Introduction to Java

Introduction to Programming

The journey of programming is a fascinating evolution that spans decades of technological advancement. It all began in the mid-20th century, with the advent of the first computers. Early computing devices like the ENIAC and UNIVAC laid the foundation for digital computation, allowing complex mathematical calculations and data processing. However, these machines required intricate and time-consuming rewiring to perform different tasks, making them impractical for general-purpose use.

The breakthrough came with the development of assembly languages in the 1950s. Assembly languages provided a symbolic representation of machine code instructions, making programming slightly more human-readable and accessible. Yet, they still demanded a deep understanding of the underlying hardware, limiting their use to highly skilled individuals.

The 1950s also saw the birth of high-level programming languages. FORTRAN (Formula Translation) was among the first, designed for scientific and engineering applications. This marked a shift towards more user-friendly programming, enabling programmers to write code using familiar mathematical notations. COBOL (Common Business-Oriented Language) followed suit, catering to business data processing needs.

The 1960s witnessed the rise of languages like BASIC (Beginner's All-purpose Symbolic Instruction Code) and ALGOL (ALGOrithmic Language). These languages further abstracted programming, reducing the gap between code and machine instructions. However, it was the 1970s that truly marked a turning point with the creation of C, a powerful language that combined low-level control with high-level abstraction. C became instrumental in developing operating systems and applications, laying the foundation for modern computing.

The late 20th century and early 21st century brought forth a proliferation of high-level languages like Python, Java, and Ruby. These languages emphasised readability, ease of use, and portability, driving innovation across various domains. Additionally, object-oriented programming (OOP) gained traction, introducing a paradigm shift that facilitated modular and reusable code.

Today, we stand at the threshold of an era where languages like Python, JavaScript, and C++ dominate the programming landscape. The evolution of programming languages continues with a focus on concurrency, parallelism, and scalability to meet the demands of modern computing, including artificial intelligence, data science, and cloud computing.

In conclusion, the history of programming is a testament to human ingenuity, from the arduous days of machine code to the user-friendly high-level languages of today. Each phase of evolution has brought us closer to computers that can understand and execute human intentions, making technology accessible and transformative across the globe.

Evolution of Java

The evolution of Java is a remarkable journey that showcases the language's adaptability and enduring relevance in the ever-changing world of programming. Developed by James Gosling and his team at Sun Microsystems (later acquired by Oracle Corporation), Java was released in 1995 and quickly gained traction due to its unique features and cross-platform capabilities.

Java's early days saw it positioned as a language for building applets, which were small programs designed to run within web browsers. This allowed for interactive content on the emerging World Wide Web. However, Java's true breakthrough came with its "Write Once, Run Anywhere" mantra. By introducing the concept of platform independence through the Java Virtual Machine (JVM), Java enabled programs to be compiled into bytecode that could be executed on any system with a compatible JVM. This marked a significant departure from languages tied to specific operating systems.

The late 1990s and early 2000s witnessed Java's expansion into various domains. The introduction of Java 2 (later renamed Java SE) brought important features like Swing for building graphical user interfaces and the Collections Framework for efficient data manipulation. Additionally, the enterprise space was revolutionised with the inception of Java EE (Enterprise Edition), which provided tools for creating large-scale, distributed applications.

As the new millennium unfolded, Java's community-driven approach to development led to the creation of open-source implementations like OpenJDK. This not only made Java more accessible but also allowed for collaborative improvements to the language's core.

The 2010s brought Java into the realm of modern programming challenges. Java's focus on performance, security, and portability made it a solid choice for building Android applications, powering millions of devices worldwide. Furthermore, the language underwent rapid updates with the introduction of features like lambdas, the Stream API, and modules in Java 8, enhancing its expressive power and making code more concise.

Java's evolution continued with the release of Java 9, which brought modularity to the language through the Java Platform Module System (JPMS). Subsequent versions further refined and expanded upon this foundation, addressing issues related to scalability and performance in a multicore and cloud-driven landscape.

In recent years, Java's relevance has extended to emerging technologies like cloud computing, microservices architecture, and serverless computing. The language remains a stalwart choice for enterprise applications, financial systems, and large-scale software development due to its reliability and ecosystem.

In summary, Java's evolution from a language for applets to a versatile, enterprise-grade programming language is a testament to its adaptability, community support, and continuous improvement. With a strong foundation and ongoing innovation, Java continues to shape the present and future of programming.

