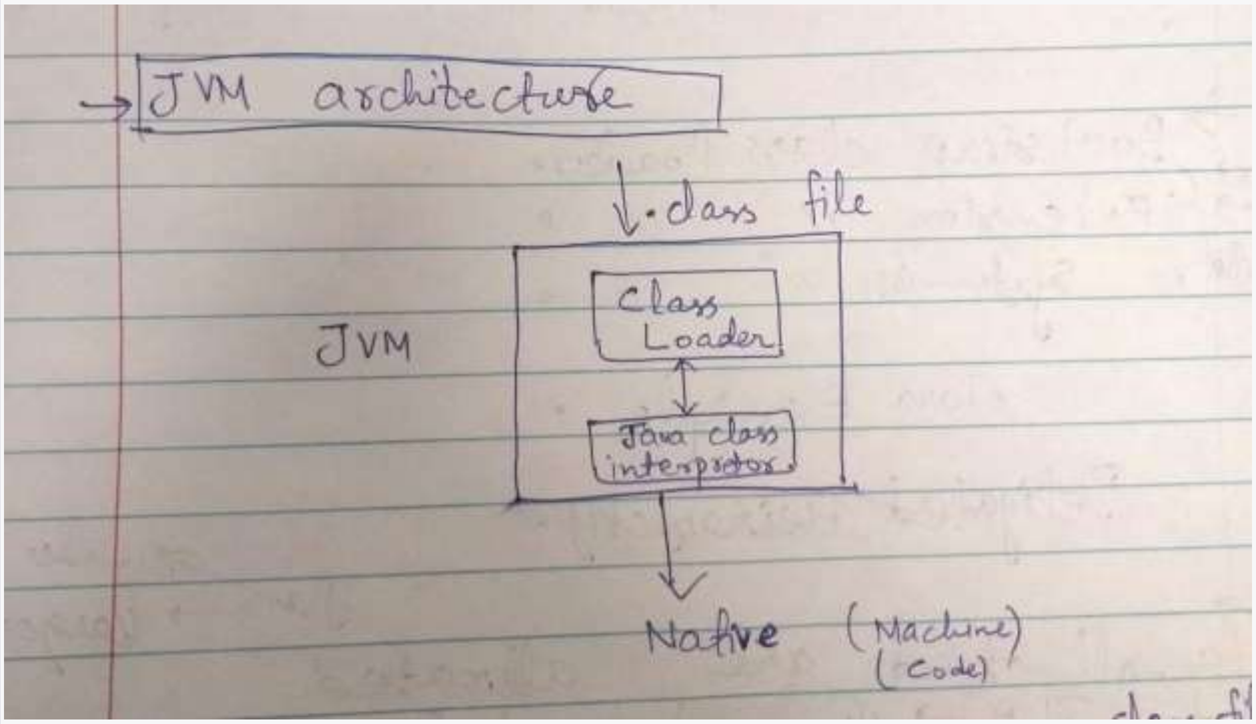
6. High Performance: While Java is not as low-level as languages like C or C++, it still offers good performance due to just-in-time (JIT) compilation and various optimization techniques employed by the JVM.

7. Garbage Collection: Java's automatic memory management system frees developers from managing memory explicitly. The garbage collector automatically identifies and reclaims unused objects, reducing the risk of memory leaks and providing memory efficiency.

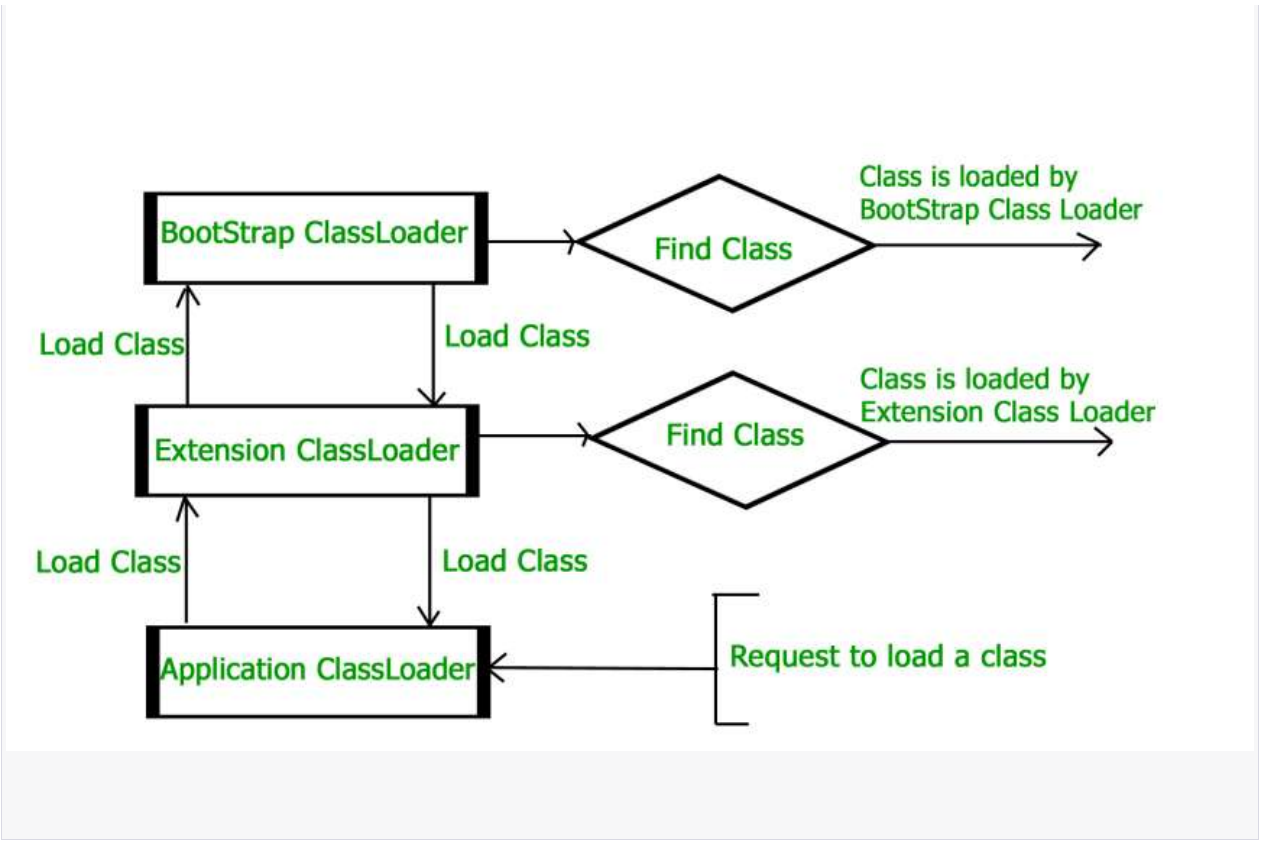
8. Exception Handling: Java provides a robust exception handling mechanism, enabling developers to catch and handle errors and exceptions. This feature promotes graceful error recovery and fault tolerance in applications.

JVM Architecture:

The Java Virtual Machine (JVM) architecture is designed to provide a platform-independent runtime environment for executing Java programs. The architecture can be divided into three main components: the class loader, the runtime data area, and the execution engine. The class loader loads Java class files into the JVM, the runtime data area stores data used during program execution, and the execution engine executes the bytecode instructions that make up the Java program. The execution engine includes an interpreter, a JIT compiler, and a garbage collector. The JVM architecture provides features such as memory management and optimization to improve performance, while also ensuring that Java programs can be executed on any system that has a compatible JVM installed.



Class Loader



The Java ClassLoader is a part of the Java Runtime Environment that dynamically loads Java classes into the Java Virtual Machine. The Java run time system does not need to know about files and file systems because of classloaders. Java classes aren't loaded into memory all at once, but when required by an application. At this point, the Java ClassLoader is called by the JRE and these ClassLoaders load classes into memory dynamically.

A Java Classloader is of three types:

BootStrap ClassLoader: A Bootstrap Classloader is a Machine code which kickstarts the operation when the JVM calls it. It is not a java class. Its job is to load the first pure Java ClassLoader. Bootstrap ClassLoader loads classes from the location rt.jar. Bootstrap ClassLoader doesn't have any parent ClassLoaders. It is also called as the Primordial ClassLoader.

2. Extension ClassLoader: The Extension ClassLoader is a child of Bootstrap ClassLoader and loads the extensions of core java classes from the respective JDK Extension library. It loads files from jre/lib/ext directory or any other directory pointed by the system property java.ext.dirs.

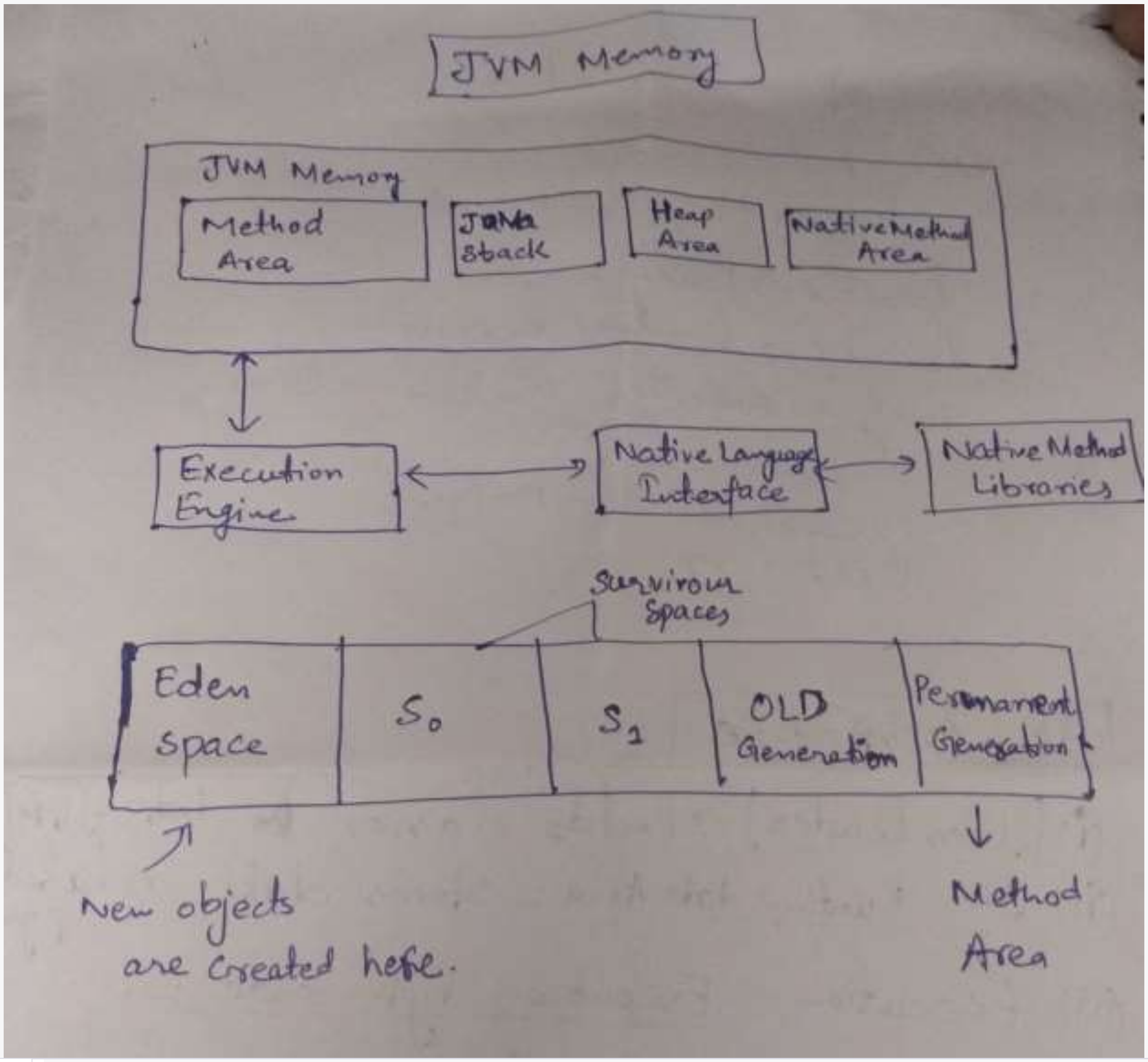
3. System ClassLoader: An Application ClassLoader is also known as a System ClassLoader. It loads the Application type classes found in the environment variable CLASSPATH, -classpath or -cp command line option. The Application ClassLoader is a child class of Extension ClassLoader**.**

JVM Memory:

The Java Virtual Machine (JVM) manages memory allocation and deallocation for Java programs. The JVM memory is divided into several areas, each with a specific purpose. Here's a brief overview of the main memory areas in the JVM:

1. Heap: The heap is where objects and their instance variables are allocated. It is a shared memory area that is used by all threads in the JVM. The heap is divided into two main sections: the young generation and the old generation.
2. Young Generation: The young generation is where newly created objects are allocated. It is divided into two areas: the Eden space and the survivor space. Objects are first allocated in the Eden space, and if they survive a garbage collection cycle, they are moved to one of the survivor spaces.
3. Old Generation: The old generation is where long-lived objects are allocated. Objects that survive multiple garbage collection cycles in the young generation are promoted to the old generation.
4. Method Area: The method area stores class-level data, such as the runtime constant pool and field and method data. It is a shared memory area that is used by all threads in the JVM.

5. Native Method Stack: The native method stack stores information for native methods, which are methods implemented in a language other than Java, such as C or C++. It is separate from the Java stack, which stores information for Java methods.
6. Java Stack: The Java stack stores information for Java methods, including local variables and method parameters.



Garbage Collector:

The JVM memory is managed by a garbage collector, which identifies and removes objects that are no longer in use. The garbage collector runs automatically in the background and frees up memory as needed. The JVM memory management system allows Java programs to run efficiently and effectively, while also ensuring that memory is used efficiently and effectively. The Java Virtual Machine (JVM) heap memory is divided into two main areas: the young generation and the old generation. The young generation is where newly created objects are allocated, while the old generation is where long-lived objects are allocated.

The young generation is further divided into three spaces: Eden, Survivor 1, and Survivor 2. New objects are allocated in the Eden space, and if they survive a garbage collection cycle, they are moved to one of the survivor spaces. The survivor spaces act as a buffer for objects that have not yet lived long enough to be promoted to the old generation.

The old generation, also known as the tenured generation, is where long-lived objects are stored. Objects that survive multiple garbage collection cycles in the young generation are promoted to the old generation. The old generation is typically larger than the young generation and is subject to less frequent garbage collection cycles.

Parallel Markup and sweep:

Parallel Mark and Sweep is a variation of the Mark and Sweep garbage collection algorithm that uses multiple threads to perform the marking and sweeping phases of garbage collection in parallel. This allows the garbage collector to take advantage of multi-core processors and perform garbage collection more quickly and efficiently.

During the marking phase, each thread traverses a portion of the heap to identify all live objects and marks them as reachable. Once the marking phase is complete, each thread processes a portion of the heap during the sweeping phase, removing any objects that are not marked as reachable.

Just in Time Compiler:

JIT Compiler is a type of compiler that compiles the Java bytecode at runtime, which means the compilation occurs while the program is executing. When a Java program is executed, the bytecode is first loaded into memory by the JVM (Java Virtual Machine) and then translated into machine code by the JIT compiler. This machine code is then executed by the CPU. The purpose of the JIT compiler is to improve the performance of the Java application by compiling frequently executed code into native machine code. JIT interprets code line by line, and if it encounters object or method that is repeated regularly (**Hotspot Tech**) , then instead of interpreting that section of code every time JIT compile it and store it in code cache.

Ahead of Time Compiler:

AOT Compiler, on the other hand, is a type of compiler that compiles the Java bytecode into native machine code before the program is executed. This means that the compilation occurs during the build process rather than at runtime. The resulting executable can then be run on a machine that has the same architecture as the one where the compilation occurred. AOT compilation is used mainly for performance-critical applications that require fast startup times and low memory usage.

Keywords

Keywords are reserved words in the Java language that have a specific meaning and cannot be used as variable names or other identifiers. Examples of keywords include "public", "class", "static", and "void".

Identifiers

Identifiers are user-defined names that are used to represent variables, methods, and classes in Java. Identifiers must start with a letter, underscore, or dollar sign, and can contain letters, digits, underscores, or dollar signs.

Literals

Literals are values that are directly represented in the source code of a Java program. There are several types of literals in Java, including integer literals, floating-point literals, boolean literals, character literals, and string literals.

Separators

Separators are symbols in Java that are used to separate different parts of a program. Examples of separators include semicolons (;), commas (,), and parentheses (()).

Comments

Comments are used to add explanatory text to a Java program. Comments can be either single-line or multi-line, and are ignored by the Java compiler.

DATA TYPES:

In Java, there are two categories of data types: primitive types and reference types.

Primitive types are the most basic types in Java, and include:

- boolean: a data type that represents true or false values.
- byte: an 8-bit signed two's complement integer.
- short: a 16-bit signed two's complement integer.
- int: a 32-bit signed two's complement integer.
- long: a 64-bit signed two's complement integer.
- float: a 32-bit single-precision floating-point number.
- double: a 64-bit double-precision floating-point number.
- char: a 16-bit Unicode character.

Reference types are more complex data types that are based on classes or interfaces. They include:

- String: a sequence of characters.
- Arrays: collections of variables of the same type.
- Classes: user-defined types that encapsulate data and behavior.
- Interfaces: collections of abstract methods that can be implemented by classes.

Note:

Java is a **strictly typed language**, which means that every variable and expression in Java has a specific data type that is determined at compile-time. In Java, the type of a variable or expression must be declared explicitly before it can be used. For example, to declare a variable of type int, you would write:

Int x=10;

Auto Type Promotion:

AutoType promotion is a feature in Java that automatically promotes certain data types to a larger, wider data type when necessary. For example, if you try to add an int and a double, the int will be promoted to a double before the addition takes place, since a double can hold more precise values than an int. This promotion happens automatically at compile-time, and can help simplify code and prevent errors related to data type mismatches.

Explicit type conversion

Explicit type conversion, also known as type casting, is a feature in Java that allows a programmer to convert a value from one data type to another. This is sometimes necessary when performing operations that require matching data types, or when assigning a value to a variable of a different type.

```
int num = 10;

double d = (double) num;
```

Array:

In Java, an array is an object that can hold a fixed number of values of the same data type. The elements in an array are stored in contiguous memory locations, and each element can be accessed using an index that represents its position in the array.

```
int[] numbers = new int[5];

int[] numbers = {1, 2, 3, 4, 5};
```

Local variable type inference :

Local variable type inference is a feature in Java that allows a programmer to declare a variable without explicitly stating its data type, and instead have the compiler infer the data type based on the context in which the variable is used. This was introduced in Java 10

For example, instead of declaring a variable with an explicit data type like this:

```
String name = "John";
```

You can use var like this:

```
var name = "John";
```

Note:

`1.var` cannot be used with class names or objects in Java. `var` is used for local variable type inference, which means that it can only be used to declare and initialize local variables within a method or block.

2. Using var we can declare only single variable.

var a=10 (allowed)

var a,b; (Not Allowed)

3.var a=new int[10] (allowed)

4. var a={1,2,3,4} (Not allowed) because due to uncertainty of datatype of array elements like a={1,2,'A,4}

5. var []a=new int[]   (Not allowed)

6. var a=a+5; (Not allowed)

Operators:

int sum = x + y; // addition

int difference = x - y; // subtraction

int product = x * y; // multiplication

int quotient = y / x; // division

int remainder = y % x; // modulus

boolean isEqual = x == y; // equal to

boolean isNotEqual = x != y; // not equal to

boolean isLessThan = x < y; // less than

boolean isGreaterThan = x > y; // greater than

```
boolean isLessThanOrEqual = x <= y; // less than or equal to

boolean isGreaterThanOrEqual = x >= y; // greater than or equal to


boolean isLogicalAnd = (x > 0) && (y < 20); // logical AND

boolean isLogicalOr = (x < 0) || (y > 20); // logical OR

boolean isLogicalNot = !(x == y); // logical NOT


x += 2; // addition assignment

y -= 3; // subtraction assignment

x *= 4; // multiplication assignment

y /= 5; // division assignment

x %= 6; // modulus assignment

x &= 7; // bitwise AND assignment

y |= 8; // bitwise OR assignment

x ^= 9; // bitwise XOR assignment

x <<= 2; // left shift assignment

y >>= 3; // right shift assignment

x >>>= 4; // unsigned right shift assignment


int bitwiseAnd = x & y; // bitwise AND

int bitwiseOr = x | y; // bitwise OR

int bitwiseXor = x ^ y; // bitwise XOR

int bitwiseNotX = ~x; // bitwise NOT

int leftShift = x << 2; // left shift

int rightShift = y >> 3; // right shift
```

int unsignedRightShift = x >>> 4; // unsigned right shift

<< (left shift): Takes two numbers as operands and shifts the bits of the first operand to the left by the number of positions specified by the second operand. The empty positions on the right are filled with zeroes.

>> (signed right shift): Takes two numbers as operands and shifts the bits of the first operand to the right by the number of positions specified by the second operand. The empty positions on the left are filled with the sign bit (the most significant bit) of the original value.

>>> (unsigned right shift): Takes two numbers as operands and shifts the bits of the first operand to the right by the number of positions specified by the second operand. The empty positions on the left are filled with zeroes, regardless of the sign bit.

Note

If(1){ doesn't mean true in java

Loops:

In Java, there are three types of loops: for, while, and do-while. These loops are used to repeat a block of code until a certain condition is met.

The for loop is used when you know the number of times you want to repeat a block of code. It consists of an initialization statement, a boolean condition, and an update statement, all separated by semicolons. The block of code is enclosed in curly braces.

Copy code

```
for (initialization; condition; update) {

  // code to be repeated

}
```

Or

```
Int nums[]={1,2,3,4,5}

for(int x:nums){


}
```

The while loop is used when you want to repeat a block of code until a certain condition is met. It consists of a boolean condition and a block of code enclosed in curly braces. The code will be repeated as long as the condition is true.

Copy code

```
while (condition) {

  // code to be repeated

}
```

The do-while loop is similar to the while loop, except that the block of code is executed at least once before the condition is checked. It consists of a block of code enclosed in curly braces, followed by the do keyword, and then the condition and a semicolon.

```
do {

  // code to be repeated

} while (condition);
```

Within the loop, you can use the break keyword to exit the loop prematurely, and the continue keyword to skip to the next iteration of the loop. You can also use nested loops to repeat a block of code within another loop.