Compilation in Action

Compilation in Java is a crucial process where the human-readable source code is transformed into bytecode that can be executed by the Java Virtual Machine (JVM). Let's explore this process using a simple example.

Consider a simple Java program that prints "Hello, World!" to the console. Here's the source code:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Here's a breakdown of the compilation process:

1. **Writing the Source Code**: You start by writing your Java code in a plain text file with a .java extension. In this case, the file would be named HelloWorld.java.

2. **Compiling the Source Code**: Open a terminal and navigate to the directory containing HelloWorld.java. Use the javac command to compile the code:

   ```
   javac HelloWorld.java
   ```

If there are no syntax errors in your code, the Java compiler (`javac`) generates a bytecode file named `HelloWorld.class`.

3. **Bytecode Generation**: The javac compiler translates your Java source code into bytecode, which is a low-level representation of your code that the JVM can understand and execute. This bytecode is stored in the HelloWorld.class file.

4. **Execution by JVM**: To run the compiled bytecode, use the *java* command followed by the name of the class containing the main method (entry point for execution):

   ```
   java HelloWorld
   ```

The JVM loads the HelloWorld class, finds the main method, and executes the code within it. You'll see the output: "Hello, World!".

This simple example demonstrates the compilation process. The javac compiler transforms human-readable Java source code into bytecode that's independent of the underlying platform. The JVM then interprets and executes this bytecode, producing the desired output.

Remember that this is a basic overview. In real-world scenarios, Java programs can be composed of multiple classes, utilise external libraries, and involve more complex compilation and execution processes. Nonetheless, this example captures the essence of how compilation works in Java.

In the world of Java development, understanding the distinctions between the Java Runtime Environment (JRE) and the Java Development Kit (JDK) is essential. These two components play distinct roles in the Java ecosystem. Let's delve into their definitions and purposes using simple explanations.

Java Runtime Environment (JRE):

The Java Runtime Environment (JRE) is a bundle of software tools and libraries that enables you to run Java applications on your computer. It provides the necessary runtime environment to execute Java programs without the need for any development-related tools. In other words, the JRE is what you need if you only want to run Java applications on your system.

When you install the JRE, you get the Java Virtual Machine (JVM) and the Java Standard Library. The JVM is responsible for interpreting and executing Java bytecode, which is the compiled form of Java source code. The Java Standard Library provides a rich collection of pre-built classes and methods that Java applications can use to perform various tasks.

Java Development Kit (JDK):

The Java Development Kit (JDK) is a comprehensive package that includes everything provided by the JRE, along with additional tools specifically designed for developing Java applications. If you're a developer, the JDK is what you need to create, compile, and test Java programs.

The JDK includes the following:

1. **JRE Components**: The JDK includes the JRE, so you can run Java applications just like with the standalone JRE installation.

2. **Compiler (javac)**: The JDK comes with the Java compiler (javac), which translates human-readable Java source code into bytecode.

3. **Development Tools**: The JDK provides various development tools such as the Java debugger (jdb), profiler, and documentation generation tools.

4. **Additional Libraries and Utilities**: The JDK includes tools for packaging and deploying applications, as well as libraries for various purposes (e.g., JavaFX for building graphical user interfaces).

To summarise, the JRE is for running Java applications, while the JDK is for developing and compiling Java applications. If you're just an end-user who wants to run Java programs, you need the JRE. If you're a developer creating Java applications, you need the more comprehensive JDK. It's worth noting that the JDK contains the JRE as part of its package, so when you install the JDK, you effectively get both development and runtime capabilities.

Anatomy of a Java Program

Here's a simple Java program that demonstrates the basic components of a Java program:

```
    // This is a simple Java program that prints a message to the console.

    // Class declaration
    public class HelloWorld {
        // Main method, the entry point of the program
        public static void main(String[] args) {
            // Statement to print a message
            System.out.println("Hello, World!");
        }
    }
```

Now, let's break down the different components of this Java program:

1. **Comments**: The program begins with comments explaining its purpose. Comments are not executed and provide context to the code.

2. **Class Declaration**: The keyword *public* denotes that the class is accessible from other classes. The class is named *HelloWorld*, and it serves as a blueprint for objects.

3. **Main Method**: The main method is the starting point of the program. It's where execution begins. It's declared as a *public static void main(String[] args)*. Here, public makes the method accessible, static means it belongs to the class itself (not an instance), void indicates that it doesn't return a value, and String[] args is an array of command-line arguments.