It's important to use loops effectively and efficiently in your code, as they can impact the performance and readability of your program. You should also be careful to avoid infinite loops, where the condition is always true and the loop never exits.

Enumeration

In Java, an enumeration is a type that consists of a fixed set of constant values. Enumerations are used to represent a group of related constants that have a known and limited range. The constants in an enumeration are called its enumerators.

```
enum DayOfWeek {

  SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY

}
```

In Java, an enumeration is a type that consists of a fixed set of constant values. Enumerations are used to represent a group of related constants that have a known and limited range. The constants in an enumeration are called its enumerators.

Here's an example of how to define an enumeration in Java:

```
enum DayOfWeek {

  SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY

}
```

In this example, DayOfWeek is the name of the enumeration, and SUNDAY, MONDAY, etc. are the enumerators. Note that the enumerators are defined in all uppercase letters by convention.

You can use an enumeration in your code to represent a specific set of values, like days of the week or directions. You can also define methods and properties on an enumeration, just like any other class in Java.

code

```
DayOfWeek today = DayOfWeek.TUESDAY;

System.out.println("Today is " + today);
```

Strings:

Strings in Java are a sequence of characters enclosed in double quotes, for example:

```
String greeting = "Hello, world!";
```

Strings are immutable, which means once created, their value cannot be changed. However, you can create a new string by concatenating two or more strings together using the + operator or the concat method:

Strings in java are immutable because java supports multi threading

```
String str1 = "Hello";

String str2 = "world";
String greeting = str1 + ", " + str2 + "!"; // "Hello, world!"

String fullGreeting = str1.concat(", ").concat(str2).concat("!"); // "Hello, world!"
```

You can access individual characters in a string using the charAt method:

```
char firstChar = greeting.charAt(0); // 'H'

char lastChar = greeting.charAt(greeting.length() - 1); // '!'
```

You can compare two strings for equality using the equals method:

```
String str1 = "Hello";

String str2 = "hello";

boolean isEqual = str1.equals(str2); // false
```

String comparison is case-sensitive, so the above example returns false. To perform a case-insensitive comparison, you can use the equalsIgnoreCase method:

```
String str1 = "Hello";

String str2 = "hello";

boolean isEqual = str1.equalsIgnoreCase(str2); // true
```

You can also check if a string contains a substring using the contains method:

```
String str = "The quick brown fox jumps over the lazy dog";

boolean containsFox = str.contains("fox"); // true

boolean containsCat = str.contains("cat"); // false
```

String manipulation is a fundamental part of Java programming, and there are many other useful methods and techniques for working with strings.

```java
import java.util.Iterator;
import java.io.*;
import java.util.Scanner;

public class lab3 {
    public static void replaceString(String str1,String str2){
        //replace method
        System.out.println("Replace Method");
        System.out.println(str1);
        System.out.println("Replace l with p");
        System.out.println(str1.replace('l', 'p'));

    }
    public static void compare(String str1,String str2){
        //compare strings
        System.out.println("Compare  Method");
        if(str1.compareTo(str2)==1){
            System.out.println("Content of two Strings is same");
        }
        else{
            System.out.println("Content of two Strings is different");
        }
    }
   static void substringMethod(String str){
       System.out.println("Substring Method");
       String s=str.substring(1,4); //1 included and 4 excluded here
       System.out.println(s);
   }
    static void IndexMethod(String str){
        System.out.println("Index Method");
        int index=str.indexOf('o');
        System.out.println(index);

    }
    static void CodePointMethod(String str){
        System.out.println("Codepoint Method");
        System.out.println(str.codePointAt(1)); // returns the code point(Unicode)
of the character at index 1

    }
    static void CharAtMethod(String str){
        System.out.println("CharAt Method");
        System.out.println(str.charAt(2));

    }
    static void lengthMethod(String str){
        System.out.println("length Method");
        System.out.println(str.length());

    }
    static void trueLengthMethod(String str){
        System.out.println("true Length Method");
        System.out.println(str.codePointCount(1,4)); // returns the number of
code points in the entire string
/* String str = "Hello, ☺!";
```

```
The length of this string is 10, since it contains 10 char values. However,
the string actually contains 11 Unicode code points, because the smiley face
character is represented by two char values.*/
    }
    static void suffixMethod(String str){
        System.out.println("suffix Method");
       if(str.endsWith("World")){
           System.out.println("suffix found");
       }

    }
    static void upperCaseMethod(String str){
        System.out.println("Upper Case  Method");
        System.out.println(str.toUpperCase());

    }
    static void lowerCaseMethod(String str){
        System.out.println("Lower Case  Method");
        System.out.println(str.toLowerCase());

    }
    static void charSequenceMethod(String str){
        System.out.println("charSequence   Method");
        System.out.println("charSequence length");
        System.out.println(str.length());
        System.out.println("charSequence lowercase");
        System.out.println(str.toLowerCase());
        System.out.println("charSequence uppercase");
        System.out.println(str.toUpperCase());
        System.out.println("Convert charseq to string");
        String s=str.toString();
        System.out.println(s);

    }
    static void CountRows(){
        System.out.println("Count Rows");
       int [][] arr=new int[10][10];
       int count=0;
       for(int x[]:arr){
           count++;

       }
        System.out.println(count);

    }
    static void ImmutableMethod(String str){
        System.out.println("Immutable ");
        String s=str.replace('e','d');
        System.out.println(str);
        System.out.println("New string");
        System.out.println(s);
    }
    public static void main(String[] args){
        String str1="Hello World";
        String str2="Java lab";
        String str3="Hello World";
        replaceString(str1,str2);
        compare(str1,str2);
        compare(str1,str3);
        substringMethod(str1);
        IndexMethod(str1);
        CodePointMethod(str1);
        CharAtMethod(str1);
        lengthMethod(str1);
        trueLengthMethod(str1);
        suffixMethod(str1);
        upperCaseMethod(str2);
        lowerCaseMethod(str2);
        String str4="This is a char sequence";
        charSequenceMethod(str4);
        CountRows();
       ImmutableMethod(str1);
       //has next
        String x="moin";
        Scanner sn=new Scanner(x);
        while(sn.hasNext()){
            System.out.println(sn.hasNext());
            sn.next();
        }

    }

}
```

CharSequence

In Java, CharSequence and String are related but distinct concepts.

String is a class in Java that represents an immutable sequence of characters. Once a String object is created, its value cannot be changed. Strings are often used to represent textual data, such as names, addresses, and messages.

CharSequence, on the other hand, is an interface that defines a sequence of characters that can be manipulated. It is implemented by several classes in Java, including String, StringBuilder, and StringBuffer. Unlike String, CharSequence is not a concrete class, but rather an interface that specifies a set of methods that can be used to manipulate character sequences.

One key difference between CharSequence and String is mutability. String objects are immutable, which means that their values cannot be changed after they are created. In contrast, classes that implement CharSequence, such as StringBuilder and StringBuffer, are mutable and allow you to modify their contents.

Another difference is that CharSequence provides a more general interface for working with character sequences, since it can be implemented by multiple classes. This makes it easier to write code that can work with a variety of different types of character sequences.

Here is an example that illustrates the difference between String and CharSequence:

```
String str = "Hello, world!";

StringBuilder sb = new StringBuilder(str);

CharSequence cs = sb.subSequence(0, 5);

CharSequence charseq = "Hello World";

System.out.println(charseq.length()); // Output: 11

System.out.println(charseq.charAt(0)); // Output: 'H'

System.out.println(charseq.subSequence(0, 5)); // Output: "Hello"
```

Note:

Since CharSequence is an interface in Java, you cannot create an instance of it directly. However, you can create instances of classes that implement the CharSequence interface, such as String, StringBuilder, and StringBuffer. Once you have an instance of a class that implements CharSequence, you can use the methods defined in the interface to manipulate the character sequence

charSequence sequence=new charSequence() is wrong as charSequence is not a class but interface

string compareTo:

```
String str1 = "apple";

String str2 = "banana";

String str3 = "Apple";



int result1 = str1.compareTo(str2); // returns a negative value because "apple" is less than "banana"

int result2 = str2.compareTo(str1); // returns a positive value because "banana" is greater than
"apple"
```

int result3 = str1.compareTo(str3); // returns a positive value because 'a' (97) is greater than 'A' (65)

String Builder :

In Java, a StringBuilder is a mutable sequence of characters. It is similar to a String, but it can be modified, whereas a String is immutable.

A StringBuilder object is created with an initial capacity, which is the number of characters it can hold without needing to allocate more memory. The capacity can be increased as needed when new characters are added to the StringBuilder. The StringBuilder class provides a number of methods for appending, inserting, and deleting characters from the sequence, as well as methods for retrieving substrings and other useful operations.

Here's an example of how to create and use a StringBuilder:

```
StringBuilder sb = new StringBuilder("hello"); // create a StringBuilder with initial value "hello"

sb.append(" world"); // append " world" to the StringBuilder

sb.insert(5, ","); // insert a comma after the "o" in "hello"

sb.deleteCharAt(7); // delete the space after the "o" in "world"

String str = sb.toString(); // convert the StringBuilder to a String

System.out.println(str); // prints "hello,world"
```

Classes:

A class is a blueprint or a template that describes the behavior and state of an object.
It defines the methods, variables, and other properties of an object.
In Java, you can create a class by using the 'class' keyword followed by the class name.
Example:

```
public class Car {
  String make;
  String model;
  int year;
}
```

Objects:

An object is an instance of a class.
It represents a real-world entity and can have its own state and behavior.
In Java, you can create an object of a class by using the 'new' keyword followed by the class name.
Example:

```
Car myCar = new Car();
myCar.make = "Toyota";
myCar.model = "Corolla";
myCar.year = 2023;
```

Inbuilt classes:

Java provides a set of inbuilt classes that can be used without creating objects.
Examples of inbuilt classes in Java are Math, String, and Arrays.
Example:

```
double x = Math.sqrt(25);
String str = "Hello";
int[] nums = {1, 2, 3, 4, 5};
Arrays.sort(nums);
```

Constructors:

A constructor is a special method that is used to initialize objects.
It has the same name as the class and does not have a return type.
You can create multiple constructors for a class with different parameters.
Example:

```
public class Car {
  String make;
  String model;
  int year;

  public Car() {
    make = "Unknown";
    model = "Unknown";
    year = 0;
  }

  public Car(String make, String model, int year) {
    this.make = make;
    this.model = model;
```

```
      this.year = year;
   }
}
```

Default constructor:

If you do not explicitly define a constructor for a class, Java provides a default constructor.
The default constructor has no parameters and initializes all instance variables to their default values (0 for numerical types, false for boolean, and null for reference types).

```
public class MyClass {
   int x;
   String str;

   // Default constructor
   public MyClass() {
      // x and str are initialized to their default values automatically
   }
}
```

Overloaded constructors:

You can define multiple constructors for a class with different parameters. This is called constructor overloading.
When you create an object of a class, you can call the constructor with the appropriate parameters to initialize its instance variables.

```
public class Car {
   String make;
   String model;
   int year;

   // Default constructor
   public Car() {
      make = "Unknown";
      model = "Unknown";
      year = 0;
   }

   // Parameterized constructor
   public Car(String make, String model, int year) {
      this.make = make;
      this.model = model;
```

```
        this.year = year;
    }
}
```

Using 'this' keyword:

In Java, 'this' is a keyword that refers to the current object instance.
You can use 'this' to refer to instance variables and methods of the current object.

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

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

    public void printDetails() {
        System.out.println("Name: " + this.name);
        System.out.println("Age: " + this.age);
    }
}
```

Date CLASS

The Date class in Java is used to work with dates and times. It is a part of the java.util package and provides a number of methods for working with dates, times, and time zones.

To create a Date object, you can use the default constructor, which creates a new Date object representing the current date and time:

```
Date date = new Date();
```

You can also create a Date object by passing the number of milliseconds since January 1, 1970, 00:00:00 GMT to the constructor:

```
Date date = new Date(1000000000000L);
```

This will create a Date object representing the date and time of 01:46:40 GMT on September 9, 2001.

Here are some other important methods of the Date class:

getTime(): Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by this Date object.

toString(): Returns a string representation of this Date object.

compareTo(Date anotherDate): Compares this Date object to another Date object.

equals(Object obj): Indicates whether some other object is "equal to" this one.

before(Date when): Tests if this Date object is before the specified Date object.

after(Date when): Tests if this Date object is after the specified Date object.

```java
import java.util.Date;

public class lab4_a{
    public static void main(String[] args) {

        // Create a Date object representing the current date and time
        Date now = new Date();
        System.out.println("Current date and time: " + now.toString());

        // Create a Date object representing a specific date and time
        Date date1 = new Date(1000000000000L);
        System.out.println("Date and time of 1000000000000L: " +
date1.toString());

        // Use getTime() to get the number of milliseconds since January 1, 1970,
00:00:00 GMT
        long milliseconds = now.getTime();
        System.out.println("Milliseconds since January 1, 1970, 00:00:00 GMT: " +
milliseconds);

        // Use compareTo() to compare two Date objects
        Date date2 = new Date(2000000000000L);
        int comparison = date1.compareTo(date2);
        if (comparison < 0) {
            System.out.println(date1.toString() + " is before " +
date2.toString());
        } else if (comparison > 0) {
            System.out.println(date1.toString() + " is after " +
date2.toString());
        } else {
            System.out.println(date1.toString() + " is equal to " +
date2.toString());
        }

        // Use equals() to test if two Date objects are equal
        Date date3 = new Date(1000000000000L);
        if (date1.equals(date3)) {
            System.out.println(date1.toString() + " is equal to " +
date3.toString());
        } else {
            System.out.println(date1.toString() + " is not equal to " +
date3.toString());
        }

        // Use before() and after() to test if one Date object is before or after
another
        if (date1.before(date2)) {
            System.out.println(date1.toString() + " is before " +
date2.toString());
        }
        if (date2.after(date1)) {
            System.out.println(date2.toString() + " is after " +
date1.toString());
        }
    }
}
```

Gregorian Calendar

```
import java.util.*;

public class lab4_b {
    public static void main(String[] args) {
        // Create a GregorianCalendar object for the current date and time
        Calendar calendar = new GregorianCalendar();
        System.out.println("Current date and time: " + calendar.getTime());

        // Set the year, month, and day
        calendar.set(Calendar.YEAR, 2022);
        calendar.set(Calendar.MONTH, Calendar.APRIL);
        calendar.set(Calendar.DAY_OF_MONTH, 10);
        System.out.println("New date: " + calendar.getTime());
    }
}
```

Java.Time

Java 8 introduced a new date and time API called **java.time** to address the shortcomings of the previous **java.util.Date** and **java.util.Calendar** classes. The **java.time** package provides a set of classes to work with date and time in a more convenient and flexible way.

Some of the key classes in **java.time** package are:

1. **LocalDate**: Represents a date (year, month, day) without time.
2. **LocalTime**: Represents a time (hour, minute, second, and fraction of a second) without date.
3. **LocalDateTime**: Represents a date and time (year, month, day, hour, minute, second, and fraction of a second) without time zone information.
4. **ZonedDateTime**: Represents a date and time with time zone information.
5. **Instant**: Represents a point in time (the number of seconds and nanoseconds since the Unix epoch).
6. **Duration**: Represents a time interval between two points in time.
7. **Period**: Represents a time period between two dates.

```java
import java.time.*;

public class lab4 {
    public static void main(String[] args) {
        // Get the current date and time
        LocalDateTime now = LocalDateTime.now();
        System.out.println("Current date and time: " + now);

        // Create a LocalDate object
        LocalDate date = LocalDate.of(2022, Month.APRIL, 10);
        System.out.println("Date: " + date);
        // Create a LocalTime object
        LocalTime time = LocalTime.of(12, 30, 0);
        System.out.println("Time: " + time);

        // Create a LocalDateTime object
        LocalDateTime datetime = LocalDateTime.of(date, time);
        System.out.println("Datetime: " + datetime);

        // Create a ZonedDateTime object
        ZoneId zoneId = ZoneId.of("America/New_York");
        ZonedDateTime zonedDateTime = ZonedDateTime.of(datetime, zoneId);
        System.out.println("ZonedDateTime: " + zonedDateTime);

        // Create an Instant object
        Instant instant = Instant.now();
        System.out.println("Instant: " + instant);

        // Create a Duration object
        Duration duration = Duration.between(datetime, now);
        System.out.println("Duration: " + duration);

        // Create a Period object
        Period period = Period.between(date, LocalDate.now());
        System.out.println("Period: " + period);
    }
}
```

User Defined Classes :

1. User-defined classes in Java are classes that are created by the programmer. They are used to represent real-world objects or concepts in code.
2. To create a user-defined class, you must first declare it using the "class" keyword followed by the class name. For example:

```java
public class Person {

  // class body goes here

}
```

```
public class Person {

  String name;

  int age;

  // other fields go here

}
```

4. You can also define methods inside the class body to define the behavior of the object. For example:

```
public class Person {

  String name;

  int age;



  public void sayHello() {

    System.out.println("Hello, my name is " + name);

  }

}

Person person1 = new Person();

person1.name = "John";

person1.age = 30;
```

8. User-defined classes can also have constructors, which are special methods that are called when an object is created. Constructors can be used to initialize the object's fields. For example:

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

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

  public void sayHello() {
    System.out.println("Hello, my name is " + name);
  }
}
Person person2 = new Person("Jane", 25);
Person[] people = new Person[2];
people[0] = new Person("John", 30);
people[1] = new Person("Jane", 25);
```

Access and mutator in classes

Accessors and mutators, also known as getters and setters, are methods in a class that provide access to the private variables of the class.

Accessors (Getters)
Accessors are methods that allow us to access the value of a private variable in a class. They typically have the prefix "get" followed by the name of the variable they are getting. Accessors are useful when we want to retrieve the value of a private variable from outside the class.

```
public class Person {
    private String name;

    public String getName() {
        return name;
    }
}
```
In this example, the getName() method allows us to retrieve the value of the private variable "name". We can call this method from outside the class to retrieve the value of the name variable.

Mutators (Setters)

Mutators are methods that allow us to set the value of a private variable in a class. They typically have the prefix "set" followed by the name of the variable they are setting. Mutators are useful when we want to change the value of a private variable from outside the class.

```
public class Person {
    private String name;

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

In this example, the setName() method allows us to set the value of the private variable "name". We can call this method from outside the class to set the value of the name variable.

By providing accessors and mutators in our classes, we can control the access to the private variables of our class and enforce encapsulation. Encapsulation is the concept of hiding the internal workings of a class from the outside world and only allowing access to it through a well-defined interface.

Mutable objects

Mutable objects are objects whose state can be changed after they are created. In other words, mutable objects are objects that have one or more mutable fields. Examples of mutable objects in Java include arrays, ArrayLists, StringBuilder, and HashMap.

```
public class Person {
    private StringBuilder name;

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

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

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

In this example, the Person class has a mutable field called "name". The setName() method allows us to change the value of the "name" field after a Person object has been created.

Mutable objects can be useful in certain situations, but they can also lead to bugs and unexpected behavior if not used carefully. When working with mutable objects, it's important to keep track of their state and make sure that they are not modified in unexpected ways.

One way to avoid the problems associated with mutable objects is to use immutable objects instead. Immutable objects are objects whose state cannot be changed after they are created. Examples of immutable objects in Java include String, Integer, and LocalDate. By using immutable objects, we can avoid the need to worry about their state changing unexpectedly.

Immutable Objects

Immutable objects are objects whose state cannot be changed after they are created. One example of an immutable object in Java is the LocalDate class.

The LocalDate class represents a date in the ISO-8601 calendar system, such as "2023-04-15". Once a LocalDate object is created, its value cannot be changed. Instead, operations on LocalDate objects return new objects with updated values. For example:

```
LocalDate date1 = LocalDate.of(2023, 4, 15);
LocalDate date2 = date1.plusDays(1);
```

In this example, we create a LocalDate object called "date1" with the value "2023-04-15". We then call the plusDays() method on date1 to create a new LocalDate object called "date2" with the value "2023-04-16". Note that date1 is not modified by this operation - instead, a new object is created with the updated value.

The LocalDate class is an example of an immutable object because its state cannot be changed once it is created. This makes it safe to use in multi-threaded environments and helps to prevent bugs related to unexpected changes in object state.

When working with immutable objects like LocalDate, it's important to keep in mind that each operation on the object will return a new object with a new value. This can result in increased memory usage if many objects are created, so it's important to use them judiciously and only when necessary.

Public Method:

A public method is a method in a Java class that can be accessed from any other class in the same package or in a different package.

The Math class in Java has many public methods, such as the sqrt() method, which returns the square root of a number:

```
double result = Math.sqrt(25);
```

In this example, we call the sqrt() method from the Math class to calculate the square root of 25.

The String class in Java has many public methods as well, such as the length() method, which returns the length of a string:

```
String str = "Hello, world!";
int length = str.length();
```

In this example, we call the length() method from the String class to get the length of the string "Hello, world!".

Another example of a public method is the append() method in the StringBuilder class, which appends a string to the end of a StringBuilder object:

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(", world!");
String result = sb.toString();
```

In this example, we create a StringBuilder object called "sb" with the value "Hello". We then call the append() method to append the string ", world!" to the end of the StringBuilder object. Finally, we call the toString() method to convert the StringBuilder object to a String.

Public methods are an important part of Java programming because they allow us to create reusable code that can be called from anywhere in our program. By making methods public, we can share our code with others and make it easier to maintain and update in the future.

Private Methods:

A private method in Java is a method in a class that can only be accessed from within the same class. Private methods are not visible or accessible from other classes, even if they are in the same package.