4. **Print Statement**: Inside the main method, there's a statement that uses the System.out.println() method to print "Hello, World!" to the console.

To run this program:

1. Save it in a file named HelloWorld.java.
2. Open a terminal and navigate to the directory containing the file.
3. Compile the program using the command javac HelloWorld.java.
4. Run the compiled program with java HelloWorld.

Executing the program will result in the output: "Hello, World!"

This example showcases the core components of a Java program: class declaration, main method, statements, and comments. Understanding these elements provides a foundation for more complex Java programming.

Printing multiple messages & Comments

In Java, you can print multiple statements and use comments to enhance the clarity and readability of your code. Let's explore how to achieve this with practical examples.

To print multiple statements in Java, you can simply add multiple System.out.println() statements or other print-related methods. Each call to System.out.println() prints a line of text followed by a line break. Here's an example:

```
public class PrintMultipleStatements {
    public static void main(String[] args) {
        System.out.println("Statement 1");
        System.out.println("Statement 2");
        System.out.println("Statement 3");
    }
}
```

When you run this program, it will output:

```
Statement 1
Statement 2
Statement 3
```

Using Comments:

Comments are essential for documenting your code and explaining its functionality. Java supports two types of comments:

1. **Single-Line Comments**: These are used to comment out a single line of code or provide explanations. They start with `//`.

```
// This is a single-line comment
int x = 10; // This comment explains the purpose of the
variable
```

2. **Multi-Line Comments**: These are used for longer explanations that span multiple lines. They start with `/*` and end with `*/`.

```
/*
This is a multi-line comment.
It can span multiple lines.
Useful for detailed explanations.
*/
```

Comments are not executed by the compiler, so they won't affect the behaviour of your program. However, they are crucial for making your code understandable to others (and to your future self).

Combining multiple statements with comments:

```
public class ExampleWithComments {
    public static void main(String[] args) {
        // Printing a message
        System.out.println("Hello!");

        // Performing a calculation
        int x = 5 + 3;
        System.out.println("The result is: " + x);

        /*
        This is a multi-line comment.
        It explains the purpose of the code block below.
        */
        System.out.println("End of the program.");
    }
}
```

In this example, comments are used to explain the purpose of each statement and provide context for the code. This practice helps make your code more understandable and maintainable.

Understanding Errors in Java

Errors in Java can be classified into three main categories: syntax errors, runtime errors, and logic errors. Understanding these types of errors is crucial for effective debugging and producing functional Java programs.

1. **Syntax Errors**: These are also known as compile-time errors. They occur when your code violates the rules of the Java language syntax. Examples include missing semicolons at the end of statements, mismatched parentheses, or using undefined variables. Such errors prevent the code from compiling and need to be fixed before the program can be executed.

2. **Runtime Errors**: These errors, also called exceptions, occur during the execution of the program. They arise when the program encounters unexpected conditions that it cannot handle. Examples include division by zero, trying to access an element that is out of bounds in an array, or attempting to read input from a closed file. Runtime errors can be identified and handled using exception handling mechanisms like try, catch, and finally.

3. **Logic Errors:** Logic errors are the most subtle and difficult to detect. They occur when the program compiles and runs without error messages, but the output is not what you expect. These errors stem from mistakes in the program's algorithm or logic. Debugging logic errors often requires careful code inspection and testing.

In summary, Java programming involves dealing with syntax errors, runtime errors (exceptions), and logic errors. Recognizing these error types and adopting effective debugging practices, like using development environments or debugging tools, helps programmers identify and resolve issues efficiently, leading to robust and functional Java applications.

Fundamentals of Java

What is a Variable?

- In Java, a variable is a name given to a memory location. It is the basic unit of storage in a program. The value stored in a variable can be changed during program execution. Variables in Java are only a name given to a memory location. All the operations done on the variable affect that memory location. Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in *fields* (variables in some programming languages) and exposes its behavior through *methods* (functions in some programming languages).

Declaring Variable

- To declare a variable, we specify its type followed by a name. For example, int age; declares an integer variable called age.

Assigning a Value

- After declaring a variable, we can assign a value to it using the assignment operator “=”. For example, age = 25; assigns the value 25 to the age variable.

- We can also declare and assign a value in a single line, like int score = 95;

Printing a Variable

- We can display the value of a variable using the System.out.println() method.

- For example, System.out.println("Age: " + age); will print the value of the age variable along with the text "Age: ".