```
public class MyClass {
    private int x;

    public MyClass(int x) {
        this.x = x;
    }

    public int getX() {
        return x;
    }

    private void setX(int x) {
        this.x = x;
    }
}
```

In this example, we have a class called "MyClass" with a private instance variable called "x". We also have a public method called "getX()" that returns the value of "x". However, we also have a private method called "setX(int x)" that sets the value of "x" to a new value.

Because "setX()" is a private method, it can only be called from within the same class. This means that other classes cannot directly modify the value of "x" - they must do so indirectly by calling a public method that in turn calls "setX()".

Private methods are useful in Java programming because they allow us to encapsulate the implementation details of a class. By making certain methods private, we can ensure that other

classes cannot access or modify the internal state of our objects directly. This can help to prevent bugs and improve the maintainability of our code.
A use case for a private method in Java might be in a banking application where we have a class representing a customer's bank account. The account class might have a private method called "calculateInterest()" that calculates the interest earned on the account

The "calculateInterest()" method would be private because we don't want other classes to be able to call it directly - it should only be called internally by the account class itself. Instead, we would provide a public method called "addInterest()" that would call "calculateInterest()" and add the interest to the account balance.

Static Method

A static method in Java is a method that belongs to a class rather than to an instance of the class. This means that a static method can be called without creating an object of the class, and it can be called using the class name instead of an instance variable

```
public class MathUtils {
    public static int max(int a, int b) {
        return (a > b) ? a : b;
    }
}
```
In this example, we have a class called "MathUtils" with a static method called "max()". This method takes two integers as parameters and returns the larger of the two.

To call the "max()" method, we don't need to create an object of the "MathUtils" class. Instead, we can call the method directly using the class name:

```
int a = 10;
int b = 20;
int result = MathUtils.max(a, b);
```
In this example, we call the "max()" method from the "MathUtils" class with the parameters "a" and "b". The method returns the larger of the two, which is assigned to the variable "result".

Static methods are useful in Java programming for utility classes, where we have a collection of methods that are not associated with any particular instance of a class. By making methods static, we can call them without needing to create objects of the class, which can make our code more efficient and easier to read
.
Note:

In a static method, you cannot directly access non-static (instance) variables of a class, because the static method does not have a reference to any instance of the class. However, you can

access non-static variables of a class by creating an instance of the class within the static method.

```
public class MyClass {
    private int x;

    public static void doSomething() {
        // Cannot access x directly in a static method
        // But we can create an instance of MyClass and access x through that instance
        MyClass instance = new MyClass();
        instance.x = 10;
        System.out.println(instance.x);
    }
}


class emp {
  public:
    static int id;
    string name;
  public:
    emp(string name) {
      this->name = name;
      id++;
    }
};

int main() {
  emp *e1 = new emp("harry");
  emp *e2 = new emp("marry");
  cout << emp.id << endl; // Output: 2
  return 0;
}
```

Factory Method

In Java, a factory method is a static method that returns an instance of a class. The purpose of a factory method is to provide a centralized and flexible way to create objects, which can be especially useful when the process of creating an object is complex or when there are many different variations of objects that can be created.

Here is an example of a factory method:

```
public class AnimalFactory {
    public static Animal createAnimal(String type) {
        if (type.equals("dog")) {
            return new Dog();
        } else if (type.equals("cat")) {
            return new Cat();
        } else {
            throw new IllegalArgumentException("Unknown animal type: " + type);
        }
    }
}
```

Here's what each part of the code does:

Defines a public class called "AnimalFactory"
Defines a public static method called "createAnimal" that returns an "Animal" object and takes a "type" string parameter
If the "type" parameter is "dog", creates and returns a new instance of the "Dog" class
If the "type" parameter is "cat", creates and returns a new instance of the "Cat" class
If the "type" parameter is something else, throws an exception with an error message
Overall, the "AnimalFactory" class provides a centralized way to create instances of different animal classes (e.g. "Dog" and "Cat"). By calling the "createAnimal" method with a specific type, the caller can get an instance of the appropriate animal class without needing to know the details of how the animal object is created.

`Final :`

`final` is a keyword that can be used to modify the behavior of a variable, method, or class.

When used with a variable, `final` indicates that the value of the variable cannot be changed once it has been assigned a value. This creates a constant that can be used throughout the program without the risk of accidentally modifying its value.

For example, the following code declares a final variable `PI` and assigns it the value of 3.14:

```
final double=3.14;
```

When used with a method, `final` indicates that the method cannot be overridden by any subclass. This is particularly useful for preventing accidental modification of critical behavior of a class.

When used with a class, `final` indicates that the class cannot be subclassed, meaning that it cannot be extended to create a new class.

Overall, `final` can help improve the security, maintainability, and performance of a Java program by ensuring that certain variables, methods, and classes cannot be modified or overridden.

Note : 1. final StringBuffer a=”hello” (Not Allowed) (because putting immutable object directly into mutable object is not allowed)

2.final StringBuffer b=new StringBuffer();

a=b; (Not allowed because if final is used before mutable or immutable object then you can’t change refrence of an object)

Null Field Instant:

In Java, when an object is instantiated, its fields are initialized with default values. For primitive data types like **int**, **double**, **boolean**, etc., the default value is 0 or false. However, for object references, the default value is **null**.

This means that if a class has a field that is an object reference and no value is assigned to it during instantiation, it will be initialized with **null**.

For example, consider the following class:

```
public class Person {

private String name;

private int age;

}
```

In this class, **name** is a **String** reference and **age** is an **int**. During instantiation of a **Person** object, **age** will be initialized with the default value of 0, while **name** will be initialized with **null**.

It is important to note that if a **NullPointerException** is thrown when trying to access a **null** field. To avoid this, it is recommended to check if a field is **null** before accessing it, using conditional statements or the null-conditional operator (**?.**).

Handle NullPointerException using Objects.requireNonNull():

In Java, one of the convenience methods provided to handle **NullPointerException** is the **Objects.requireNonNull()** method. This method is used to check if a reference is **null** and throw a **NullPointerException** with a specific error message if it is.

For example, consider the following code that accepts a **String** parameter and assigns it to a variable:

```
public void setName(String name) {

this.name = name;

 }
```

If **name** is **null**, a **NullPointerException** will be thrown when trying to access it later in the code. To avoid this, the **Objects.requireNonNull()** method can be used to check if **name** is **null**:

```
public void setName(String name) {

this.name = Objects.requireNonNull(name, "Name cannot be null");

 }
```

This code will throw a **NullPointerException** with the message "Name cannot be null" if **name** is **null**. This helps to make the code more robust and prevents unexpected errors.

Parameter passing in Method:

In Java, parameters are always passed by value, regardless of whether they are primitive types or reference types. This means that when you pass a variable to a method, a copy of its value is passed, rather than a reference to the variable itself. However, when you pass an object or array to a method, the copy of the value is actually a reference to the object or array. This is sometimes referred to as "call by value" for primitives and "call by reference" for objects and arrays, but it's important to understand that Java always passes by value.

Case 1:

```
public class Person {
    String name;

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

public class Main {
    public static void main(String[] args) {
        Person harry = new Person("Harry");
        Person marry = new Person("Marry");

        System.out.println("Before swapping:");
        System.out.println("harry: " + harry.name); // Output: Harry
        System.out.println("marry: " + marry.name); // Output: Marry

        swapNames(harry, marry);

        System.out.println("After swapping:");
        System.out.println("harry: " + harry.name);
        System.out.println("marry: " + marry.name);

    public static void swapNames(Person p1, Person p2) {
       Person tempName = p1;
        p1 = p2;
        p2 = tempName;
    }
}
```

Output

```
Before swapping:
harry: Harry
marry: Marry
After swapping:
harry: Harry
marry: Marry
```

This is because the swapNames method does not actually swap the names of the Person objects passed to it. This is because Java is pass-by-value, meaning that the method receives a copy of the references to the Person objects, not the actual objects themselves. When the references are swapped inside the method, it does not affect the references outside of the method, so the names of harry and marry remain unchanged.

Case II:

```
public class Person {
    String name;

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

public class Main {
    public static void main(String[] args) {
        Person harry = new Person("Harry");
        Person marry = new Person("Marry");

        System.out.println("Before swapping:");
        System.out.println("harry: " + harry.name); // Output: Harry
        System.out.println("marry: " + marry.name); // Output: Marry

        swapNames(harry, marry);

        System.out.println("After swapping:");
        System.out.println("harry: " + harry.name);
        System.out.println("marry: " + marry.name);
    }

    public static void swapNames(Person p1, Person p2) {
        String tempName = p1.name;
        p1.name = p2.name;
        p2.name = tempName;
    }
}
```

Output:

```
Before swapping:
harry: Harry
marry: Marry
After swapping:
harry: Marry
marry: Harry
```

Explanation:

This is because the `swapNames()` method actually swaps the `name` fields of the `Person` objects `p1` and `p2`. So when `swapNames(harry, marry)` is called, the `name` field of `harry` is swapped with the `name` field of `marry`, resulting in the output above.

Method overloading:

- Method overloading is a feature in Java that allows us to create multiple methods with the same name but different parameters.
- The signature of a method includes the method name and the parameter list. Overloaded methods must have different parameter lists.
- The return type and access modifiers of the methods can be the same or different.
- Method overloading can improve code readability and reduce code duplication.
- Method overloading is resolved at compile-time based on the number, types, and order of the arguments passed to the method.
- If no matching method is found at compile-time, a compile-time error occurs.
- Method overloading is not the same as method overriding, which is a feature of inheritance in Java.

Example of method overloading:

```
public class Calculator {
    public int add(int x, int y) {
        return x + y;
    }

    public int add(int x, int y, int z) {
        return x + y + z;
    }

    public double add(double x, double y) {
        return x + y;
    }
}
```

Note:
Void M1();
void M1(int a,int b)
double M1(double a)

M1(3) //it will promote 3 to double and call double M1(double a)

M1(3,6);

Default constructor in java:

In Java, a default constructor is a no-argument constructor that is automatically generated by the compiler if no constructor is explicitly defined in a class. It has the same name as the class and has no parameters or arguments.

The default constructor initializes the instance variables of a class with default values. For example, integer variables are initialized to 0, boolean variables to false, and reference variables to null.

If a class has any explicit constructor, the default constructor is not automatically generated. In this case, if a no-argument constructor is required, it must be explicitly defined in the class**.**

Constructor Chaining :

In Java, it is possible to call one constructor inside another constructor of the same class. This is known as constructor chaining.

Constructor chaining is useful when you have multiple constructors in a class with different parameters, but they all need to perform some common initialization tasks. Instead of repeating the initialization code in each constructor, you can define a single constructor that performs the common tasks and call it from other constructors.

To call a constructor from another constructor, you use the this() keyword followed by the arguments of the constructor you want to call. The this() keyword must be the first statement in the constructor body.

Here is an example of constructor chaining in Java:

```
public class MyClass {
   private int value;

   public MyClass() {
      this(0); // calling another constructor with default value
   }
```

```
    public MyClass(int value) {
        this.value = value;
    }
}
```

In the above example, the default constructor calls the parameterized constructor using `this(0)`, passing it a default value of 0. The parameterized constructor sets the value of the `value` field based on the argument passed to it.

Note: Java doesn't have destructor it has garbage collector .

Inner classes :

In Java, inner classes are classes defined within another class. There are four types of inner classes in Java:

1. **Member inner class:** It is a non-static inner class that is a member of its enclosing class. It has access to all the members (variables and methods) of its enclosing class, including private members.

2. **Local inner class**: Local inner classes in Java are classes that are defined inside a block of code such as a method, an if statement, or a loop. These classes can only be accessed within the block in which they are defined and are not visible outside the block.

3. **Anonymous inner class**: It is a local inner class without a name. It is usually used to define a class that implements an interface or extends a class in a single expression.

4. **Static nested class:** It is a static class defined within another class. It does not have access to the non-static members of its enclosing class, but it can access them through an object reference.

Member inner class

```
public class OuterClass {
    private int outerVariable;
    public OuterClass(int outerVariable) {
        this.outerVariable = outerVariable;
    }

    public void outerMethod() {
        System.out.println("Outer method is called");
    }

    public class InnerClass {
        private int innerVariable;

        public InnerClass(int innerVariable) {
            this.innerVariable = innerVariable;
        }

        public void innerMethod() {
            System.out.println("Inner method is called");
            System.out.println("Inner variable: " + innerVariable);
            System.out.println("Outer variable: " + outerVariable);
            outerMethod();
        }
    }
}
```

In this example, `InnerClass` is a member inner class of `OuterClass`. `InnerClass` has access to all the members of `OuterClass`, including its private members. `InnerClass` can also have its own methods and fields.

Here is how you can create an instance of `InnerClass` and call its method:

```
OuterClass outerObj = new OuterClass(10);
OuterClass.InnerClass innerObj = outerObj.new InnerClass(20);
innerObj.innerMethod();

Output:
Inner method is called
Inner variable: 20
```

```
Outer variable: 10
Outer method is called
```

Static Inner class:

In Java, a static nested class is a class that is defined within another class but has the `static` modifier. It is also known as a nested top-level class. It is similar to an inner class, but does not have access to the enclosing class's instance variables and methods.

Here's an example of a static inner class:

```
public class OuterClass {
  static int outerNum;
  int num;

  public static class InnerClass {
    public void display() {
      System.out.println("Outer num: " + outerNum);
    }
  }
}
```

In the example above, `InnerClass` is a static inner class of `OuterClass`. It has access to the static variable `outerNum` defined in `OuterClass`, but not to the non-static instance variable `num`.

We can create an instance of `InnerClass` like this:

```
OuterClass.InnerClass inner = new OuterClass.InnerClass();
inner.display(); // Output: Outer num: 0
```

Local Inner class :

```
public class Outer {
    private int num = 10;

    public void printNum() {
        int localVar = 20;

        class LocalInner {
            public void print() {
                System.out.println("num: " + num);
                System.out.println("localVar: " + localVar);
            }
        }

        LocalInner inner = new LocalInner();
        inner.print();
    }
}
```

In the above example, we have a class called `Outer` that has a method called `printNum()`. Inside the `printNum()` method, we define a local inner class called `LocalInner`. This class has a method called `print()` which prints out the value of the `num` variable from the outer class as well as a local variable called `localVar` that is defined within the method.

We then create an instance of the `LocalInner` class and call its `print()` method from within the `printNum()` method.

Note that the `LocalInner` class can only be accessed within the `printNum()` method and is not visible outside of that method.

Anonymous class

An anonymous class in Java is a local class without a name. It is defined and instantiated in a single expression, typically as an argument to a method or as a nested class within another class. Anonymous classes are useful when you need to create a class that implements an interface or extends a class, but you don't want to create a separate named class for it.

Here's an example of using an anonymous class to implement the Runnable interface:

```
public class Main {
    public static void main(String[] args) {
        // Creating an object of an anonymous class
        Runnable myRunnable = new Runnable() {
            @Override
            public void run() {
                System.out.println("Hello from anonymous class!");
            }
        };

        // Running the anonymous class
        myRunnable.run();
    }
}
```

In this example, we create an anonymous class that implements the `Runnable` interface. We override the `run()` method of the `Runnable` interface with our own implementation that simply prints a message to the console. We then create an instance of the anonymous class and assign it to a variable of type `Runnable`. Finally, we call the `run()` method on the variable, which executes the code in our anonymous class and prints the message to the console.

Note : Constructor can't be defined in case of anonymous class.

Variable length arguments :

In Java, variable length arguments, also known as varargs, allow a method to accept a variable number of arguments of the same type. Varargs are represented by three dots (...) after the parameter type.

```
public static void printNumbers(int... numbers) {
    for (int num : numbers) {
        System.out.print(num + " ");
    }
    System.out.println();
}

public static void main(String[] args) {
    printNumbers(1, 2, 3, 4);
    printNumbers(10, 20);
    printNumbers();
}
```

Output:

```
1 2 3 4
10 20
```

Note that a varargs parameter must be the last parameter in a method's parameter list, and there can be at most one varargs parameter in a method.

Method overloading with varargs:

```
public class Example {
    public void printValues(int... values) {
        System.out.println("Printing integer values:");
        for (int value : values) {
            System.out.print(value + " ");
        }
        System.out.println();
    }

    public void printValues(double... values) {
        System.out.println("Printing double values:");
        for (double value : values) {
            System.out.print(value + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        Example example = new Example();

        example.printValues(1, 2, 3);
        example.printValues(1.5, 2.5, 3.5);
    }
}
```

In this example, we have two methods named `printValues`, each with a variable-length argument. One takes an array of integers and the other takes an array of doubles. When we call the `printValues` method with integer arguments, the first overload is invoked, and when we call it with double arguments, the second overload is invoked.

Inheritance

Inheritance is a mechanism in Java that allows a class to be based on another class, inheriting its properties and methods. The class that is being inherited from is called the superclass, and the class that inherits from it is called the subclass.

Inheritance provides the ability to create a hierarchy of classes with shared attributes and behaviors. This can reduce code duplication and make code more modular, as well as make it easier to extend and maintain.

In Java, inheritance is implemented using the extends keyword. When a subclass extends a superclass, it inherits all of the public and protected fields and methods of the superclass. The subclass can also define its own fields and methods, which may override or add to those inherited from the superclass.

For example, consider the following code:

```
public class Animal {
   public void eat() {
      System.out.println("The animal is eating");
   }
}

public class Dog extends Animal {
   public void bark() {
      System.out.println("The dog is barking");
   }
}

public class Main {
   public static void main(String[] args) {
      Dog dog = new Dog();
      dog.eat(); // Output: The animal is eating
      dog.bark(); // Output: The dog is barking
   }
}
```

In this example, the `Dog` class extends the `Animal` class, inheriting its `eat()` method. The `Dog` class also defines its own `bark()` method. When an instance of `Dog` is created and its `eat()` and `bark()`

methods are called, the output shows that the `eat()` method is inherited from the `Animal` class and the `bark()` method is defined in the `Dog` class.

Note: 1. Child class can access all public, protected methods, variables of parent class but not private members , however they can be accessed through getter and setter function.

Super Keyword:

In Java, the `super` keyword is used to refer to the parent class. It is often used to call the constructor of the parent class from the child class.

Here is an example code snippet to demonstrate calling the constructor of the parent class using `super`:

```
class Animal {
    String name;

    public Animal(String name) {
        this.name = name;
        System.out.println("Animal constructor called");
    }
}

class Dog extends Animal {
    int age;

    public Dog(String name, int age) {
        super(name);
        this.age = age;
        System.out.println("Dog constructor called");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog("Max", 3);
        System.out.println("Name: " + myDog.name);
        System.out.println("Age: " + myDog.age);
```

```
  }
}

Output:
Animal constructor called
Dog constructor called
Name: Max
Age: 3
```

Method Overriding:

If a parent and child class have a method with the same name, it is called method overriding. In this case, the method in the child class overrides the method in the parent class.

When the method is called on the child class object, the method in the child class is executed instead of the method in the parent class. However, if we want to call the parent class method from the child class method, we can use the "super" keyword.

```
class Parent {
  public void myMethod() {
    System.out.println("Parent class method");
  }
}

class Child extends Parent {
  public void myMethod() {
    System.out.println("Child class method");
    super.myMethod(); // calling parent class method
  }
}

public class Main {
  public static void main(String args[]) {
    Child obj = new Child();
    obj.myMethod();
  }
}
Output:

Child class method
Parent class method

Example :
class Animal {
    public void makeSound() {
        System.out.println("Animal is making a sound");
    }
}

class Dog extends Animal {
    public void makeSound() {
        System.out.println("Dog is barking");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog();
        animal.makeSound();
    }
}
Output :

Dog is barking
```

can we assign object of subclass to super class in java

Yes, in Java, you can assign an object of a subclass to a variable of its superclass type using inheritance. This is called upcasting.

For example, suppose you have a superclass `Animal` and a subclass `Dog`:

1. **Multilevel inheritance:** When a class extends a superclass, and that superclass also extends another superclass.

```
• class Animal {
      public void move() {
          System.out.println("Animals can move");
      }
  }
  class Mammal extends Animal {
      public void eat() {
          System.out.println("Mammals can eat");
      }
  }
  class Dog extends Mammal {
      public void bark() {
          System.out.println("Dogs can bark");
      }
  }
```

1. **Hierarchical inheritance:** When multiple subclasses inherit from a single superclass.

```
class Animal {
    public void move() {
        System.out.println("Animals can move");
    }
}

class Dog extends Animal {
    public void bark() {
        System.out.println("Dogs can bark");
    }
}
class Cat extends Animal {
    public void meow() {
        System.out.println("Cats can meow");
    }
}
```

. Multiple inheritance (using interfaces): When a class implements multiple interfaces, which can be thought of as contracts specifying a set of methods that the implementing class must provide.

```
interface Animal {
    public void move();
}
interface Bird {
    public void fly();
}
class Eagle implements Animal, Bird {
    public void move() {
        System.out.println("Eagles can walk and run");
    }
    public void fly() {
        System.out.println("Eagles can fly");
    }
}
```

In this example, the Eagle class implements both the Animal and Bird interfaces, so it must provide implementations for both the move() and fly() methods. It's important to note that Java does not support multiple inheritance of classes (i.e., a class can only extend one other class), but it does support multiple inheritance of interfaces.

To create an object of an inherited class in Java, you can follow these steps: 1. Create a superclass with some properties and methods

```
class Animal {
    private String name;
    public Animal(String name) {
        this.name = name;
    }
    public void eat() {
        System.out.println(name + " is eating");
    }
}
```

Create a subclass that extends the superclass and adds its own properties and methods:

```
class Dog extends Animal {
    private String breed;
    public Dog(String name, String breed) {
        super(name);
        this.breed = breed;
    }
    public void bark() {
        System.out.println(breed + " is barking");
    }
}
```

1. **Create an object of the subclass using its constructor:**

Dog myDog = new Dog("Max", "Labrador")

In this example, we create an object of the Dog class named myDog with a name of "Max" and a breed of "Labrador". The Dog constructor calls the constructor of the Animal superclass using the super() method to set the name property. Now we can call methods on our myDog object

```
        myDog.eat(); // Output: Max is eating
        myDog.bark(); // Output: Labrador is barking
```

Note that we can call the eat() method from the Animal superclass because the Dog class inherits it through the extends keyword. We can also call the bark() method that is specific to the Dog class.

Type Casting in Inheritance

First, let's review the basics of inheritance and typecasting. Inheritance allows a subclass to inherit properties (fields and methods) from its superclass. Typecasting is the process of converting an object from one data type to another. In Java, there are two types of typecasting: upcasting and downcasting. Upcasting is when you convert a subclass object to a superclass reference type. Downcasting is when you convert a superclass reference type to a subclass object. Here are some examples that demonstrate the different types of typecasting in inheritance

```java
class Animal {
    public void makeSound() {
        System.out.println("Animal makes a sound");
    }
}
class Dog extends Animal {
    public void makeSound() {
        System.out.println("Dog barks");
    }
}
public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog(); // upcasting
        animal.makeSound(); // outputs "Dog barks"
    }
}
```

In this example, we have an Animal class and a Dog class that extends the Animal class. The Dog class overrides the makeSound() method of the Animal class. In the main method, we create an object of the Dog class and upcast it to an Animal object. We then call the makeSound() method on the animal object, which outputs "Dog barks". This is because the Dog class overrides the makeSound() method of the Animal class.

Example 2: Downcasting

```java
class Animal {
    public void makeSound() {
        System.out.println("Animal makes a sound");
    }
}
class Dog extends Animal {
    public void makeSound() {
        System.out.println("Dog barks");
    }
    public void playFetch() {
        System.out.println("Dog plays fetch");
    }
}
public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog(); // upcasting
        animal.makeSound(); // outputs "Dog barks"
        Dog dog = (Dog) animal; // downcasting
        dog.makeSound(); // outputs "Dog barks"
        dog.playFetch(); // outputs "Dog plays fetch"
    }
}
```

In this example, we have an Animal class and a Dog class that extends the Animal class. The Dog class overrides the makeSound() method of the Animal class and adds a new playFetch() method. In the main method, we create an object of the Dog class and upcast it to an Animal object. We then call the makeSound() method on the animal object, which outputs "Dog barks"

We then downcast the animal object to a Dog object using an explicit cast. We can then call the makeSound() and playFetch() methods on the dog object, which output "Dog barks" and "Dog plays fetch", respectively. Note that downcasting can cause a ClassCastException if the object being casted is not an instance of the subclass.

Example 3: Using instanceof

```java
class Animal {
    public void makeSound() {
        System.out.println("Animal makes a sound");
    }
}
class Dog extends Animal {
    public void makeSound() {
        System.out.println("Dog barks");
    }
    public void playFetch() {
        System.out.println("Dog plays fetch");
    }
}
public class Main {

        public static void main(String[] args) {
            Animal animal = new Dog();
            if (animal instanceof Dog) {
                Dog dog = (Dog) animal;
                dog.playFetch();
            } else {
                System.out.println("Animal is not a dog");
            }
        }
    }
```

In this example, we have an Animal class and a Dog class that extends the Animal class. The Dog class overrides the makeSound() method of the Animal class and adds a new playFetch() method

In the main method, we create an object of the Dog class and assign it to an Animal variable. We then use the instanceof operator to check if the animal object is an instance of the Dog class. If the animal object is a Dog , we downcast it to a Dog object and call the playFetch() method. If the animal object is not a Dog , we output a message saying "Animal is not a dog". Using instanceof is a good way to prevent a ClassCastException when downcasting an object. It allows you to check if an object is an instance of a particular class before attempting to downcast it.

Object Class in Java

Object Class: The Object class is the root class in Java, from which all other classes inherit. It provides several methods that are available to all objects in Java.

1. equals(Object obj):
The equals method is used to compare the equality of two objects. By default, it checks if two object references point to the same memory location. However, you can override this method in your own classes to define your own custom equality check.

2.final: The final keyword can be used in various contexts. When applied to a class, it prevents the class from being subclassed. When applied to a method, it prevents the method from being overridden. When applied to a variable, it makes the variable a constant (i.e., its value cannot be changed).

3.Clone: The clone method creates and returns a copy of an object. To make an object cloneable, the class must implement the Cloneable interface and override the clone method

4.getClass: The getClass method returns the runtime class of an object. It is useful when you need to determine the actual class of an object at runtime.

5.wait, notify, notifyAll: These methods are used for inter-thread communication in Java. wait : Causes the current thread to wait until another thread invokes the notify() or notifyAll() method. notify : Wakes up a single thread that is waiting on the object's monitor. notifyAll : Wakes up all the threads that are waiting on the object's monitor.

6.toString: The toString method returns a string representation of an object. By default, it returns the class name along with the object's hash code.

7.HashCode : In Java, the hashCode() method is used to generate a unique integer value (hash code) for an object. The hash code is used by data structures like hash tables to optimize searching and retrieval operations. The default implementation of hashCode() in the Object class returns the memory address of the object in hexadecimal format. However, it is common to override this method in custom classes to provide a meaningful and consistent hash code based on the object's state.

Example :

```java
import demo.package1.myclass;

public class object{
    private String s;
    private int a=10;

    object(String str,int a){
        this.s=str;
        this.a=a;

    }

    void setA(String s){
        this.s=s;
    }
    String getA(){
        return this.s;
    }
    @Override public int hashCode(){
        return this.a;
    }
    @Override public String toString(){
        return "My name is "+this.s;
    }

    public boolean equals(Object t)
    {
        if(!(t instanceof object) || t==null)
        {
            return false;
        }

        object t1=(object) t;

        if(t1.getA()==this.s && t1.hashCode()==this.hashCode())
        {
            return true;
        }

        return false;
    }



    public static void main(String args[]){
        Object s=new String("java lab");
        Class get_class=s.getClass();
        System.out.println("Class is "+get_class);
        Object a=new object("yasir",9);
        object b=new object("yasir",9);
        System.out.println(((object)b).equals(a));
        object c=new object("yasir",70);
        System.out.println(c.hashCode());
        object d=new object("moin",85);
        System.out.println(((object)b).equals(d));
        System.out.println(d.toString());
        myclass myObject = new myclass();
        myObject.ImmutableMethod("moin");

    }
}
```

- **Packages in java:**

In Java, a package is a way of organizing related classes, interfaces, and other resources. It provides a mechanism for grouping related code together, making it easier to manage and maintain larger projects. A package can contain multiple classes and other packages.

Here are some key points to understand about packages in Java:

1. **Package Structure**: Packages are organized hierarchically using a dot (.) notation. For example, the package name **com.example.myapp** indicates that the package **myapp** is inside the package **example**, which is inside the package **com**.

2. **Package Declaration**: At the beginning of a Java source file, you can declare the package to which the file belongs using the **package** keyword. For example:

```
package com.example.myapp;
```

This statement declares that the current file belongs to the **com.example.myapp** package.

3. **Package Naming Conventions**: By convention, package names are written in lowercase letters. Also, it's common to use a reverse domain name as the package name to avoid naming conflicts.

4. **Import Statements**: To use classes from other packages within your code, you need to import them. Import statements allow you to specify which classes you want to use from external packages. For example:

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

This statement imports the **Scanner** class from the **java.util** package, allowing you to use it in your code.

5. **Default Package**: If you don't specify a package at the beginning of your source file, it belongs to the default package. However, it's considered good practice to always use explicit package declarations.

6. **Access Modifiers**: Packages provide a level of encapsulation and access control. Classes and members within a package can be accessed by other classes within the same package without using explicit access modifiers. This is known as package-private or default access. If you want to make a class or member accessible to classes outside the package, you need to use appropriate access modifiers (e.g., **public**, **protected**, **private**).

Packages are used extensively in Java libraries and frameworks to organize and manage large codebases. They help in avoiding naming conflicts, promoting code reusability, and enhancing project organization and maintenance.

Note: 1. **In Java, it is not possible to import all packages in a single import statement. Each package or class needs to be imported individually or using wildcard imports for specific packages.**

Importing all packages in a single import statement is not supported by the Java language. This is because importing all packages can result in a significant amount of overhead and potential naming conflicts. It is generally considered good practice to import only the specific packages and classes that you need in your code, rather than importing everything.

```
E.g import java.util.*;
 import java.io.*;
```

In this example, we are importing all classes from the java.util package and all classes from the java.io package.

Name Conflict in classes

When importing classes from different packages, there is a possibility of name conflicts if two or more imported classes have the same name. This situation can arise if different packages define classes with identical names.

To handle name conflicts, you can use fully qualified class names or import specific classes with their package names to disambiguate them. Here's an example to illustrate this:

```
import com.example.package1.MyClass;

import com.example.package2.MyClass;
```

In this case, we have two classes named MyClass from different packages, package1 and package2. By specifying the package name along with the class name in the import statements, we avoid any naming conflicts. When using the classes in the code, we can refer to them using their fully qualified names:

```
com.example.package1.MyClass obj1 = new com.example.package1.MyClass();
com.example.package2.MyClass obj2 = new com.example.package2.MyClass();
```

Alternatively, you can import one of the classes using its fully qualified name while importing the other class directly:

```
import com.example.package1.MyClass;

import com.example.package2.*;
```

Now, you can refer to MyClass from package1 directly, but you need to use its fully qualified name when referring to MyClass from package2:

```
MyClass obj1 = new MyClass(); // Referring to MyClass from package1
com.example.package2.MyClass obj2 = new com.example.package2.MyClass();
```

By using the fully qualified names or selectively importing classes, you can resolve name conflicts and differentiate between classes with the same name from different packages.

It's important to be mindful of potential name conflicts when working with multiple packages and to choose appropriate import strategies to avoid ambiguity in your code.

Import Static Methods , variables of class using package

The import static statement in Java allows you to import static members (fields and methods) from a class or an interface, making them accessible in your code without having to use the fully qualified class name every time.

Here are some key points to understand about the import static statement:

Static Members: In Java, static members are associated with the class itself rather than with specific instances of the class. They can be accessed directly using the class name, followed by the member name. For example, if you have a static method printMessage() in a class Utility, you can invoke it using Utility.printMessage().

Importing Static Members: The import static statement allows you to import specific static members from a class or an interface, so you can use them directly without the class name. This simplifies your code and improves readability.

Syntax: The syntax for importing static members is as follows:

```
import static <class/interface name>.<member name>;
```

For example, to import the static method printMessage() from the class Utility, you would use:

import static com.example.Utility.printMessage;

Wildcard Import: Similar to regular imports, you can also use the wildcard (*) character to import all static members from a class or an interface. For example:

```
import static com.example.Utility.*;
```

This imports all static members (fields and methods) from the class Utility, and you can directly access them without specifying the class name.

Usage: Once you have imported the static members using the import static statement, you can use them directly in your code without using the class name. For example:

```
public class MyClass {
    public static void main(String[] args) {
        printMessage(); // Using the imported static method directly
        int value = DEFAULT_VALUE; // Using the imported static field directly
    }
}
```

In the above example, printMessage() and DEFAULT_VALUE are static members imported using import static.

The import static statement can help simplify your code by allowing direct access to static members. However, it's important to use it judiciously and avoid excessive static imports, as it may lead to code confusion or naming conflicts.

Import vs Include :

Import:

Import is a directive used in languages like Java, Python, and C#.

It is used to bring classes, packages, or modules from external files or libraries into your current file so that you can use them.

The import statement allows you to access the imported code using its name without fully qualifying it with the file or library it came from.

It is primarily used for organizing and reusing code, as well as avoiding naming conflicts between different modules or libraries.

In languages like Java, you can import classes, interfaces, and static members using the import statement.

Include:

Include is a directive used in languages like C, C++, and PHP.

It is used to include the content of another file into the current file at the point where the include directive is placed.

The included file is processed by the preprocessor and its content is inserted into the source code before compilation.

It is commonly used for code reuse, separating code into different files for better organization, and sharing common definitions.

The included file is treated as if its content was physically present in the file, and any code or declarations in the included file become part of the current file.

In summary, import is used to bring external code into your file for usage, while include is used to physically insert the content of another file into the current file during preprocessing.

- **JAR Files in Java**

A JAR (Java Archive) file is a compressed file format used to package multiple Java class files, resources, and metadata into a single file. It simplifies the distribution and deployment of Java applications or libraries by bundling all necessary files into one portable archive. JAR files can be made executable, include dependencies, and are commonly used for distributing Java applications. They are created and manipulated using the **jar** command-line tool provided with the JDK. JAR files are an efficient and standardized way to package and distribute Java code.

- **Interface in Java :**

In Java, an interface is a reference type that defines a contract for classes to follow. It defines a set of methods that a class must implement. It provides a way to achieve abstraction and support multiple inheritance-like behavior.

Here's an example of an interface declaration in Java:

```
public interface Printable {
   void print();
}
```

In this example, the Printable interface declares a single method print(). Any class that implements this interface must provide an implementation for the print() method.

To implement an interface, a class uses the implements keyword and provides implementations for all the methods defined in the interface. Here's an example:

```
public class Printer implements Printable {
    public void print() {
        System.out.println("Printing...");
    }
}
```

The Printer class implements the Printable interface and provides an implementation for the print() method.

Interfaces can also inherit from other interfaces using the extends keyword. This allows for the creation of hierarchical interfaces. Here's an example:

```
public interface Showable extends Printable {
    void show();
}
```

The Showable interface extends the Printable interface and adds an additional method show (). Any class that implements Showable must provide implementations for both print() and show() methods.

1. **Partial Interface :**

In Java, you cannot directly implement a partial interface, meaning you cannot implement only a subset of the methods defined in an interface. When you implement an interface, you are required to provide an implementation for all the methods declared in that interface.

However, you can achieve a similar effect by using abstract classes. In an abstract class, you can provide default implementations for some methods while leaving others as abstract, which must be implemented by the concrete classes that extend the abstract class. Here's an example:

```java
 //partial interface

public class lab6_1 {
    interface MyInterface {
        void method1();
        void method2();
        void method3();
    }

    abstract static class PartialImplementation implements MyInterface {
        public void method1() {
            System.out.println("Default implementation for method1");
        }


        public abstract void method2();
        public abstract void method3();
    }

    static class ConcreteClass extends PartialImplementation {
        public void method2() {
            System.out.println("Implementation for method2");
        }

        public void method3() {
            System.out.println("Implementation for method3");
        }
    }

    public static void main(String[] args) {
        ConcreteClass concreteObj = new ConcreteClass();
        concreteObj.method1();
        concreteObj.method2();
        concreteObj.method3();
    }
}
```

Output

Default implementation for method1

Implementation for method2

Implementation for method3

2. Dynamic dispatch using interface

Dynamic dispatch, also known as runtime polymorphism or late binding, is a mechanism in object-oriented programming that allows different methods to be executed based on the actual type of the object at runtime. In Java, dynamic dispatch is commonly achieved through interfaces.

```
interface Shape {
    void draw();
}

class Circle implements Shape {
    public void draw() {
        System.out.println("Drawing a circle");
    }
}

class Rectangle implements Shape {
    public void draw() {
        System.out.println("Drawing a rectangle");
    }
}
```

```
public class lab6_2 {
    public static void main(String[] args) {
        Shape shape1 = new Circle();
        Shape shape2 = new Rectangle();

        shape1.draw();
        shape2.draw();
    }

};
```

Output:

Drawing a circle

Drawing a rectangle

When the draw() method is called on shape1 and shape2, dynamic dispatch occurs. The JVM determines the actual types of the objects (Circle and Rectangle respectively) and dispatches the method calls to the appropriate implementations.

3. Class implementing multiple interfaces with same default method:

In Java, if a class implements multiple interfaces that have the same default method, it can cause a compilation error due to a conflict known as "diamond problem" or "multiple inheritance of default methods".

To resolve this conflict, you have a few options:

1. Override the default method in the implementing class: You can provide your own implementation of the default method in the implementing class, resolving the conflict. Here's an example:

```java
interface MyInterface {
    default void myMethod() {
        System.out.println("Default method in MyInterface");
    }
}

interface MyAnotherInterface {
    default void myMethod() {
        System.out.println("Default method in MyAnotherInterface");
    }
}

class MyClass implements MyInterface, MyAnotherInterface {
    public void myMethod() {
        // Provide your own implementation
        System.out.println("Custom implementation in MyClass");
    }
}

public class Main {
    public static void main(String[] args) {
        MyClass myObj = new MyClass();
        myObj.myMethod();  // Output: Custom implementation in MyClass
    }
}
```

2. Explicitly call the desired default method: If you want to use the default method from a specific interface, you can call it explicitly using the interface name. Here's an example:

```java
interface MyInterface {
    default void myMethod() {
        System.out.println("Default method in MyInterface");
    }
}

interface MyAnotherInterface {
    default void myMethod() {
        System.out.println("Default method in MyAnotherInterface");
    }
}

class MyClass implements MyInterface, MyAnotherInterface {
    public void myMethod() {
        // Call the desired default method explicitly
        MyInterface.super.myMethod();
    }
}

public class lab6_3 {
    public static void main(String[] args) {
        MyClass myObj = new MyClass();
        myObj.myMethod();
    }
}
```

Output:

```
Default method in MyInterface
```

1. Constants in interface

In Java, you can define constants in an interface by declaring them as **`public static final`** variables. These variables are implicitly considered as constants, and they can be accessed directly through the interface name. Here's an example:

```
interface Constants {
    int MAX_SIZE = 100;
    String DEFAULT_NAME = "John";
}

public class lab6_4 {
    public static void main(String[] args) {
        System.out.println("Max size: " + Constants.MAX_SIZE);
        System.out.println("Default name: " + Constants.DEFAULT_NAME);
    }
}
```

Output:

Max size: 100

Default name: John

By convention, constant names are usually written in uppercase letters, and using the interface name as the qualifier when accessing the constants helps to clearly indicate their origin.

2. Use of static , private, default methods in interface

1. `static` methods: In Java 8 and later versions, interfaces can contain `static` methods. These methods are associated with the interface itself and can be invoked using the interface name, without requiring an instance of the implementing class. Here's an example:

```
interface MyInterface {
    static void staticMethod() {
        System.out.println("Static method in MyInterface");
    }
}

public class Main {
    public static void main(String[] args) {
        MyInterface.staticMethod();  // Output: Static method in
MyInterface
    }
}
```

2. **`default`** methods: In Java 8 and later versions, interfaces can have **`default`** methods, which provide a default implementation for a method. These methods can be overridden by the implementing classes, but they are not required to do so. Here's an example:

```
interface MyInterface {
    default void defaultMethod() {
        System.out.println("Default method in MyInterface");
    }
}

class MyClass implements MyInterface {
    // The defaultMethod() is inherited and can be used as is
}

public class Main {
    public static void main(String[] args) {
        MyClass myObj = new MyClass();
        myObj.defaultMethod();  // Output: Default method in MyInterface
    }
}
```

3. **`private`** methods: Starting from Java 9, interfaces can have **`private`** methods. These methods are only accessible within the interface and cannot be inherited or overridden by implementing classes or other interfaces. They are typically used to encapsulate common code or to assist in implementing default methods. Here's an example:

```
interface MyInterface {
    default void defaultMethod() {
        commonMethod();
        System.out.println("Default method in MyInterface");
    }

    private void commonMethod() {
        System.out.println("Private method in MyInterface");
    }
}

class MyClass implements MyInterface {
    // The defaultMethod() is inherited and can be used as is
}

public class Main {
    public static void main(String[] args) {
        MyClass myObj = new MyClass();
        myObj.defaultMethod();
        // Output:
        // Private method in MyInterface
        // Default method in MyInterface
    }
}
```

3. Interface Extending Another Interface

In Java, an interface can extend another interface to create a hierarchy of interfaces. This allows for inheritance and dynamic method dispatch when implementing classes or interfaces.

Here's an example that demonstrates interface extension and dynamic method dispatch:

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

interface Mammal extends Animal {
    void eat();
}

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

    public void eat() {
        System.out.println("Eating like a dog");
    }
}

class Cat implements Mammal {
    public void makeSound() {
        System.out.println("Meow!");
    }

    public void eat() {
        System.out.println("Eating like a cat");
    }
}

public class Main {
    public static void main(String[] args) {
        Mammal mammal1 = new Dog();
        Mammal mammal2 = new Cat();

        mammal1.makeSound();  // Output: Bark!
        mammal1.eat();  // Output: Eating like a dog

        mammal2.makeSound();  // Output: Meow!
        mammal2.eat();  // Output: Eating like a cat
    }
}
```

Note: Interface can't extend a class

4. Nested Interface

In Java, it is possible to define an interface within another interface, which is referred to as a nested interface. Here's an example demonstrating the concept of a nested interface:

```java
interface OuterInterface {
    void outerMethod();

    interface NestedInterface {
        void nestedMethod();
    }
}

class MyClass implements OuterInterface, OuterInterface.NestedInterface {
    public void outerMethod() {
        System.out.println("Implementation of outerMethod");
    }

    public void nestedMethod() {
        System.out.println("Implementation of nestedMethod");
    }
}

public class Main {
    public static void main(String[] args) {
        MyClass myObj = new MyClass();
        myObj.outerMethod();   // Output: Implementation of outerMethod
        myObj.nestedMethod();  // Output: Implementation of nestedMethod
    }
}
```

OUTPUT:

```
Implementation of outerMethod

Implementation of nestedMethod
```

Exception Handling

1. **Compile-time errors**: Compile-time errors, also known as compilation errors or syntax errors, occur during the compilation phase of the program. These errors indicate that the code violates the rules of the programming language and cannot be compiled into executable code. They are detected by the compiler and must be fixed before the program can be executed.

E.g : Syntax error, Type error.

```
public class CompileTimeErrorExample {

    public static void main(String[] args) {

        int x = "Hello"; // Type error: incompatible types

        System.out.println("Value of x: " + x);

    }

}
```

Runtime errors:

Runtime errors, also known as exceptions or runtime exceptions, occur during the execution of a program. These errors are not detected by the compiler during compilation and only become apparent when the program is running. Runtime errors indicate exceptional situations or errors that occur at runtime and can cause the program to terminate abnormally if not handled properly.

Common examples of runtime errors include:

Arithmetic exceptions: Division by zero or taking the square root of a negative number.

Null pointer exceptions: Accessing or manipulating an object that is null.

Array index out of bounds: Accessing an array element with an invalid index.

Class cast exceptions: Incorrectly casting an object to an incompatible type.

Here's an example of a runtime error:

```java
public class RuntimeErrorExample {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3};
        System.out.println(numbers[3]); // ArrayIndexOutOfBoundsException
    }
}
```

Error vs Exception

1. Errors: Errors are exceptional conditions that typically arise due to external factors or situations that are beyond the control of the programmer. They indicate serious problems that may prevent the program from functioning properly and are generally not recoverable. Errors are typically not caught or handled by the program.

Examples of errors include:

- OutOfMemoryError: Occurs when the JVM cannot allocate enough memory to perform an operation.
- StackOverflowError: Occurs when the call stack exceeds its maximum limit, usually due to infinite recursion.
- LinkageError: Occurs when there are problems with linking classes or resources at runtime.

Errors are usually caused by critical system failures, hardware malfunctions, or severe programming mistakes. Since they are generally irrecoverable, the typical response to an error is to terminate the program and possibly log the error for debugging purposes.

2. Exceptions: Exceptions, on the other hand, are abnormal conditions that occur within the program's normal flow of execution. Exceptions represent exceptional circumstances or errors that can be handled and recovered from within the program. They are objects that are thrown and caught using exception handling mechanisms.

Try Catch

1. Handling Exceptions with `try-catch`: The `try-catch` block is used to catch and handle exceptions gracefully. Here's the syntax:

```
try {
    // Code that may throw an exception
} catch (ExceptionType1 e1) {
    // Handle exception of type ExceptionType1
} catch (ExceptionType2 e2) {
    // Handle exception of type ExceptionType2
} finally {
    // Optional block for cleanup or final statements (executed regardless of whether an exception occurs or not)
}
```

Throwing Exceptions with throw:

You can manually throw an exception using the throw keyword. This is useful when you want to raise an exception based on certain conditions or custom error handling logic. Here's an example:

java

Copy code

```
if (condition) {
    throw new ExceptionType("Error message"); // Throwing an exception with a custom error message
}
```

By throwing exceptions, you can communicate and handle exceptional scenarios in your program.