Example:

```java
public class Example {
    public static void main(String[] args) {
        int age;        // Declare an integer variable
        age = 25;       // Assign a value to it
        System.out.println("Age: " + age);  // Print the value
    }
}
```

In this simple example, we declared an integer variable called age, assigned the value 25 to it, and then printed its value, resulting in the output "**Age: 25**".

Variables (Literals) Naming Conventions

Every programming language has its own set of rules and conventions for the kinds of names that you're allowed to use, and the Java programming language is no different. The rules and conventions for naming your variables can be summarised as follows:

- Variable names are case-sensitive. A variable's name can be any legal identifier — an unlimited-length sequence of Unicode letters and digits, beginning with a letter, the dollar sign **"$"**, or the underscore character **"_"**.

- The convention, however, is to always begin your variable names with a letter, not **"$"** or **"_"** or digits.

- Additionally, the dollar sign character, by convention, is never used at all. In some situations, auto-generated names will contain the dollar sign, but your variable names should always avoid using it.

- A similar convention exists for the underscore character; while it's technically legal to begin your variable's name with **"_"**, this practice is discouraged. White space is not permitted.

- Subsequent characters may be letters, digits, dollar signs, or underscore characters. Conventions (and common sense) apply to this rule as well. When choosing a name for your variables, use full words instead of cryptic abbreviations. Doing so will make your code easier to read and understand. In many cases it will also make your code self-documenting; fields named `cadence`, `speed`, and `gear`, for example, are much more intuitive than abbreviated versions, such as `s`**,** `c`**,** and `g`. Also, keep in mind that the name you choose must not be a keyword or reserved word.

- If the name you choose consists of only one word, spell that word in all lowercase letters. If it consists of more than one word, capitalize the first letter of each subsequent word (also known as **Camel Case**). The names `gearRatio` and `currentGear` are prime examples of this convention. If your variable stores a constant value, such as `static final int NUM_GEARS = 6`, the convention changes slightly, capitalizing every letter and separating subsequent words with the underscore character. By convention, the underscore character is never used elsewhere.

Data Types

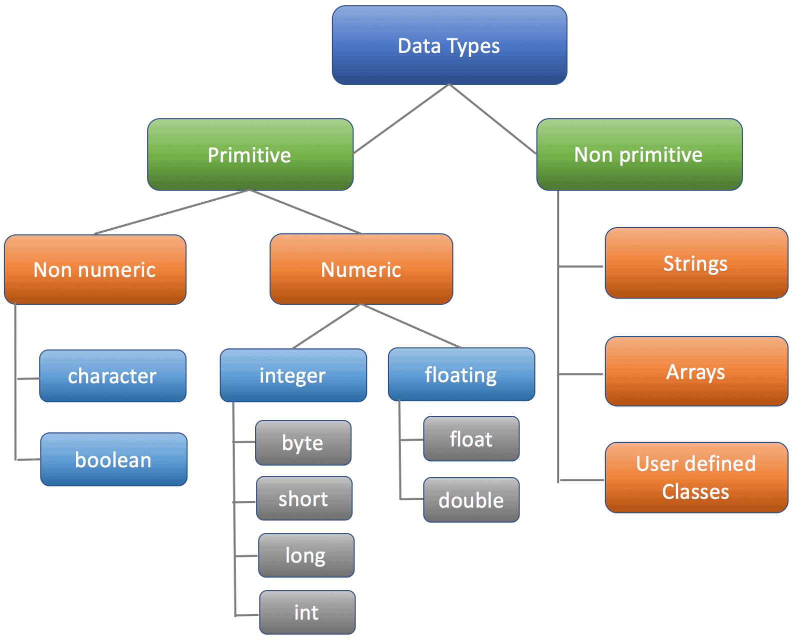
- **Declaration:** The Java programming language is statically typed, which means that all variables must first be declared before they can be used. This involves stating the variable's type and name, as you've already seen:

  ```
  int gear = 1;
  ```

  Doing so tells your program that a field named **"gear"** exists, holds numerical data, and has an initial value of **"1"**. A variable's data type determines the values it may contain, plus the operations that may be performed on it. In addition to `int`, the Java programming language supports seven other *primitive data types*.

  A primitive type is predefined by the language and is named by a reserved keyword.

  Primitive values do not share a state with other primitive values.



  The eight primitive data types supported by the Java programming language are:

- **byte**: The `byte` data type is an 8-bit signed two's complement integer. It has a minimum value of **-128** and a maximum value of **127** (inclusive). The `byte` data type can be useful for saving memory in large arrays, where memory savings actually matter. They can also be used in place of `int` where their limits help to clarify your code.

- **short**: The `short` data type is a 16-bit signed two's complement integer. It has a minimum value of **-32,768** and a maximum value of **32,767** (inclusive). As with `byte`, the same guidelines apply: you can use a `short` to save memory in large arrays, in situations where the memory savings actually matters.

- **int**: By default, the `int` data type is a **32-bit** signed two's complement integer, which has a minimum value of **-2^{31}** and a maximum value of **$2^{31}-1$**.
  - In Java SE 8 and later, you can use the `int` data type to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of **$2^{32}-1$**. Use the Integer class to use `int` data type as an unsigned integer.

- **long**: The `long` data type is a 64-bit two's complement integer. The signed long has a minimum value of **-2^{63}** and a maximum value of **$2^{63}-1$**. In Java SE 8 and later, you can use the `long` data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of **$2^{64}-1$**. Use this data type when you need a range of values wider than those provided by `int`.

- **float**: The `float` data type is a single-precision **32-bit** floating point. As with the recommendations for `byte` and `short`, use a `float` (instead of `double`) if you need to save memory in large arrays of floating point numbers.

- **double**: The `double` data type is a double-precision **64-bit** floating point.

- **boolean**: The `boolean` data type has only two possible values: `true` and `false`. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.

- **char**: The `char` data type is a single **16-bit Unicode** character. It has a minimum value of `'\u0000'` (or 0) and a maximum value of `'\uffff'` (or 65,535 inclusive).

- In addition to the eight primitive data types listed above, the Java programming language also provides special support for character strings via the java.lang.String class.

- Enclosing your character string within double quotes will automatically create a new String object; for example, String s = **"this is a string";** .

- String objects are immutable, which means that once created, their values cannot be changed. The String class is not technically a primitive data type.

- For the primitive data types, mention the associated wrapper classes too (Byte, Short, Integer, Long, Float, Double, Boolean, Character, etc.).

Literal Types

You may have noticed that the new keyword isn't used when initializing a variable of a primitive type. Primitive types are special data types built into the language; they are not objects created from a class. A *literal* is the source code representation of a fixed value; literals are represented directly in your code without requiring computation.

As shown below, it's possible to assign a literal to a variable of a primitive type:

```
boolean result = true;
char capitalC = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
```

Integer Literals:

- An integer literal is of type long if it ends with the letter `L` or `l`; otherwise, it is of type `int`. It is recommended that you use the uppercase letter L because the lowercase letter l is hard to distinguish from the digit 1.

- Values of the integral types byte, short, int, and long can be created from int literals. Values of type long that exceed the range of int can be created from long literals. Integer literals can be expressed by these number systems:

    - **Decimal**: Base 10, whose digits consist of the numbers 0 through 9; this is the number system you use every day
    - **Hexadecimal**: Base 16, whose digits consist of the numbers 0 through 9 and the letters A through F
    - **Binary**: Base 2, whose digits consists of the numbers 0 and 1 (you can create binary literals in Java SE 7 and later)

- For general-purpose programming, the decimal system is likely to be the only number system you'll ever use. However, if you need to use another number system, the following example shows the correct syntax. The prefix 0x indicates hexadecimal and 0b indicate binary:

```
// The number 26, in decimal
int decVal = 26;
//  The number 26, in hexadecimal
int hexVal = 0x1a;
// The number 26, in binary
int binVal = 0b11010;
```

Floating Point Literals

- A floating-point literal is of type `float` if it ends with the letter `F` or `f`; otherwise its type is `double` and it can optionally end with the letter `D` or `d`.

- The floating point types (`float` and `double`) can also be expressed using E or e (for scientific notation), F or f (32-bit float literal), and D or d (64-bit double literal; this is the default and by convention is omitted).

```
double d1 = 123.4;
// same value as d1, but in scientific notation
double d2 = 1.234e2;
float f1  = 123.4f;
```

Characters and String Literals

- Literals of types `char` and `String` may contain any Unicode (UTF-16) characters. If your editor and file system allow it, you can use such characters directly in your code. If not, you can use a "Unicode escape" such as `'\u0108'` (capital C with circumflex), or `"S\u00ED Se\u00F1or"` (Sí Señor in Spanish).