3. Checked Exceptions: Checked exceptions are exceptions that are checked by the compiler at compile-time. If a method throws a checked exception, the caller must either catch and handle the exception or declare that it throws the exception. This ensures that potential exceptions are properly handled by the calling code.

Examples of checked exceptions include:

- IOException: Occurs when there is an error in input/output operations.
- SQLException: Occurs when there are problems with database access.

4. Unchecked Exceptions (Runtime Exceptions): Unchecked exceptions, also known as runtime exceptions, do not require explicit handling or declaration. They are usually caused by programming errors, invalid input, or unexpected conditions. Unchecked exceptions extend the `RuntimeException` class or its subclasses.

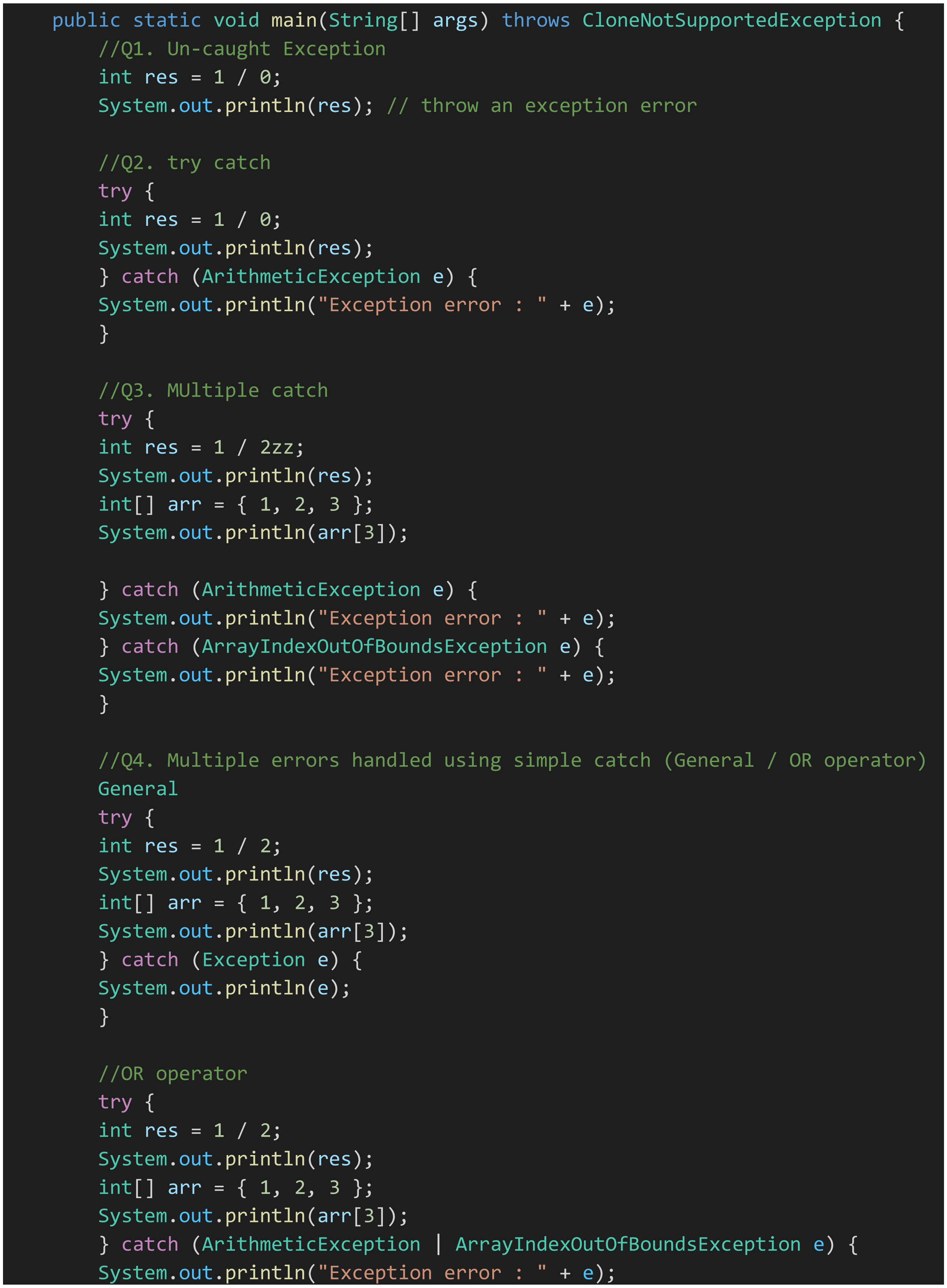
Examples of unchecked exceptions include:

- NullPointerException: Occurs when a null reference is accessed.
- ArrayIndexOutOfBoundsException: Occurs when accessing an array with an invalid index.
- ArithmeticException: Occurs when performing illegal arithmetic operations like division by zero.

6. The `finally` Block: The `finally` block is used for cleanup operations or final statements that need to be executed regardless of whether an exception occurred or not. The `finally` block is placed after the `try-catch` block and is optional. It ensures that critical cleanup tasks are performed, such as closing resources (e.g., file handles, database connections).

Note: If two or more exceptions exists inside try block , then only first exception will be handled by catch block and control will come out of try block

```java
import java.lang.reflect.Array;

public class Q {
```



```
    public static void main(String[] args) throws CloneNotSupportedException {
        //Q1. Un-caught Exception
        int res = 1 / 0;
        System.out.println(res); // throw an exception error

        //Q2. try catch
        try {
        int res = 1 / 0;
        System.out.println(res);
        } catch (ArithmeticException e) {
        System.out.println("Exception error : " + e);
        }

        //Q3. MUltiple catch
        try {
        int res = 1 / 2zz;
        System.out.println(res);
        int[] arr = { 1, 2, 3 };
        System.out.println(arr[3]);

        } catch (ArithmeticException e) {
        System.out.println("Exception error : " + e);
        } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Exception error : " + e);
        }

        //Q4. Multiple errors handled using simple catch (General / OR operator)
        General
        try {
        int res = 1 / 2;
        System.out.println(res);
        int[] arr = { 1, 2, 3 };
        System.out.println(arr[3]);
        } catch (Exception e) {
        System.out.println(e);
        }

        //OR operator
        try {
        int res = 1 / 2;
        System.out.println(res);
        int[] arr = { 1, 2, 3 };
        System.out.println(arr[3]);
        } catch (ArithmeticException | ArrayIndexOutOfBoundsException e) {
        System.out.println("Exception error : " + e);
```

```java
}

//Q5. Try catch with finally block (use return statement)
int num = finallyBlock();
System.out.println(num);

//Q6.
try {
int res = 1 / 0;
System.out.println(res);
} finally {
System.out.println("This is finally");
}

//Q7. Checked Exception
//Throws Clause
try {
divide(0);
} catch (ArithmeticException e) {
System.out.println("Exception error : " + e);
}

//throws clause in clone object
Test t1 = new Test();
t1.a = 10;
t1.b = 20;
Test t2 = (Test) t1.clone();

t2.a = 100;
System.out.println(t1.a + " " + t1.b + " ");
System.out.println(t2.a + " " + t2.b + " ");

//ii. Nested try catch & throws clause
try {
nestedTryCatch();
} catch (ArithmeticException e) {
System.out.println("Exception error : " + e);
}

//Q8. Use defined exceptions
MoinNameChecker obj = new MoinNameChecker();
try {
String name = "John";
obj.checkName(name);
} catch (MoinException e) {
```

```
            System.out.println("Error: " + e);
        }
    }

    // Nested try catch
    public static void nestedTryCatch() throws ArithmeticException {
        try {
            int result = 1 / 2;
            System.out.println("Result: " + result);
            // int[] arr = { 1, 3, 4 };
            // System.out.println(arr[5]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Exception error : " + e);
        }
    }

    // finally block
    public static int finallyBlock() {
        try {
            int res = 1 / 0;
            return res;
        } catch (ArithmeticException e) {
            System.out.println("Exception error : " + e);
        } finally {
            System.out.println("Finally block executed");
        }
        return -1;
    }

    public static int divide(int a) throws ArithmeticException {
        return 1 / a;
    }

}

// Q8. Custom error
class  MoinException extends Exception {
    public MoinException(String message) {
        super(message);
    }

}

class MoinNameChecker {
    public void checkName(String name) throws MoinException {
        if (!name.equals("Moin")) {
            throw new MoinException("Invalid name. Name should be Moin. Given
name: " + name);
        }
    }
}

class Test implements Cloneable {
    int a;
    int b;

    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

Nested Try Catch

```
try {
    // Outer try block
    // Code that may throw exceptions

    try {
        // Inner try block
        // Code that may throw exceptions
    } catch (ExceptionType2 e2) {
        // Handle exception of type ExceptionType2 (inner catch block)
    }

} catch (ExceptionType1 e1) {
    // Handle exception of type ExceptionType1 (outer catch block)

}
```

Here's the flow of execution:

1. The program execution enters the outer try block.
2. If an exception occurs within the inner try block, the control transfers to the inner catch block. The inner catch block handles the exception of type `ExceptionType2`.
3. If an exception occurs within the outer try block, but outside the inner try block, the control transfers to the outer catch block. The outer catch block handles the exception of type `ExceptionType1`.

Custom Exception :

In Java, you can create your own custom exceptions by defining a class that extends either `Exception` or `RuntimeException` (or one of their subclasses). Creating custom exceptions allows you to define and handle exceptional conditions specific to your application or domain.

Here's an example of creating a custom exception that extends the `Exception` class:

```
public class CustomException extends Exception {

    public CustomException() {

        super();

    }



    public CustomException(String message) {

        super(message);

    }

}
```

To use this custom exception in your code, you can throw it using the `throw` keyword:

```
public void someMethod() throws CustomException {

    // Code that might throw a CustomException

    if (/* Some condition */) {

        throw new CustomException("This is a custom exception.");

    }

}
```

To handle the custom exception, you can use a try-catch block:

```
try {

    someMethod();

} catch (CustomException e) {

    // Handle the CustomException

    System.out.println("CustomException occurred: " + e.getMessage());

}
```

Data Type

1. Primitive Types: Primitive types in Java are the most basic data types. They represent simple values and are not objects. Java provides eight primitive types:

- `boolean`: Represents the truth values `true` or `false`.
- `byte`: Represents 8-bit signed integers with a range of -128 to 127.
- `short`: Represents 16-bit signed integers with a range of -32,768 to 32,767.
- `int`: Represents 32-bit signed integers with a range of -2,147,483,648 to 2,147,483,647.

- `long`: Represents 64-bit signed integers with a range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
- `float`: Represents single-precision 32-bit floating-point numbers.
- `double`: Represents double-precision 64-bit floating-point numbers.
- `char`: Represents a single Unicode character.

Primitive types have some important characteristics:

- They are stored by value, meaning that when you assign a primitive variable to another variable or pass it as a method argument, the value itself is copied.
- They have a default value (e.g., 0 for numeric types, false for `boolean`, '\u0000' for `char`).
- They are not nullable; they cannot hold a `null` value.

2. Object Types: Object types, also known as reference types, represent complex data structures and are based on classes or interfaces. They are instances of classes or subclasses created using the `new` keyword. Examples of object types include:

- Classes: Objects created from classes defined in Java or custom classes you create.
- Arrays: Objects that store multiple elements of the same type.

Object types have the following characteristics:

- They are stored by reference, meaning that variables of object types store a reference (memory address) to the actual object in memory.
- They can be `null`, indicating the absence of an object.
- They can have methods and fields defined by their class, allowing for complex behavior and data storage.

Wrapper classes:

Wrapper classes in Java are a set of classes that wrap primitive types, allowing them to be treated as objects. They provide useful methods and additional functionality when working with primitive types in an object-oriented context. Each primitive type in Java has a corresponding wrapper class.

Here are the wrapper classes for primitive types in Java:

Boolean: Represents the boolean primitive type.

Example usage:

java

Copy code

Boolean boolObj = Boolean.valueOf(true);

boolean boolValue = boolObj.booleanValue();

Byte: Represents the byte primitive type.

Example usage:

java

Copy code

Byte byteObj = Byte.valueOf((byte) 10);

byte byteValue = byteObj.byteValue();

Short: Represents the short primitive type.

Example usage:

java

Copy code

Short shortObj = Short.valueOf((short) 100);

short shortValue = shortObj.shortValue();

Integer: Represents the int primitive type.

Example usage:

java

Copy code

```
Integer intObj = Integer.valueOf(1000);
int intValue = intObj.intValue();
```

Long: Represents the long primitive type.

Example usage:

java

Copy code

```
Long longObj = Long.valueOf(100000L);
long longValue = longObj.longValue();
```

Float: Represents the float primitive type.

Example usage:

java

Copy code

```
Float floatObj = Float.valueOf(3.14f);
float floatValue = floatObj.floatValue();
```

Double: Represents the double primitive type.

Example usage:

java

Copy code

Double doubleObj = Double.valueOf(3.14159);

double doubleValue = doubleObj.doubleValue();

Character: Represents the char primitive type.

Example usage:

java

Copy code

Character charObj = Character.valueOf('A');

char charValue = charObj.charValue();

1. **Autoboxing**: Autoboxing is the automatic conversion of primitive types to their corresponding wrapper classes. It allows you to assign primitive values directly to wrapper class variables or pass them as arguments to methods that expect wrapper class parameters.

Here's an example of autoboxing:

javaCopy code

int num = 10;

Integer numObj = num; // Autoboxing: int to Integer

2. **Auto unboxing**: Auto unboxing is the automatic conversion of wrapper class objects to their corresponding primitive types. It allows you to extract the primitive value directly from the wrapper class object without explicitly calling a method.

Here's an example of auto unboxing:

```
Integer numObj = 5;

int num = numObj; // Auto unboxing: Integer to int
```

Auto unboxing also applies in expressions or operations involving wrapper class objects:

java

```
Integer numObj = 10;

int result = numObj + 5; // Auto unboxing: Integer to int
```

Generics in Java

Generics in Java provide a way to create reusable code that can work with different data types. It allows you to define classes, interfaces, and methods that can be parameterized with one or more types. Generics add compile-time type safety and eliminate the need for explicit type casting, enhancing the readability and maintainability of the code.

Generics in Java provide a way to create reusable code that can work with different data types. It allows you to define classes, interfaces, and methods that can be parameterized with one or more types. Generics add compile-time type safety and eliminate the need for explicit type casting, enhancing the readability and maintainability of the code.

To understand generics in Java, let's start with an example. Suppose you want to create a class that represents a container for storing elements. Traditionally, without generics, you would define the class like this:

```
public class Container {
    private Object element;

    public void setElement(Object element) {
        this.element = element;
    }

    public Object getElement() {
        return element;
    }
}
```

With generics, you can modify the `Container` class to be parameterized with a type. Here's how it looks:

```
public class Container<T> {
    private T element;

    public void setElement(T element) {
        this.element = element;
    }
    public T getElement() {
        return element;
    }
}
```

In this modified version, `T` is a type parameter that represents the type of the element stored in the container. Now, when you create an instance of `Container`, you can specify the desired type:

```
Container<String> stringContainer = new Container<>();
Container<Integer> intContainer = new Container<>();
```

Generics with Interfaces

Generics are not limited to classes; you can also use them with interfaces and methods. Here's an example of using generics with an interface:

```
public interface List<T> {
    void add(T element);
    T get(int index);
}
```

Generics can also have multiple type parameters. For example:

```
public class Pair<K, V> {
    private K key;
    private V value;
    public Pair(K key, V value) {

        this.key = key;

        this.value = value;

    }



    public K getKey() {

        return key;

    }



    public V getValue() {

        return value;

    }

}

Pair<String, Integer> pair = new Pair<>("key", 10);
```

Example:

```
interface myinterface<T>{
   public void add(T element);
    public T get();

}
class abc implements myinterface <Integer>{
    Integer x;
    public void add(Integer num)  {
        x = num;
    }
    public Integer get(){
        return x;
    }

}

public class generics<T> {
    private T element;
    public void setElement(T element){
        this.element=element;
    }
    public T getElement(){
        return element;
    }


}
class Main{
    public static void main(String[] args) {
        generics<String> o1=new generics<>();
        generics<String> o2=new generics<>();
        o1.setElement("Moin");
        System.out.println(o1.getElement());
        o2.setElement("Zargar");
        System.out.println(o2.getElement());
        abc o3=new abc();
        o3.add(123);
        System.out.println(o3.get());
    }
}
```

Bounded Types in Generics

In Java, bounded types in generics allow you to restrict the type parameter of a generic class or method to a specific range or set of types. This provides compile-time safety by enforcing that only certain types can be used as type arguments.

There are two types of bounded types in Java generics: upper bounded and lower bounded types.

Upper Bounded Types:

An upper bounded type is specified using the extends keyword.

It restricts the type parameter to be a specific type or any of its subclasses.

Syntax: class ClassName<T extends Type> { ... } or void methodName(T extends Type parameter) { ... }

Example: Suppose we have a generic method that accepts a List of objects that extend the Number class. The method can be defined as follows:

java

Copy code

```
public <T extends Number> void processList(List<T> list) {
    // Process the list
}
```

Here, T can be any subclass of Number (e.g., Integer, Double).

Lower Bounded Types:

A lower bounded type is specified using the super keyword.

It restricts the type parameter to be a specific type or any of its superclasses.

Syntax: class ClassName<T super Type> { ... } or void methodName(T super Type parameter) { ... }

Example: Suppose we have a method that accepts a list and adds elements of a specific type or any of its superclasses to the list. The method can be defined as follows:

java

Copy code

```
public void addElements(List<? super Integer> list) {
    list.add(42);
    list.add(10);
}
```

Here, list can be a List of Integer or any superclass of Integer (e.g., Number, Object).

Using bounded types allows you to write more generic and reusable code while maintaining type safety. It ensures that only compatible types are used as type arguments, thereby preventing potential runtime errors.

Wildcard argument in generics (?)

In Java generics, a wildcard argument is denoted by the ? symbol and is used to represent an unknown type. It allows you to write generic code that can operate on different types without specifying the actual type.

There are three types of wildcard arguments: the unbounded wildcard (?), the upper bounded wildcard (? extends Type), and the lower bounded wildcard (? super Type).

Unbounded Wildcard (?):

The unbounded wildcard represents an unknown type.

It allows you to accept any type as a parameterized argument or use any type as an argument when calling a generic method.

Example: Suppose we have a generic method that accepts a List of any type. The method can be defined as follows:

java

Copy code

```
public void processList(List<?> list) {
    // Process the list
}
```

Upper Bounded Wildcard (? extends Type):

The upper bounded wildcard restricts the unknown type to be a specific type or any of its subclasses.

It allows you to accept a parameterized argument that is of the specified type or a subtype of it.

Example: Suppose we have a method that accepts a List of objects that extend the Number class. The method can be defined as follows:

java

Copy code

```
public void processNumbers(List<? extends Number> numbers) {

   // Process the numbers

}
```

Lower Bounded Wildcard (? super Type):

The lower bounded wildcard restricts the unknown type to be a specific type or any of its superclasses.

It allows you to accept a parameterized argument that is of the specified type or a superclass of it.

Example: Suppose we have a method that accepts a list and adds elements of a specific type or any of its superclasses to the list. The method can be defined as follows:

java

Copy code

```
public void addElements(List<? super Integer> list) {

   list.add(42);

   list.add(10);

}
```

We can have non generic method inside a generic method

```
public class Example {
    public <T> void genericMethod(T value) {
        // Generic method logic
        System.out.println("Generic method: " + value.toString());

        // Non-generic method call
        nonGenericMethod();
    }

    public void nonGenericMethod() {
        // Non-generic method logic
        System.out.println("Non-generic method");
    }
}
```

Create a generic interface for stack

```
public interface Stack<E> {
    void push(E element);
    E pop();
    E peek();
    boolean isEmpty();
    int size();
}
```

```java
public class ArrayStack<E> implements Stack<E> {

    private E[] elements;

    private int top;



    public ArrayStack(int capacity) {

        elements = (E[]) new Object[capacity];

        top = -1;

    }



    @Override

    public void push(E element) {

        elements[++top] = element;

    }



    @Override

    public E pop() {

        if (isEmpty()) {

            throw new IllegalStateException("Stack is empty");

        }

        return elements[top--];

    }



    @Override

    public E peek() {
        if (isEmpty()) {
            throw new IllegalStateException("Stack is empty");
        }
        return elements[top];
    }

    @Override
    public boolean isEmpty() {
        return top == -1;
    }

    @Override
    public int size() {
        return top + 1;
    }
}
```

Allowed Syntax:

Generic class declaration:

java

Copy code

```
class MyClass<T> {
    // Class implementation
```

}

Generic class declaration with bounded type parameter:

java

Copy code

```
class MyClass<T extends SomeType> {
    // Class implementation
}
```

Generic class declaration implementing a generic interface:

java

Copy code

```
class MyClass<T> implements SomeInterface<T> {
    // Class implementation
}
```

Generic class declaration extending a generic class:

java

Copy code

```
class MyClass<T> extends SomeClass<T> {
    // Class implementation
}
```

Disallowed Syntax:

Incorrect usage of type parameter when implementing a generic interface:

java

Copy code

```
class MyClass<T> implements SomeInterface<T extends SomeType> {

    // Class implementation

}
```

Explanation: The bounds for a type parameter should be specified immediately after declaring the type parameter, not when implementing an interface.

Incorrect placement of type parameter in the class declaration:

java

Copy code

```
class MyClass implements SomeInterface<T> {

    // Class implementation

}
```

Explanation: If you want to use a type parameter, it must be declared in the class declaration, within angle brackets (<>).

Using a non-existent type parameter:

java

Copy code

```
class MyClass<T> implements SomeInterface<U> {

    // Class implementation

}
```

Explanation: If you want to use a type parameter, it must be declared in the class declaration and cannot reference a non-existent type parameter.

Generic classes inherited by Generic /Non Generic class

Yes, generic classes can be inherited in Java.

When a generic class is inherited, the subclass can either retain the generic type parameter(s) of the superclass or provide its own specific type argument(s) to specialize the generic class further.

Here's an example to illustrate the inheritance of a generic class:

java

Copy code

```
class GenericClass<T> {
    private T value;

    public GenericClass(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}
```

```
class Subclass<T> extends GenericClass<T> {

    public Subclass(T value) {

        super(value);