- Always use 'single quotes' for `char` literals and "double quotes" for `String` literals. Unicode escape sequences may be used elsewhere in a program (such as in field names, for example), not just in `char` or `String` literals.

- The Java programming language also supports a few special escape sequences for `char` and `String` literals: `\b` (backspace), `\t` (tab), `\n` (line feed), `\f` (form feed), `\r` (carriage return), `\"` (double quote), `\'` (single quote), and `\\` (backslash).

- There's also a special `null` literal that can be used as a value for any reference type. `null` may be assigned to any variable, except variables of primitive types. There's little you can do with a `null` value beyond testing for its presence. Therefore, `null` is often used in programs as a marker to indicate that some object is unavailable.

- Finally, there's also a special kind of literal called a *class literal*, formed by taking a type name and appending "`.class`"; for example, `String.class`. This refers to the object (of type `Class`) that represents the type itself.

- ASCII Table (Unicode 15.0 Character Code Charts).

Using Underscore Characters in Numeric Literals

In Java SE 7 and later, any number of underscore characters (_) can appear anywhere between digits in a numerical literal. This feature enables you, for example. to separate groups of digits in numeric literals, which can improve the readability of your code. For instance, if your code contains numbers with many digits, you can use an underscore character to separate digits into groups of three, similar to how you would use a punctuation mark like a comma, or a space, as a separator.

The following example shows other ways you can use the underscore in numeric literals:

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi =  3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

You can place underscores only between digits; you cannot place underscores in the following places:

- At the beginning or end of a number.
- Adjacent to a decimal point in a floating point literal.
- Prior to an `F` or `L` suffix.
- In positions where a string of digits is expected.

The following examples demonstrate valid and invalid underscore placements (which are highlighted) in numeric literals:

```
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi1 = 3_.1415F;

// Invalid: cannot put underscores
// adjacent to a decimal point
float pi2 = 3._1415F;

// Invalid: cannot put underscores
// prior to an L suffix
```

```java
long socialSecurityNumber1 = 999_99_9999_L;

// OK (decimal literal)
int x1 = 5_2;

// Invalid: cannot put underscores
// At the end of a literal
int x2 = 52_;

// OK (decimal literal)
int x3 = 5_______2;

// Invalid: cannot put underscores
// in the 0x radix prefix
int x4 = 0_x52;

// Invalid: cannot put underscores
// at the beginning of a number
int x5 = 0x_52;

// OK (hexadecimal literal)
int x6 = 0x5_2;

// Invalid: cannot put underscores
// at the end of a number
int x7 = 0x52_;
```

String Literals

Strings, which are widely used in Java programming, is a sequence of characters. In the Java programming language, strings are objects. The Java platform provides the `String` class to create and manipulate strings.

Creating Strings

The most direct way to create a string is to write:

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

In this case, "Hello world!" is a *string literal*—a series of characters in your code that is enclosed in double quotes. Whenever it encounters a string literal in your code, the compiler creates a `String` object with its value—in this case, `Hello world!`.

As with any other object, you can create `String` objects by using the `new` keyword and a constructor. The `String` class has thirteen constructors that allow you to provide the initial value of the string using different sources, such as an array of characters:

```
char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };
String helloString = new String(helloArray);
System.out.println(helloString);
```

The last line of this code snippet displays `hello`.

Java Wrapper Classes

In Java, data can be categorized into two types: primitive and non-primitive (objects). While primitive data types like int, float, and boolean are efficient for simple data storage, they lack the flexibility of objects. This is where Java wrapper classes come to the rescue. Wrapper classes provide a way to represent primitive data types as objects, enabling us to use them in object-oriented contexts and collections. In this beginner-friendly tutorial, we'll explore Java wrapper classes and how to use them effectively.

What Are Wrapper Classes?

Java wrapper classes are a set of classes that wrap primitive data types into objects. There's a wrapper class for each of the eight primitive data types:

1. Byte: Wraps byte.
2. Short: Wraps short.
3. Integer: Wraps int.
4. Long: Wraps long.
5. Float: Wraps float.
6. Double: Wraps double.
7. Character: Wraps char.
8. Boolean: Wraps boolean.

Why Use Wrapper Classes?

1. Compatibility: Many Java libraries and APIs require objects. Wrapper classes make it possible to use primitive data types where objects are needed.