    }

}
```

In this example, GenericClass<T> is a generic class with a type parameter T. The Subclass<T> extends GenericClass<T> and retains the same type parameter T. It inherits all the members and behavior of the GenericClass class.

You can also specialize the generic class by providing a specific type argument for the subclass:

java

Copy code

```
//subclass is non generic in this case

class Subclass extends GenericClass<Integer> {

    public Subclass(Integer value) {

        super(value);

    }

}
```

In this case, the Subclass provides a specific type argument Integer for the generic class GenericClass. It inherits the behavior of GenericClass<Integer> and only works with Integer values.

Non Generic class extended by generic class

```
class NonGenericClass {
    public void method() {
        System.out.println("Non-generic method");
    }
}

class GenericSubclass<T> extends NonGenericClass {
    private T value;

    public GenericSubclass(T value) {
        this.value = value;
    }

    public void method() {
        System.out.println("Generic method with value: " + value);
    }
}
```

Method overriding in generic class

```
class GenericClass<T> {
    public void method(T value) {
        System.out.println("Generic method with value: " + value);
    }
}

class Subclass<T> extends GenericClass<T> {
    @Override
    public void method(T value) {
        System.out.println("Overridden method with value: " + value);
    }
}
```

Note:

Local variable inference

var x=new Gen<Integer> (9)   (Allowed)

var x=new Gen< > (9)   (Not Allowed)

var y=new Gen (9)   (Allowed)   //y -> object class instance

Static with type parameter T

In Java, it is not possible to use the type parameter T directly in a static context, such as a static method or a static field.

Static members are associated with the class itself rather than with instances of the class. Since type parameters are resolved at the instance level, they cannot

be used in a static context because there is no specific instance to determine the type argument for T.

However, you can still use a type parameter in a static context by introducing it at the class level. Here's an example:

java

Copy code

```
class MyClass<T> {
   private static T staticField; // Compilation error: Cannot make a static
reference to the non-static type T

   public static <T> void staticMethod(T parameter) {
       // Static method implementation
   }
}
```

In the example, trying to use T directly as a static field results in a compilation error. However, you can declare a static method with its own type parameter T within the generic class. This allows you to use T within the scope of that method, even though it is different from the type parameter of the class itself.

Keep in mind that static methods and static fields are not associated with the type arguments of generic classes. They operate at the class level and do not have access to the type parameter T of an instance.

Can we create an Array of type parameter T ?

In Java, it is not directly possible to create an array of a type parameter T because the type erasure mechanism removes the generic type information at runtime. However, there are workarounds to achieve similar functionality using type casting or reflection.

Using type casting:

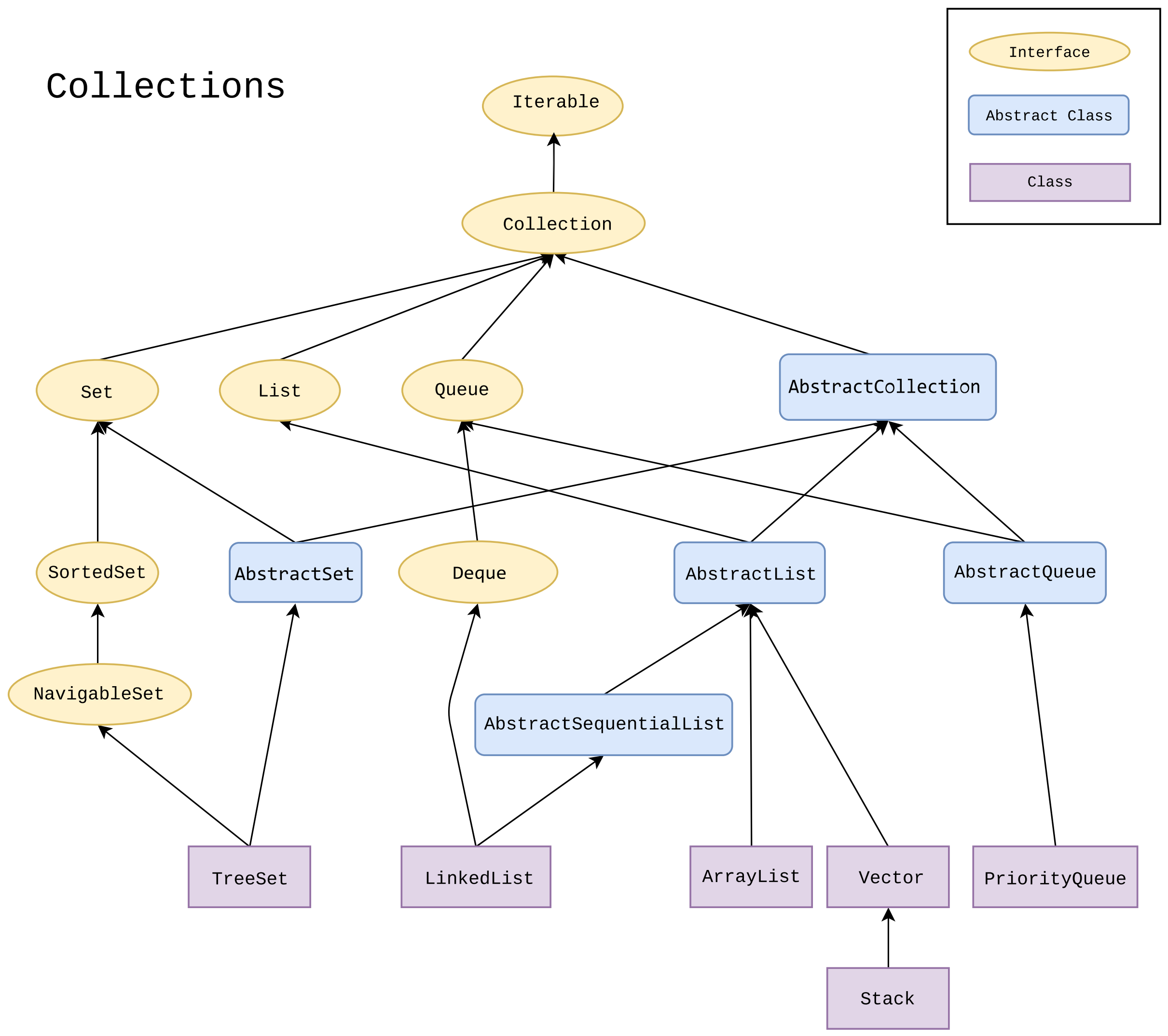
You can create an array of type Object and then cast it to an array of type T:

java

```
T[] array = (T[]) new Object[size];
```

Please note that this approach may generate an unchecked cast warning and can lead to a ClassCastException if the wrong type is assigned to the array elements.

Collections :



Collections in Java refer to a set of classes and interfaces that provide various data structures and algorithms for storing, managing, and manipulating groups of objects. These collections are part of the Java Collections Framework, which is a built-in framework in the Java Standard Library.

The Java Collections Framework includes several core interfaces and classes, such as:

Collection: The root interface that represents a group of objects. Subinterfaces include List, Set, and Queue.

List: An ordered collection that allows duplicate elements. Implementations include ArrayList, LinkedList, and Vector.

Set: A collection that does not allow duplicate elements. Implementations include HashSet, LinkedHashSet, and TreeSet.

Queue: A collection designed for holding elements prior to processing. Implementations include LinkedList, PriorityQueue, and ArrayDeque.

Map: An object that maps keys to values. Implementations include HashMap, LinkedHashMap, TreeMap, and HashTable.

These interfaces and classes provide a wide range of functionality for performing common operations such as adding, removing, and searching for elements, as well as sorting, iterating, and filtering collections.

Collection interface

```
public interface Collection<E> {

   int size();

   boolean isEmpty();
```

```
    boolean contains(Object o);

    Iterator<E> iterator();

    Object[] toArray();

    <T> T[] toArray(T[] a);

    boolean add(E e);

    boolean remove(Object o);

    boolean containsAll(Collection<?> c);

    boolean addAll(Collection<? extends E> c);

    boolean removeAll(Collection<?> c);

    boolean retainAll(Collection<?> c);

    void clear();

}
```

Let's briefly describe the purpose of each method:

size(): Returns the number of elements in the collection.

isEmpty(): Checks if the collection is empty.

contains(Object o): Checks if the collection contains the specified object.

iterator(): Returns an iterator over the elements in the collection.

toArray(): Returns an array containing all the elements of the collection.

toArray(T[] a): Returns an array containing all the elements of the collection, using the provided array if it is large enough.

add(E e): Adds an element to the collection.

remove(Object o): Removes the specified object from the collection.

containsAll(Collection<?> c): Checks if the collection contains all the elements in the specified collection.

addAll(Collection<? extends E> c): Adds all the elements from the specified collection to the current collection.

removeAll(Collection<?> c): Removes all the elements from the current collection that are also present in the specified collection.

retainAll(Collection<?> c): Removes all the elements from the current collection that are not present in the specified collection.

clear(): Removes all elements from the collection.

The Collection interface provides a common set of methods that can be implemented by different collection types, such as ArrayList, HashSet, or LinkedList. Each collection type can provide its own implementation of these methods based on its specific requirements and characteristics.

List Interface :

Introduction to Lists:

The List interface is a sub-interface of the Collection interface and represents an ordered collection of elements.

Lists allow duplicate elements and maintain the insertion order of elements.

Common implementations of the List interface include ArrayList, LinkedList, and Vector.

Creating Lists:

You can create a List instance using the ArrayList class, which internally uses an array to store elements.

Alternatively, you can use the LinkedList class, which implements a doubly-linked list to store elements.

Here's an example of creating a List using ArrayList:

java

Copy code

```java
List<String> list = new ArrayList<>();
```

Adding and Accessing Elements:

Use the add(element) method to append an element to the end of the list.

Use the add(index, element) method to insert an element at a specific index.

Use the get(index) method to retrieve an element at the specified index.

Example:

java

Copy code

```java
list.add("Apple");

list.add(1, "Banana");

String element = list.get(0);  // Retrieves the first element
```

Removing Elements:

Use the remove(index) method to remove an element at the specified index.

Use the remove(element) method to remove the first occurrence of the specified element.

Example:

list.remove(1); // Removes the element at index 1

list.remove("Apple"); // Removes the first occurrence of "Apple"

Updating Elements:

Use the set(index, element) method to replace an element at the specified index.

Example:

java

Copy code

list.set(0, "Orange"); // Replaces the element at index 0 with "Orange"

Searching Elements:

Use the indexOf(element) method to find the index of the first occurrence of the specified element.

Use the lastIndexOf(element) method to find the index of the last occurrence of the specified element.

Example:

int index = list.indexOf("Apple"); // Retrieves the index of the first occurrence of "Apple"

int lastIndex = list.lastIndexOf("Apple"); // Retrieves the index of the last occurrence of "Apple"

Iterating Over a List:

Use a for-each loop or an iterator to iterate over the elements of a list.

Example using a for-each loop:

java

Copy code

```
for (String element : list) {
   System.out.println(element);
}
```

List Iterators:

Use the listIterator() method of a List to obtain a ListIterator.

ListIterators allow bidirectional traversal and additional operations like adding and removing elements during iteration.

Example:

```
ListIterator<String> iterator = list.listIterator();
while (iterator.hasNext()) {
   String element = iterator.next();
   System.out.println(element);
}
```

Sublists:

Use the subList(fromIndex, toIndex) method to get a view of a portion of the list.

The sublist retains the underlying list's structural changes.

Example:

java

Copy code

```
List<String> sublist = list.subList(1, 3);  // Retrieves a sublist from index 1
(inclusive) to 3 (exclusive)
```

Other List Operations:

The List interface provides additional methods like isEmpty(), size(), contains(element), clear(), toArray(), etc., which you can explore in the Java documentation.

That covers the basics of using the List interface in Java. Remember to import the appropriate classes (java.util.List, java.util.ArrayList, java.util.LinkedList, etc.) before using them in your code.

Set Interface

1. Introduction to Sets:

   - The Set interface is a sub-interface of the Collection interface and represents an unordered collection of unique elements.
   - Sets do not allow duplicate elements, and they do not maintain the insertion order of elements.
   - Common implementations of the Set interface include HashSet, TreeSet, and LinkedHashSet.

2. **Creating Sets:**

   - **You can create a Set instance using the HashSet class, which internally uses a hash table to store elements.**
   - **Alternatively, you can use the TreeSet class, which maintains elements in sorted order.**
   - **Here's an example of creating a Set using HashSet:**

Set<String> set = new HashSet<>();

3. **Adding Elements:**

   - **Use the add(element) method to add an element to the set.**
   - **The add operation returns true if the element was successfully added, and false if the element already exists in the set.**
   - **Example:**

javaCopy code

set.add("Apple"); set.add("Banana"); set.add("Apple"); // This element will not be added since "Apple" already exists in the set

4. **Removing Elements:**

   - **Use the remove(element) method to remove a specific element from the set.**
   - **The remove operation returns true if the element was successfully removed, and false if the element does not exist in the set.**
   - **Example:**

set.remove("Apple");

5. **Checking Element Existence:**

- Use the contains(element) method to check if a specific element exists in the set.
- The contains operation returns true if the element is found in the set, and false otherwise.
- Example:

javaCopy code

boolean contains = set.contains("Banana");

6. Set Operations:

- Sets support various set operations like union, intersection, and difference.
- The addAll(collection) method can be used to perform a union of two sets, adding all elements from another collection to the set.
- The retainAll(collection) method can be used to perform an intersection of two sets, retaining only the common elements.
- The removeAll(collection) method can be used to perform a difference of two sets, removing all elements that are also present in another collection.
- Example:

  Set<String> otherSet = new HashSet<>(); otherSet.add("Orange"); set.addAll(otherSet); // Union of sets set.retainAll(otherSet); // Intersection of sets set.removeAll(otherSet); // Difference of sets

7. Iterating Over a Set:

- Use a for-each loop or an iterator to iterate over the elements of a set.
- Example using a for-each loop:

for (String element : set) { System.out.println(element); }

8. Set Implementations:
    - HashSet: Provides constant-time performance for add, remove, and contains operations, but does not guarantee any specific order of elements.
    - TreeSet: Maintains elements in sorted order and provides efficient performance for add, remove, and contains operations (logarithmic time complexity).
    - LinkedHashSet: Maintains the insertion order of elements and provides performance similar to HashSet.
9. Other Set Operations:
    - The Set interface provides additional methods like isEmpty(), size(), clear(), toArray(), etc., which you can explore in the Java documentation.

That covers the basics of using the Set interface in Java. Remember to import the appropriate classes (java.util.Set, java.util.HashSet, java.util.TreeSet, etc.) before using them in your code.

NavigableSet Interface:

- The NavigableSet interface is a sub-interface of the SortedSet interface in Java.
- It extends the SortedSet interface by providing additional navigation methods for accessing elements based on their values.
- NavigableSet implementations include TreeSet.

2. Navigation Methods:

- **lower(element): Returns the greatest element in the set that is strictly less than the given element.**
- **floor(element): Returns the greatest element in the set that is less than or equal to the given element.**
- **ceiling(element): Returns the smallest element in the set that is greater than or equal to the given element.**
- **higher(element): Returns the smallest element in the set that is strictly greater than the given element.**
- **Example:**

NavigableSet<Integer> set = new TreeSet<>(); set.add(1); set.add(2); set.add(3); Integer floorElement = set.floor(2); // Returns 2 Integer ceilingElement = set.ceiling(2); // Returns 2 Integer lowerElement = set.lower(2); // Returns 1 Integer higherElement = set.higher(2); // Returns 3

3. **Subset Views:**

- **The NavigableSet interface provides methods to obtain subsets of elements from the set.**
- **headSet(toElement, inclusive): Returns a view of the portion of the set whose elements are strictly less than the given element. The inclusive parameter determines whether the given element is included in the subset.**
- **tailSet(fromElement, inclusive): Returns a view of the portion of the set whose elements are greater than or equal to the given element. The inclusive parameter determines whether the given element is included in the subset.**
- **Example:**

```
NavigableSet<Integer> set = new TreeSet<>(); set.add(1); set.add(2); set.add(3);
SortedSet<Integer> headSet = set.headSet(3, true); // Returns [1, 2, 3]
SortedSet<Integer> tailSet = set.tailSet(2, false); // Returns [3]
```

4. **Polling Methods:**

   - **pollFirst(): Retrieves and removes the first (lowest) element in the set, or returns null if the set is empty.**
   - **pollLast(): Retrieves and removes the last (highest) element in the set, or returns null if the set is empty.**
   - **Example:**

```
NavigableSet<Integer> set = new TreeSet<>(); set.add(1); set.add(2); Integer
firstElement = set.pollFirst(); // Retrieves and removes 1 Integer lastElement =
set.pollLast(); // Retrieves and removes 2
```

5. **Common Exceptions:**

   - **ClassCastException: This exception is thrown when an incompatible element type is added to a set. For example, adding an object of the wrong type to a set of integers.**
   - **NullPointerException: This exception is thrown when a null element is not allowed in a set and an attempt is made to add a null element.**
   - **IllegalArgumentException: This exception is thrown when an invalid argument is passed to a method. For example, passing a negative value to methods like headSet() or tailSet().**

Queue Interface

The Queue interface in Java represents a collection that holds elements prior to processing. It follows the First-In-First-Out (FIFO) principle, meaning that

elements are processed in the order they were added. The Queue interface provides methods for adding, removing, and inspecting elements. Here's an overview of the Queue interface and its methods:

1. **Declaration:**

javaCopy code

Queue<E> queue = new LinkedList<>();

2. **Common Methods:**
   - **add(element): Adds an element to the queue. Throws an exception if the operation fails.**
   - **offer(element): Adds an element to the queue. Returns true if the operation is successful, false otherwise.**
   - **remove(): Removes and returns the element at the front of the queue. Throws an exception if the queue is empty.**
   - **poll(): Removes and returns the element at the front of the queue. Returns null if the queue is empty.**
   - **element(): Retrieves, but does not remove, the element at the front of the queue. Throws an exception if the queue is empty.**
   - **peek(): Retrieves, but does not remove, the element at the front of the queue. Returns null if the queue is empty.**
3. **Queue Implementations:**
   - **LinkedList: Implements the Queue interface and provides efficient insertion and removal operations.**
   - **PriorityQueue: Implements the Queue interface and provides a priority-based ordering of elements.**
4. **Additional Methods:**

- **In addition to the methods inherited from the Queue interface, the LinkedList and PriorityQueue classes provide some additional methods:**
    - **addAll(collection): Adds all elements from the specified collection to the queue.**
    - **clear(): Removes all elements from the queue.**
    - **size(): Returns the number of elements in the queue.**
    - **isEmpty(): Returns true if the queue is empty, false otherwise.**
    - **contains(element): Returns true if the queue contains the specified element, false otherwise.**

5. **Iterating Over a Queue:**
    - **Since the Queue interface extends the Collection interface, you can use an iterator or a for-each loop to iterate over the elements in the queue.**
    - **Example using a for-each loop:**

```
Queue<String> queue = new LinkedList<>();

queue.add("Apple");

queue.add("Banana");

for (String element : queue) {

   System.out.println(element);

}
```

Dequeue Interface

The Deque interface in Java represents a double-ended queue, which allows the insertion and removal of elements from both ends. It extends the Queue interface and provides additional methods for operations such as inserting and removing elements from the front and back of the deque. Here's an overview of the Deque interface and its methods:

1. **Declaration:**

   javaCopy code

   Deque<E> deque = new LinkedList<>();

2. **Common Methods:**

   - **addFirst(element): Inserts the specified element at the front of the deque. Throws an exception if the operation fails.**
   - **addLast(element): Inserts the specified element at the end of the deque. Throws an exception if the operation fails.**
   - **offerFirst(element): Inserts the specified element at the front of the deque. Returns true if the operation is successful, false otherwise.**
   - **offerLast(element): Inserts the specified element at the end of the deque. Returns true if the operation is successful, false otherwise.**
   - **removeFirst(): Removes and returns the element at the front of the deque. Throws an exception if the deque is empty.**
   - **removeLast(): Removes and returns the element at the end of the deque. Throws an exception if the deque is empty.**
   - **pollFirst(): Removes and returns the element at the front of the deque. Returns null if the deque is empty.**
   - **pollLast(): Removes and returns the element at the end of the deque. Returns null if the deque is empty.**

- getFirst(): Retrieves, but does not remove, the element at the front of the deque. Throws an exception if the deque is empty.
- getLast(): Retrieves, but does not remove, the element at the end of the deque. Throws an exception if the deque is empty.
- peekFirst(): Retrieves, but does not remove, the element at the front of the deque. Returns null if the deque is empty.
- peekLast(): Retrieves, but does not remove, the element at the end of the deque. Returns null if the deque is empty.

3. Deque Implementations:
    - The most commonly used implementation of the Deque interface is LinkedList, which provides efficient insertion and removal operations at both ends of the deque.
    - Another implementation is ArrayDeque, which internally uses an array to store elements and provides fast constant-time operations.
4. Additional Methods:
    - In addition to the methods inherited from the Queue interface, the Deque interface provides some additional methods:
        - offer(element): Inserts the element at the end of the deque, equivalent to offerLast(element).
        - remove(): Removes and returns the element at the front of the deque, equivalent to removeFirst().
        - poll(): Removes and returns the element at the front of the deque, equivalent to pollFirst().
        - element(): Retrieves, but does not remove, the element at the front of the deque, equivalent to getFirst().

- peek(): Retrieves, but does not remove, the element at the front of the deque, equivalent to peekFirst().

Abstract Collection Classes

Abstract collection classes in Java are abstract base classes that provide a partial implementation of the Collection interface. They serve as a foundation for creating custom collection classes by extending them and implementing the remaining abstract methods. Some notable abstract collection classes include AbstractCollection, AbstractList, AbstractSequentialList, and AbstractSet. They provide default implementations for common methods, reducing the effort required to create new collection classes. By extending these abstract classes, you can focus on implementing specific behavior and reuse the common methods provided by the abstract class.

Array List

ArrayList is a class in Java that implements the List interface and provides a dynamic array-like data structure. It is a resizable array that can grow or shrink dynamically as elements are added or removed. Here's a brief overview of ArrayList:

1. Declaration:

```
ArrayList<E> list = new ArrayList<>();
```

2. Common Methods:

   - add(element): Appends the specified element to the end of the list.
   - get(index): Retrieves the element at the specified index.
   - set(index, element): Replaces the element at the specified index with the specified element.

- **remove(index): Removes the element at the specified index.**
- **size(): Returns the number of elements in the list.**
- **isEmpty(): Returns true if the list is empty, false otherwise.**
- **contains(element): Returns true if the list contains the specified element, false otherwise.**
- **clear(): Removes all elements from the list.**

3. **ArrayList Iteration:**
    - **You can iterate over the elements of an ArrayList using a for-each loop or an iterator.**
    - **Example using a for-each loop:**

    javaCopy code

    ArrayList<String> list = new ArrayList<>(); list.add("Apple"); list.add("Banana"); for (String element : list) { System.out.println(element); }

4. **ArrayList with Generics:**
    - **ArrayList supports generics, allowing you to specify the type of elements it can hold.**
    - **Example with generics:**

    javaCopy code

    ArrayList<Integer> numbers = new ArrayList<>(); numbers.add(10); numbers.add(20); int firstNumber = numbers.get(0);

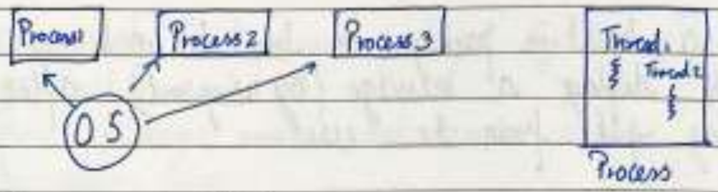
5. **Other ArrayList Methods:**
    - **addAll(collection): Appends all elements from the specified collection to the end of the list.**

- **indexOf(element): Returns the index of the first occurrence of the specified element in the list, or -1 if not found.**
- **lastIndexOf(element): Returns the index of the last occurrence of the specified element in the list, or -1 if not found.**
- **subList(fromIndex, toIndex): Returns a new list containing elements from the specified range.**

MultiThreading :

Chapter 13 - Multithreading

Multiprocessing and multithreading both are used to acheive multitasking



In a nut Shell ...

→ Threads use shared memory area

→ Threads ⇒ Faster Context Switching

→ A Thread is light-weight whereas a process is heavyweight

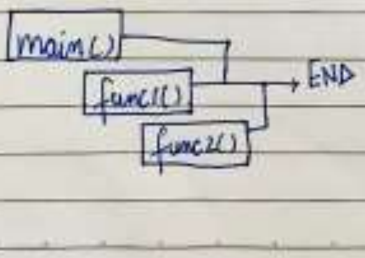
For Example → A word processor can have one thread running in foreground as an editor and another in the background auto saving the document!

Flow of Control in Java

1. Without threading:

$$\begin{tikzcd} \boxed{main()} \arrow[r] & \boxed{func1()} \arrow[r] & \boxed{func2} \arrow[r] & END \end{tikzcd}$$

2. With threading:



Creating a Thread

There are two ways to create a thread in Java.

1. By extending Thread class
2. By implementing Runnable interface

Life cycle of a Thread

$$\begin{tikzcd}
① \boxed{\text{New}} \arrow[d] & \\
② \boxed{\text{Runnable}} \arrow[d] & \boxed{\text{Non Runnable (Blocked)}} ④ \arrow[l] \\
③ \boxed{\text{Running}} \arrow[d] \arrow[ur] & \\
⑤ \boxed{\text{Terminated}} &
\end{tikzcd}$$

① New → Instance of thread created which is not yet started by invoking start()

② Runnable → After invocation of start() & before it is selected to be run by the scheduler.

③ Running → After thread scheduler has selected it.

④ Non Runnable → Thread alive, not eligible to run.

⑤ Terminated → run() method has exitted

The Thread class

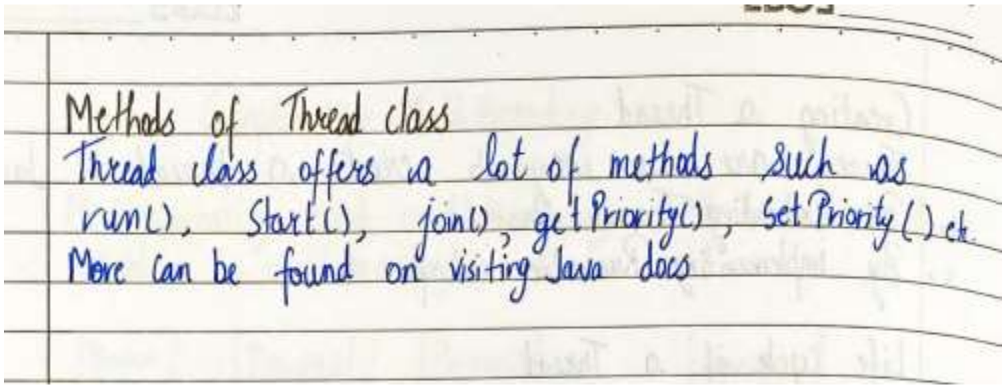
Below are the commonly used constructors of Thread class:

① Thread()

② Thread(String name)

③ Thread(Runnable r)

④ Thread(Runnable r, String name)



Methods of Thread class
Thread class offers a lot of methods such as run(), Start(), join(), get Priority(), Set Priority() etc.
More can be found on visiting Java docs

Creating a Thread by Extending Thread class

Multithreading In Java :

- **Used to maximize the CPU utilization.**
- **We don't want our CPU to be in a free state; for example, Func1() comes into the memory and demands any input/output process. The CPU will need to wait for unit Func1() to complete its input/output operation in such a condition. But, while Func1() completes its I/O operation, the CPU is free and not executing any thread. So, the efficiency of the CPU is decreased in the absence of multithreading.**
- **In the case of multithreading, if a thread demands any I/O operation, then the CPU will let the thread perform its I/O operation, but it will start the execution of a new thread parallelly. So, in this case, two threads are executing at the same time.**

Flow Of Control In Java :

1. Without threading :

```
class ThreadExample{
    public static void main(String[] args) {
        Func1();
        Func2();
    }
}
```

In the above code, you can see that Func1() and Func2() are called inside the main() function. But the execution of Func2() will start only after the completion of the Func1().

2. With threading :

```
class Multi extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi func1=new Multi();
func1.start();
Multi func2=new Multi();
func2.start();
 }
}
```

Again, Func1() and Func2() are called inside the main function, but none of the two functions is waiting for the execution of the other function. Both the functions are getting executed concurrently.

Ways To Create A Thread In Java

1. By extending the thread class
2. By implementing a Runnable interface

Let's see how we can create a thread by extending the thread class.

Extending Thread Class :

To create a thread using the thread class, we need to extend the thread class. Java's multithreading system is based on the thread class.

```
class MyThread extends Thread{
    @Override
    public void run(){
                          //code that we want to get executed
on running the thread
        }
    }
```

Copy

- In the above code, we're first inheriting the Thread class and then overriding the run() method.
- The code you want to execute on the thread's execution goes inside the run() method.

```
class MyThread extends Thread{
    @Override
    public void run(){
        int i =0;
        while(i<40000){
            System.out.println("My Cooking Thread is
Running");
            System.out.println("I am happy!");
            i++;
        }
    }
}
public class cwh_70 {
    public static void main(String[] args) {
    MyThread t1 = new MyThread();
    t1.start();
    }
}
```

In order to execute the thread, the start() method is used. start() is called on the object of the MyThread class. It automatically calls the run() method, and a new stack is provided to the thread. So, that's how you easily create threads by extending the thread class in Java.

```
class MyThread1 extends Thread{
    @Override
    public void run(){
        int i =0;
        while(i<40000){
            System.out.println("Thread1 is running");

            i++;
        }
    }
}

class MyThread2 extends Thread{
    @Override
```

```java
    public void run(){
        int i =0;
        while(i<40000){
            System.out.println("Thread 2 is running");

            i++;
        }
    }
}

public class cwh_70 {
    public static void main(String[] args) {
    MyThread1 t1 = new MyThread1();
    MyThread2 t2 = new MyThread2();
    t1.start();
    t2.start();

    }
}
```

Output :

```
Thread 2 is running

Thread 1 is running

Thread 2 is running

Thread 1 is running

Thread 1 is running
```

Thread 2 is running

...

Since both threads are running concurrently, their output statements will be intermixed, and the exact sequence of lines may differ in each execution. The threads will continue executing in a concurrent manner until the while loop condition becomes false (i.e., `i` reaches 40000 in both threads).

Creating a Java Thread Using Runnable Interface

Steps To Create A Java Thread Using Runnable Interface:

1. Create a class and implement the `Runnable` interface by using the implements keyword.
2. Override the `run()` method inside the implementer class.
3. Create an object of the implementer class in the `main()` method.
4. Instantiate the `Thread` class and pass the object to the `Thread` constructor.
5. Call `start()` on the thread. `start()` will call the `run()` method.

Example :

```
classs t1 implements Runnable{
       @Override
       public void run(){
       System.out.println("Thread is running");
          }
}

public class ClassName{
    public static void main(String[] args) {
          t1 obj1 = new t1();
```

```
        Thread t = new Thread(obj1);
        t.start();
    }
}
```

1. `class t1` is implementing the `Runnable` interface.
2. Overriding of the `run()` method is done inside the `t1 class`.
3. In the `main()` method, `obj1,` an object of the `t1 class,` is created.
4. The constructor of the `Thread` class accepts the `Runnable` instance as an argument, so `obj1` is passed to the constructor of the Thread class.
5. Finally, the `start()` method is called on the thread that will call the `run()` method internally, and the thread's execution will begin.

Runnable Interface Vs Extending Thread Class :

Since we've discussed both the ways to create a thread in Java. There might be a question in your mind that should we use the Runnable interface or Thread class to implement a thread in Java. Let me answer this question for you. The `Runnable` interface is preferred over extending the `Thread` class because of the following reasons :

1. As multiple inheritance is not supported in Java, it is impossible to extend the Thread class if your class had already extended some other class.
2. While implementing Runnable, we do not modify or change the thread's behavior.
3. More memory is required while extending the Thread class because each thread creates a unique object.
4. Less memory is required while implementing Runnable because multiple threads share the same object.

```
class MyThreadRunnable1 implements Runnable {
    public void run() {
        // Print the message multiple times
        for (int i = 0; i < 100; i++) {
            System.out.println("I am a thread 1 not a threat
1");
        }
    }
}

class MyThreadRunnable2 implements Runnable {
    public void run() {
        // Print the message multiple times
        for (int i = 0; i < 100; i++) {
            System.out.println("I am a thread 2 not a threat
2");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        // Create instances of MyThreadRunnable1 and
MyThreadRunnable2
        MyThreadRunnable1 thread1 = new MyThreadRunnable1();
        MyThreadRunnable2 thread2 = new MyThreadRunnable2();

        // Create Thread objects and pass the instances of
MyThreadRunnable1 and MyThreadRunnable2
        Thread t1 = new Thread(thread1);
        Thread t2 = new Thread(thread2);
        // Start the threads
        t1.start();
        t2.start();
    }
}
```

When you run the code, you will see the messages printed by both threads interleaved with each other since they are running concurrently.

Constructors from Thread class in Java

The Thread class

Below are the commonly used constructors of the thread class:

1. Thread ( )
2. Thread (string)
3. Thread (Runnable r)
4. Thread (Runnable r, String name)

```
class MyThr extends Thread{
    public MyThr(String name){
        super(name);
    }
    public void run(){
        int i = 34;
        System.out.println("Thank you");
//          while(true){
//              System.out.println("I am a thread");
//          }
```

```
    }
}
public class cwh_73_thread_constructor {
    public static void main(String[] args) {
    MyThr t1 = new MyThr("Moin");
    MyThr t2 = new MyThr("Zargar");
    t1.start();
    t2.start();
    System.out.println("The id of the thread t is " + t1.getId());
    System.out.println("The name of the thread t is " + t1.getName());
    System.out.println("The id of the thread t is " + t2.getId());
    System.out.println("The name of the thread t is " + t2.getName());

    }
}
```

Java Thread Priorities

In a Multithreading environment, all the threads which are ready and waiting to be executed are present in the Ready queue. The thread scheduler is responsible for assigning the processor to a thread. But the question is on what basis the thread scheduler decides that a particular thread will run before other threads. Here comes the concept of priority in the picture.

1. Every single thread created in Java has some priority associated with it.
2. The programmer can explicitly assign the priority to the thread. Otherwise, JVM automatically assigns priority while creating the thread.
3. In Java, we can specify the priority of each thread relative to other threads. Those threads having higher priorities get greater access to the available resources than lower priorities threads.
4. Thread scheduler will use priorities while allocating processor.
5. The valid range of thread priorities is 1 to 10 (but not 0 to 10), where 1 is the least priority, and 10 is the higher priority.
6. If there is more than one thread of the same priority in the queue, then the thread scheduler picks any one of them to execute.

7. The following static final integer constants are defined in the Thread class:

- **MIN_PRIORITY**: Minimum priority that a thread can have. Value is 1.
- **NORM_PRIORITY**: This is the default priority automatically assigned by JVM to a thread if a programmer does not explicitly set the priority of that thread. Value is 5.
- **MAX_PRIORITY**: Maximum priority that a thread can have. Value is 10.

Priority Methods In Java :

1. setPriority():
   - This method is used to set the priority of a thread. IllegalArgumentException is thrown if the priority given by the user is out of the range [1,10].

   Syntax :

```
public final void setPriority(int x)   // x is the priority [1,10] that is to be set for the thread.
```

2. getPriority():
   - This method is used to display the priority of a given thread.

   Syntax :

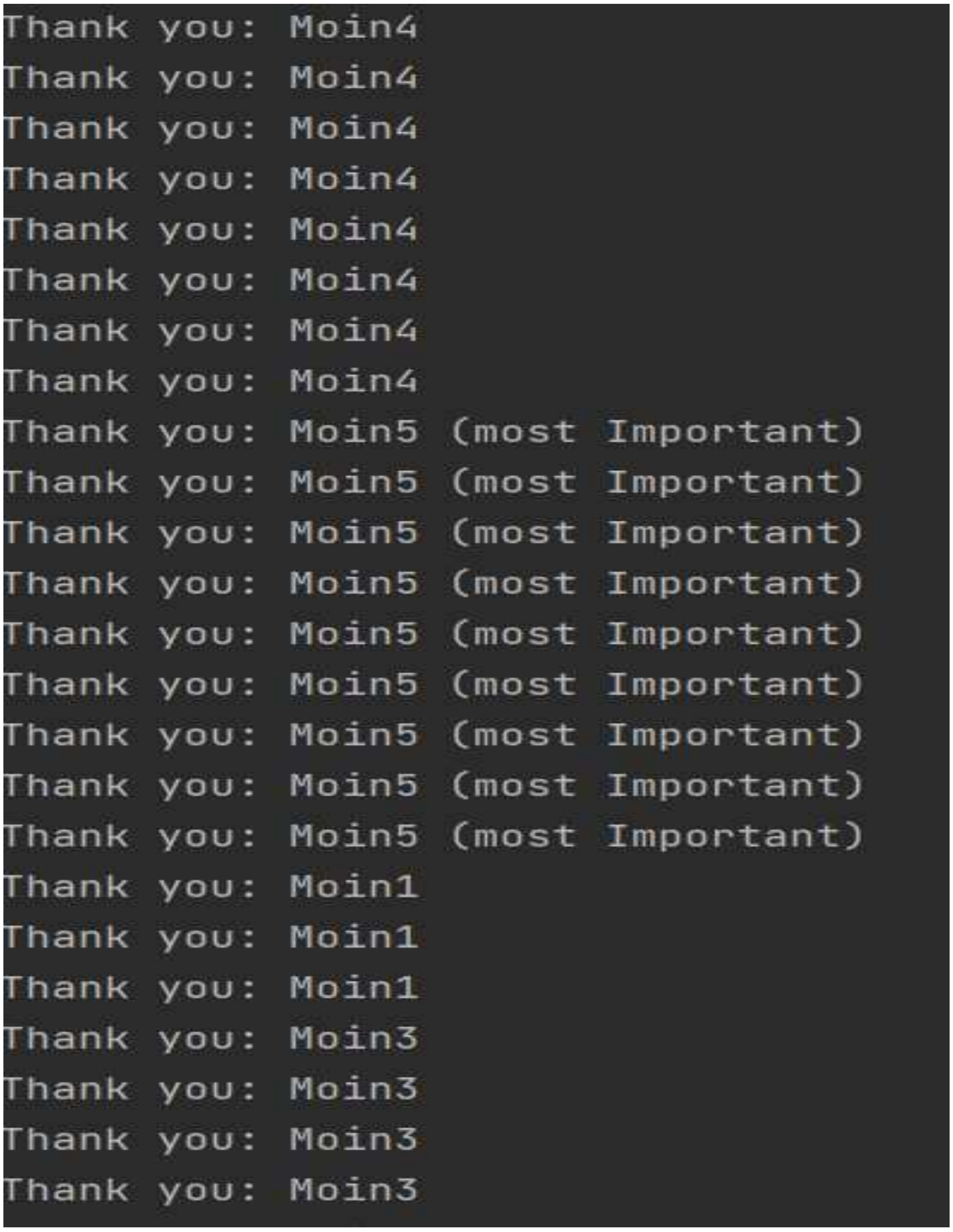
```
t1.getPriority() // Will return the priortity of the t1 thread.
```

Code :

```
     class MyThr1 extends Thread{
  public MyThr1(String name){
      super(name);
```

```
    }
    public void run(){
        int i = 34;

        while(i<=100){
//
            System.out.println("Thank you: " + this.getName());
            i++;
        }

    }
}

public class cwh_74_thread_priorities {
    public static void main(String[] args) {
        // Ready Queue: T1 T2 T3 T4 T5
        MyThr1 t1 = new MyThr1("Moin1");
        MyThr1 t2 = new MyThr1("Moin2");
        MyThr1 t3 = new MyThr1("Moin3");
        MyThr1 t4 = new MyThr1("Moin4");
        MyThr1 t5 = new MyThr1("Moin5 (most Important)");
        t5.setPriority(Thread.MAX_PRIORITY);
        t1.setPriority(Thread.MIN_PRIORITY);
        t2.setPriority(Thread.MIN_PRIORITY);
        t3.setPriority(Thread.MIN_PRIORITY);
        t4.setPriority(Thread.MIN_PRIORITY);
        t5.setPriority(Thread.MIN_PRIORITY);

        t1.start();
        t2.start();
        t3.start();
        t4.start();
        t5.start();

    }
}
```

Output :



```
Thank you: Moin4
Thank you: Moin4
Thank you: Moin4
Thank you: Moin4
Thank you: Moin4
Thank you: Moin4
Thank you: Moin4
Thank you: Moin4
Thank you: Moin5 (most Important)
Thank you: Moin5 (most Important)
Thank you: Moin5 (most Important)
Thank you: Moin5 (most Important)
Thank you: Moin5 (most Important)
Thank you: Moin5 (most Important)
Thank you: Moin5 (most Important)
Thank you: Moin5 (most Important)
Thank you: Moin5 (most Important)
Thank you: Moin1
Thank you: Moin1
Thank you: Moin1
Thank you: Moin3
Thank you: Moin3
Thank you: Moin3
Thank you: Moin3
```

Java Thread Methods

Join() method In Java :

- The join() method in Java allows one thread to wait until the execution of some other specified thread is completed.
- If t is a Thread object whose thread is currently executing, then t.join() causes the current thread to pause execution until t's thread terminates.
- Join() method puts the current thread on wait until the thread on which it is called is dead.

Syntax :

```
public final void join()
```

You can also specify the time for which you need to wait for the execution of a particular thread by using the Join() method. Syntax :

```
public final void join(long millis)
```

Sleep() Method :

- The sleep() method in Java is useful to put a thread to sleep for a specified amount of time.
- When we put a thread to sleep, the thread scheduler picks and executes another thread in the queue.
- Sleep() method returns void.
- sleep() method can be used for any thread, including the main() thread also.

Syntax :

- public static void sleep(long milliseconds)throws InterruptedException
- public static void sleep(long milliseconds, int nanos)throws InterruptedException

Parameters Passed To Sleep() Method :

1. long millisecond: Time in milliseconds for which thread will sleep.
2. nanos : Ranges from [0,999999]. Additional time in nanoseconds.

Example :

```java
import java.io.*;
import java.lang.Thread;
public class cwh {
    public static void main(String[] args)
    {
        try {
            for (int i = 1; i <=5; i++) {
                Thread.sleep(2000);
                System.out.println(i);
            }
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

In the above example, the main() method will be put to sleep for 2 seconds every time the for loop executes.

Output :

```
1
2
3
4
5
```

Interrupt() method :

- A thread in a sleeping or waiting state can be interrupted by another thread with the help of the interrupt() method of the Thread class.
- The interrupt() method throws InterruptedException.
- The interrupt() method will not throw the InterruptedException if the thread is not in the sleeping/blocked state, but the interrupt flag will be changed to true.

Syntax :

```
Public void interrupt()
```

Different scenarios where Interrupt() method can be used:

Case 1: Interrupting a thread that doesn't stop working :

```
class CWH1 extends Thread{
    public void run(){
        try {
            for (int i=0;i<5;i++){
                System.out.println("Child Thread");
                Thread.sleep(4000); /* Child thread is put to
sleep for 4000ms. As soon as child thread goes to sleep main
thread interrupts it. And, InterruptedException is generated
which is handled by the catch block. */


            }
        }
        catch (InterruptedException e)
        {
            System.out.println("Child Thread Interrupted");
        }
      System.out.println("Thread is running");
    }
}

public class CWH extends Thread{
    public static void main(String[] args) {
            CWH1 t= new CWH1();
            t.start();
            t.interrupt();
           System.out.println("Main Thread");

    }
}
```

Copy

In the above code, the for loop runs for the first time, but the child thread is put to sleep after that. So, the main() method interrupts the child thread, and InterruptedException is generated. Here, the child thread comes out of the sleeping state, but it does not stop working.

Output :

```
Main Thread
Child Thread
Child Thread Interrupted
Thread is running
```

Case 2: Interrupting a thread that works normally :

```
class CWH1 extends Thread{
    public void run(){
        for (int i=0;i<10;++i){
                System.out.println(i);
            }
        }
}

public class CWH extends Thread{
        public static void main(String[] args) {
                CWH1 t= new CWH1();
                t.start();
                t.interrupt();
                System.out.println("Main Thread");

        }
}
```

Here the thread works normally because no exception occurred during the thread's execution, so the interrupt() only sets the value of the thread flag to true.

Output :

```
0
1
2
3
4
5
6
7
8
9
```

Important methods related to threads in Java:

1. `isAlive()`: This method checks whether a thread is still alive or has finished its execution. It returns `true` if the thread is alive and `false` otherwise.
2. `notify()`: This method wakes up a single thread that is waiting on the object's monitor. If multiple threads are waiting, it is arbitrary which one is awakened.
3. `notifyAll()`: This method wakes up all threads that are waiting on the object's monitor. The awakened threads will compete for the object's lock.
4. `wait()`: This method causes the current thread to wait until another thread invokes the `notify()` or `notifyAll()` method on the same object. The thread releases the object's monitor and enters a waiting state.
5. `sleep(long millis)`: This method causes the currently executing thread to pause for the specified number of milliseconds. It is commonly used for adding delays or controlling the timing of thread execution.
6. `yield()`: This method suggests to the thread scheduler that the current thread is willing to yield its current use of the CPU. It allows other threads with the same priority to run.
7. `join()`: This method allows one thread to wait for the completion of another thread. When a thread calls `join()` on another thread, it will pause its execution and wait until the other thread finishes before resuming.
8. `interrupt()`: This method interrupts a thread, causing it to stop execution if it's in a blocking or waiting state. It can be used to gracefully terminate a thread or to communicate with a running thread.
9. `isInterrupted()`: This method checks whether a thread has been interrupted. It returns `true` if the thread has been interrupted and `false` otherwise.
10. `setPriority(int priority)`: This method sets the priority of a thread. Threads with higher priority are more likely to be scheduled for execution by the thread scheduler.
11. `getPriority()`: This method returns the priority of a thread.
12. `setDaemon(boolean on)`: This method marks a thread as either a daemon thread or a user thread. Daemon threads are automatically terminated when all user threads have finished.

Supurious wakeup

Spurious wakeup refers to a situation in multithreading where a thread wakes up from its wait state without any explicit notification or signal. This can occur even if no other thread has called the `notify()` or `notifyAll()` methods on the object the thread is waiting on.

The Java documentation for `Object.wait()` method states:

"A thread can also wake up without being notified, interrupted, or timing out, a so-called spurious wakeup. While this will rarely occur in practice, applications must guard against it by testing for the condition that should have caused the thread to be awakened, and continuing to wait if the condition is not satisfied."

In other words, although rare, there can be situations where a waiting thread resumes its execution without any external signal. This can be due to various factors, including underlying JVM or operating system behavior.

To handle spurious wakeups, it is recommended to use a loop combined with a condition check when waiting for a particular condition to be satisfied. The loop ensures that the thread rechecks the condition after waking up and goes back to waiting if the condition is not met. This guards against false wakeups and ensures that the thread waits until the desired condition is truly satisfied.

Applets

Applets are a type of Java program that are designed to be embedded within web pages and run on the client-side. They were widely used in the early days of the internet for creating interactive web content. However, with the introduction of modern web technologies, such as HTML5 and JavaScript, the usage of applets has significantly decreased.

Structure of an Applet:

An applet is a subclass of the java.applet.Applet class. It extends the Applet class to inherit the basic functionality required for an applet. Here's a simple structure of an applet:

java

Copy code

```
import java.applet.Applet;
import java.awt.Graphics;

public class MyFirstApplet extends Applet {
    // Applet code here
}
```

Applet Lifecycle:

Applets have a well-defined lifecycle that includes the following methods:

init(): This method is called when the applet is initialized. It is used for one-time initialization tasks.

start(): This method is called when the applet is started or resumed.

stop(): This method is called when the applet is stopped or paused.

destroy(): This method is called when the applet is about to be destroyed.

paint(Graphics g): This method is called whenever the applet needs to be redrawn.

You can override these methods in your applet class to define the desired behavior.

Applet Display:

The paint(Graphics g) method is responsible for rendering the applet's visual content. It receives a Graphics object as a parameter, which you can use to draw shapes, images, text, etc. The Graphics class provides various methods like drawLine(), drawRect(), drawString(), etc., to perform drawing operations.

Embedding an Applet:

To embed an applet in an HTML page, you can use the <applet> tag. Here's an example:

html

Copy code

```
<html>

<head>

  <title>My Applet</title>

</head>

<body>

  <applet code="MyFirstApplet.class" width="300" height="200">

    Your browser does not support the <code>applet</code> tag.

  </applet>

</body>

</html>
```

In this example, the code attribute specifies the applet class file (e.g., MyFirstApplet.class). The width and height attributes specify the dimensions of the applet display area.

Running Applets:

Traditionally, applets were run in web browsers with Java Plugin support. However, most modern browsers have deprecated or removed support for Java applets due to security concerns. As a result, running applets directly in web browsers may not be possible in many cases. To run applets for testing or learning purposes, you can use Java development environments like Eclipse or NetBeans, which provide built-in applet support.

AWT (Abstract Window Toolkit)

AWT (Abstract Window Toolkit) is a Java GUI (Graphical User Interface) toolkit used for creating desktop applications. It provides a set of classes and methods for building user interfaces with components like buttons, labels, menus, and text fields. AWT relies on the native windowing system of the operating system for rendering.

Key Features of AWT:

1. Component classes: AWT offers a variety of classes representing GUI components.
2. Layout managers: AWT includes layout managers for arranging components within containers.
3. Event handling: A robust event handling mechanism allows developers to respond to user actions.
4. Graphics and drawing: AWT provides methods for drawing shapes, images, and text.

Advantages of AWT:

1. Platform independence: AWT allows the creation of platform-independent user interfaces.
2. Simplicity: AWT offers a straightforward API for creating basic GUI applications.
3. Native integration: AWT components blend well with the host operating system's look and feel.

Limitations of AWT:

1. Limited component set: AWT's component set is relatively small compared to other GUI toolkits.
2. Less customizable: AWT components have limited customization options.
3. Performance: AWT's reliance on the native windowing system can affect performance.

Usage :

1. Importing the AWT Package: To use AWT, you need to import the necessary classes from the `java.awt` package. Add the following import statement at the beginning of your Java file:

WT Package:

```
import java.awt.*;
```

2. **Creating a Basic Frame:**

The Frame class represents a top-level window with a title and border. You can create a basic frame using the following code:

```
Frame frame = new Frame("My Frame");
frame.setSize(400, 300); // Set frame dimensions
frame.setVisible(true); // Make the frame visible
```

Components :

1. **Button**

```
import java.awt.*;
import java.awt.event.*;

public class ButtonExample {
    public static void main(String[] args) {
        // Create a Frame (window) to hold the components
        Frame frame = new Frame("Button Example");

        // Create a Button
        Button button = new Button("Click Me");

        // Add ActionListener to handle button click events
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("Button clicked!");
            }
        });

        // Add the button to the Frame
        frame.add(button);

        // Set Frame size and visibility
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

2. Label

```
import java.awt.*;

public class ButtonExample {
    public static void main(String[] args) {
        // Create a Frame (window) to hold the components
        Frame frame = new Frame("Label Example");

        // Create a Label
        Label label = new Label("Hello, World!");

        // Add the label to the Frame
        frame.add(label);

        // Set Frame size and visibility
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

3. Text Field

```
import java.awt.*;

public class ButtonExample {
    public static void main(String[] args) {
        // Create a Frame (window) to hold the components
        Frame frame = new Frame("TextField Example");

        // Create a TextField
        TextField textField = new TextField();

        // Add the TextField to the Frame
        frame.add(textField);

        // Set Frame size and visibility
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

4. Text Area

```java
import java.awt.*;

public class ButtonExample {
    public static void main(String[] args) {
        // Create a Frame (window) to hold the components
        Frame frame = new Frame("TextArea Example");

        // Create a TextArea
        TextArea textArea = new TextArea();

        // Add the TextArea to the Frame
        frame.add(textArea);

        // Set Frame size and visibility
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

5. CheckBox

```java
import java.awt.*;

public class ButtonExample {
    public static void main(String[] args) {
        // Create a Frame (window) to hold the components
        Frame frame = new Frame("Checkbox Example");

        // Create Checkbox components
        Checkbox checkbox1 = new Checkbox("Option 1");
        Checkbox checkbox2 = new Checkbox("Option 2");

        // Add the Checkboxes to the Frame
        frame.add(checkbox1);
        frame.add(checkbox2);

        // Set Frame size and visibility
        frame.setSize(300, 200);
        frame.setLayout(new FlowLayout());
        frame.setVisible(true);
    }
}
```

6. Radio Button

```java
import java.awt.*;

public class RadioButtonExample {
    public static void main(String[] args) {
        // Create a Frame (window) to hold the components
        Frame frame = new Frame("RadioButton Example");

        // Create RadioButton components
        CheckboxGroup checkboxGroup = new CheckboxGroup();
        Checkbox radioButton1 = new Checkbox("Option 1", 
checkboxGroup, false);
        Checkbox radioButton2 = new Checkbox("Option 2", 
checkboxGroup, false);

        // Add the RadioButtons to the Frame
        frame.add(radioButton1);
        frame.add(radioButton2);

        // Set Frame size and visibility
        frame.setSize(300, 200);
        frame.setLayout(new FlowLayout());
        frame.setVisible(true);
    }
}
```

7. Choice

```java
import java.awt.*;

public class ChoiceExample {
    public static void main(String[] args) {
        // Create a Frame (window) to hold the components
        Frame frame = new Frame("Choice Example");

        // Create a Choice component
        Choice choice = new Choice();

        // Add items to the Choice
        choice.add("Option 1");
        choice.add("Option 2");
        choice.add("Option 3");
        // Add the Choice to the Frame
        frame.add(choice);

        // Set Frame size and visibility
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

1. Frame:
   - Frame is a top-level container that represents a window with a title bar and borders.
   - It serves as the main container for AWT applications and can hold other components.
   - Frames can be resized, maximized, minimized, and closed by the user.
   - Example usage: Creating the main window for an application.

```java
import java.awt.*;

public class FrameExample {
    public static void main(String[] args) {
        // Create a Frame
        Frame frame = new Frame("Frame Example");

        // Set Frame size and visibility
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

2. Panel:
   - Panel is a container class that provides an area to group and organize other components.
   - It does not have a title bar or borders like a frame.
   - Panels are often used to group related components together.
   - Example usage: Creating sections or sections within a frame.

```java
import java.awt.*;

public class PanelExample {
    public static void main(String[] args) {
        // Create a Frame
        Frame frame = new Frame("Panel Example");

        // Create a Panel
        Panel panel = new Panel();
        panel.setBackground(Color.LIGHT_GRAY);

        // Add components to the Panel
        panel.add(new Button("Button 1"));
        panel.add(new Button("Button 2"));
        panel.add(new Button("Button 3"));

        // Add the Panel to the Frame
        frame.add(panel);

        // Set Frame size and visibility
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

3. Window:

- Window is a top-level container class that represents a standalone window without borders.
- It is similar to a frame but does not have a title bar or system-provided decorations.
- Windows are often used for custom pop-up windows or dialogs.
- Example usage: Creating custom windows or dialog boxes.

```
import java.awt.*;

public class WindowExample {
    public static void main(String[] args) {
        // Create a Window
        Window window = new Window(new Frame("Window Example"));

        // Set Window size and visibility
        window.setSize(300, 200);
        window.setVisible(true);
    }
}
```

4. Dialog:

- Dialog is a top-level container class that represents a pop-up window with a title bar and borders.
- It is typically used to prompt the user for input or display messages.
- Dialogs can be modal (blocks input to other windows) or non-modal (allows interaction with other windows).
- Example usage: Creating pop-up dialogs for user interaction or notifications.

```
import java.awt.*;

public class DialogExample {
    public static void main(String[] args) {
        // Create a Dialog
        Dialog dialog = new Dialog(new Frame("Dialog Example"), "Dialog",
true);

        // Add components to the Dialog
        dialog.add(new Label("This is a dialog"));

        // Set Dialog size and visibility
        dialog.setSize(300, 200);
        dialog.setVisible(true);
    }
}
```

5. ScrollPane:
    - ScrollPane is a container class that adds scroll bars to its content when it exceeds the visible area.
    - It is useful when dealing with components or content that may not fit within the available space.
    - Scroll panes can hold a single component or multiple components using other container classes.
    - Example usage: Wrapping components with scrollable content, such as large text areas or tables.

```
import java.awt.*;

public class ScrollPaneExample {
    public static void main(String[] args) {
        // Create a Frame
        Frame frame = new Frame("ScrollPane Example");

        // Create a ScrollPane
        ScrollPane scrollPane = new ScrollPane();

        // Add a large TextArea to the ScrollPane
        TextArea textArea = new TextArea(10, 30);
        scrollPane.add(textArea);

        // Add the ScrollPane to the Frame
        frame.add(scrollPane);

        // Set Frame size and visibility
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

6. CardLayout:

- CardLayout is a layout manager that allows multiple components to be stacked on top of each other.
- Only one component is visible at a time, and you can switch between them using methods like next() or show().
- It is useful for creating multi-step wizards or displaying different views in a single container.
- Example usage: Implementing multi-panel interfaces with distinct screens or steps.

7.
```
import java.awt.*;
import java.awt.event.*;

public class ButtonExample {
    public static void main(String[] args) {
        // Create a Frame
        Frame frame = new Frame("CardLayout Example");

        // Create a CardLayout and Panel
        CardLayout cardLayout = new CardLayout();
        Panel panel = new Panel();
        panel.setLayout(cardLayout);

        // Add components to the Panel with different names
        panel.add(new Label("Card 1"), "card1");
        panel.add(new Label("Card 2"), "card2");
        panel.add(new Label("Card 3"), "card3");

        // Create a Button and ActionListener to switch between cards
        Button button = new Button("Next");
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                cardLayout.next(panel);
            }
        });

        // Add the Panel and Button to the Frame
        frame.add(panel, BorderLayout.CENTER);
        frame.add(button, BorderLayout.SOUTH);

        // Set Frame size and visibility
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

Layouts :

1. FlowLayout:

- FlowLayout arranges components in a left-to-right flow, wrapping to the next row when it reaches the container's edge.
- Components are aligned horizontally and vertically centered within each row.
- It is the default layout manager for Panels.
- Example usage: Creating a simple linear arrangement of components.

```java
import java.awt.*;

public class FlowLayoutExample {
    public static void main(String[] args) {
        // Create a Frame
        Frame frame = new Frame("FlowLayout Example");

        // Set FlowLayout for the Frame
        frame.setLayout(new FlowLayout());

        // Add components to the Frame
        frame.add(new Button("Button 1"));
        frame.add(new Button("Button 2"));
        frame.add(new Button("Button 3"));

        // Set Frame size and visibility
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

2. BorderLayout:

- BorderLayout divides the container into five regions: North, South, East, West, and Center.
- The Center region takes up the remaining space, while the other regions have fixed widths or heights.
- Components added to a specific region will expand to fill that region.
- Example usage: Creating a main window with distinct sections or panels.

```java
import java.awt.*;

public class BorderLayoutExample {
    public static void main(String[] args) {
        // Create a Frame
        Frame frame = new Frame("BorderLayout Example");

        // Set BorderLayout for the Frame
        frame.setLayout(new BorderLayout());
        // Add components to different regions
        frame.add(new Button("North"), BorderLayout.NORTH);
        frame.add(new Button("South"), BorderLayout.SOUTH);
        frame.add(new Button("East"), BorderLayout.EAST);
        frame.add(new Button("West"), BorderLayout.WEST);
        frame.add(new Button("Center"), BorderLayout.CENTER);

        // Set Frame size and visibility
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

3. GridLayout:

- GridLayout organizes components in a grid of rows and columns.
- All cells in the grid have the same size, and components are placed sequentially, row by row.
- The number of rows and columns can be specified when creating the layout.
- Example usage: Creating a grid-like arrangement of components, such as a calculator or board game.

```
import java.awt.*;

public class ButtonExample {
    public static void main(String[] args) {
        // Create a Frame
        Frame frame = new Frame("GridLayout Example");

        // Set GridLayout for the Frame
        frame.setLayout(new GridLayout(2, 3));

        // Add components to the Frame
        frame.add(new Button("Button 1"));
        frame.add(new Button("Button 2"));
        frame.add(new Button("Button 3"));
        frame.add(new Button("Button 4"));
        frame.add(new Button("Button 5"));
        frame.add(new Button("Button 6"));

        // Set Frame size and visibility
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

BoxLayout:

- BoxLayout arranges components in a single row or column, aligning them sequentially.

- Components can have fixed or variable sizes, and you can specify their alignment within the container.
- It is useful for creating horizontal or vertical stacks of components with flexible sizing.
- Example usage: Creating simple layouts with a linear arrangement of components.

-
  ```
  import java.awt.*;

  public class BoxLayoutExample {
      public static void main(String[] args) {
          // Create a Frame
          Frame frame = new Frame("BoxLayout Example");

          // Create a Panel
          Panel panel = new Panel();

          // Set BoxLayout for the Panel
          panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));

          // Add components to the Panel
          panel.add(new Button("Button 1"));
          panel.add(new Button("Button 2"));
          panel.add(new Button("Button 3"));

          // Add the Panel to the Frame
          frame.add(panel);

          // Set Frame size and visibility
          frame.setSize(300, 200);
          frame.setVisible(true);
      }
  }
  ```

Event Handling in AWT

Event handling in AWT allows you to respond to user interactions such as button clicks, mouse movements, and keyboard input. Here are explanations of some commonly used event classes and interfaces in AWT:

1. ActionListener:
   - ActionListener is an interface used for handling action events, typically generated by components like buttons.
   - It contains a single method actionPerformed(ActionEvent e) that is invoked when an action event occurs.

- To handle action events, you implement the ActionListener interface and register it with the component using addActionListener().

```java
import java.awt.*;
import java.awt.event.*;

public class ActionListenerExample {
    public static void main(String[] args) {
        // Create a Frame
        Frame frame = new Frame("ActionListener Example");

        // Create a Button
        Button button = new Button("Click Me");

        // Add an ActionListener to the Button
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("Button clicked!");
            }
        });

        // Add the Button to the Frame
        frame.add(button);

        // Set Frame size and visibility
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

2. MouseListener:

- MouseListener is an interface used for handling mouse events, such as mouse clicks, movements, and button presses.
- It includes methods like mouseClicked(MouseEvent e), mouseEntered(MouseEvent e), and mouseExited(MouseEvent e).
- To handle mouse events, you implement the MouseListener interface and register it with the component using addMouseListener().

```java
import java.awt.*;
import java.awt.event.*;

public class MouseListenerExample {
    public static void main(String[] args) {
        // Create a Frame
        Frame frame = new Frame("MouseListener Example");

        // Create a Label
        Label label = new Label("Click or hover over me");
        // Add a MouseListener to the Label
        label.addMouseListener(new MouseListener() {
            public void mouseClicked(MouseEvent e) {
                System.out.println("Label clicked!");
            }

            public void mouseEntered(MouseEvent e) {
                System.out.println("Mouse entered the label!");
            }

            public void mouseExited(MouseEvent e) {
                System.out.println("Mouse exited the label!");
            }

            public void mousePressed(MouseEvent e) {
                // Not used in this example
            }

            public void mouseReleased(MouseEvent e) {
                // Not used in this example
            }
        });

        // Add the Label to the Frame
        frame.add(label);

        // Set Frame size and visibility
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

3. KeyListener:

- KeyListener is an interface used for handling keyboard events, such as key presses and key releases.
- It includes methods like keyPressed(KeyEvent e), keyReleased(KeyEvent e), and keyTyped(KeyEvent e).
- To handle keyboard events, you implement the KeyListener interface and register it with the component using addKeyListener().

```
import java.awt.*;
import java.awt.event.*;

public class KeyListenerExample {
    public static void main(String[] args) {
        // Create a Frame
        Frame frame = new Frame("KeyListener Example");

        // Create a TextField
        TextField textField = new TextField();

        // Add a KeyListener to the TextField
        textField.addKeyListener(new KeyListener() {
            public void keyTyped(KeyEvent e) {
                // Not used in this example
            }

            public void keyPressed(KeyEvent e) {
                System.out.println("Key pressed: " + e.getKeyChar());
            }

            public void keyReleased(KeyEvent e) {
                System.out.println("Key released: " + e.getKeyChar());
            }
        });

        // Add the TextField to the Frame
        frame.add(textField);

        // Set Frame size and visibility
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

4. WindowListener:

- WindowListener is an interface used for handling window events, such as window closing, opening, and iconifying.
- It includes methods like windowClosing(WindowEvent e), windowOpened(WindowEvent e), and windowIconified(WindowEvent e).
- To handle window events, you implement the WindowListener interface and register it with the window using addWindowListener().

```
import java.awt.*;
import java.awt.event.*;

public class WindowListenerExample {
    public static void main(String[] args) {
        // Create a Frame
        Frame frame = new Frame("WindowListener Example");

        // Create a WindowListener
        WindowListener windowListener = new WindowListener() {
            public void windowOpened(WindowEvent e) {
                System.out.println("Window opened");
            }

            public void windowClosing(WindowEvent e) {
                System.out.println("Window closing");
                System.exit(0);
            }

            public void windowClosed(WindowEvent e) {
                // Not used in this example
            }

            public void windowIconified(WindowEvent e) {
                System.out.println("Window iconified");
            }

            public void windowDeiconified(WindowEvent e) {
                System.out.println("Window deiconified");
            }

            public void windowActivated(WindowEvent e) {
                System.out.println("Window activated");
            }

            public void windowDeactivated(WindowEvent e) {
                System.out.println("Window deactivated");
            }
        };

        // Add the WindowListener to the Frame
        frame.addWindowListener(windowListener);

        // Set Frame size and visibility
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

5. Item Listener

The ItemListener interface in AWT is used for handling item events, typically generated by components like checkboxes, radio buttons, and combo boxes. It allows you to respond to changes in the selection state of these components

```
import java.awt.*;
import java.awt.event.*;

public class ItemListenerExample {
    public static void main(String[] args) {
        // Create a Frame
        Frame frame = new Frame("ItemListener Example");

        // Create a Checkbox
        Checkbox checkbox = new Checkbox("Check me");

        // Create a Label
        Label label = new Label("Checkbox state: Unchecked");

        // Create an ItemListener
        ItemListener itemListener = new ItemListener() {
            public void itemStateChanged(ItemEvent e) {
                if (e.getStateChange() == ItemEvent.SELECTED) {
                    label.setText("Checkbox state: Checked");
                } else {
                    label.setText("Checkbox state: Unchecked");
                }
            }
        };

        // Add the ItemListener to the Checkbox
        checkbox.addItemListener(itemListener);

        // Create a Panel
        Panel panel = new Panel();
        panel.setLayout(new FlowLayout());
        panel.add(checkbox);
        panel.add(label);

        // Add the Panel to the Frame
        frame.add(panel);

        // Set Frame size and visibility
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```