2. Null Values: Wrapper classes allow you to assign null values to variables, which isn't possible with primitives.

3. Utility Methods: They provide useful methods for conversions, comparisons, and other operations.

Creating Wrapper Objects

You can create wrapper objects using constructors or static factory methods. For example:

```
Integer intObj = new Integer(42); // Using constructor
Double doubleObj = Double.valueOf(3.14); // Using valueOf()
method
Boolean boolObj = Boolean.TRUE; // Using a constant
```

Auto-Boxing and Auto-Unboxing

Java simplifies the process of converting between primitive types and wrapper objects. With auto-boxing, you can assign primitive values to wrapper objects without explicit conversion, and auto-unboxing lets you extract primitive values from wrappers. For instance:

```
Integer num = 10; // Auto-boxing (primitive to wrapper)
int value = num; // Auto-unboxing (wrapper to primitive)
```

Common Operations

Wrapper classes provide methods for common tasks:

- `toString(): Convert to a string.`
- `parseInt(): Parse a string to an int.`
- `equals(): Compare two wrapper objects.`
- `compareTo(): Compare two wrapper objects.`

In conclusion, Java wrapper classes bridge the gap between primitive and object-oriented programming. They offer flexibility and compatibility while simplifying data manipulation and usage in various Java contexts. Learning to use wrapper classes effectively will enhance your Java programming skills.

Operators in Java

Introduction

Operators are specific symbols in Java that are the basic building blocks of any language used to perform operations on variables and values. Java provides different operators that can be categorised into other groups based on functionality. Here are some operators discussed below.

1. Arithmetic Operators

Arithmetic operators in Java perform various mathematical operations on numeric values and variables representing such values as integers, float-point numbers, etc.

Types:

- **'+' Addition**: adds two numbers and concatenates two string literals.

```
  int x = 4+5; //assigns value 9 to x.

  String a= "hello";
  String b ="world";
  String sentence =a+b   //Assigns value "helloworld" to sentence

Note - '+' operator works in two ways while using numbers it performs
addition while with Strings it performs concatenation.
```

- **'-' Subtraction**: subtraction of numbers.

```
 int z = 10/2; //assigns value 5 to z.
```

- **'*' Multiplication**: Multiplication of numbers.

```
int z = 3 * 4;     //assigns value 12 to z
```

- **'/' Division**: gives quotient after the division of numbers.

```
 int z = 10/2; //assigns value 5 to z.
```

- **%' Modulo**: gives remainder after division of numbers.

```
System.out.println(10/3);   //prints value 1.
```

2. Assignment Operators

As the name suggests, assignment operators assign values to variables that are fundamental to programming. The right side operand of the assignment operator is a value, and the left side operand is a variable. Assignment operators follow right-side associativity, i.e., expressions are evaluated from right to left.

Key points to remember:

- Datatypes of value and variable should be the same.
- The value must be declared before using it or should be a constant.
- **'=' operator**: assigns the value on the right to the variable on the left.
  int x = 5;   //assigns value 5 to the variable x.
  int y = x;   //assigns value of x (i.e. 5) to the variable y.
- **Augmented Operators:** Expressions involving the same variable on both sides of the operand to manipulate the variable's value are augmented operators. For example,

```
int x = 4;
int x = x + 5;  //value of x will change from  4 to 10.

//the above code can also be written as:
x += 5;   //addition assignment operator.

The same can be done with other arithmetic operators.

x -= 2; // x = 2
x *= 4; // x = 16
x /= 2; // x = 2
```

3. Precedence:

Precedence determines the order in which operators are evaluated in an expression. When an expression contains multiple operators, Java follows a specific set of rules to determine which operation should first be performed. Below is the table with decreasing order of precedence, i.e. the operator above the other operators is given a higher preference than the one below.

Operators	Precedence
Postfix	expr++ expr–
Unary	++expr –expr +expr -expr ~!
Multiplicative	* / %

Additive	+ -
Shift	<< >> >>>
Relational	< > <= >= instanceOf
Equality	== !=
Bitwise And	&
Bitwise Exclusive OR	^
Bitwise Inclusive OR	|
Logical AND	&&
Logical OR	||
Ternary	? :
Assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Will learn about the other operators ahead.

```
System.out.println(2+5*4)  //prints 22, as multiplication is executed
first.
```

4. Implicit Conversions:

Implicit conversion, also known as type promotion, is the automatic conversion of one data type to another by the Java compiler without writing any extra instructing code. We'll now see some string operations depicting conversion.

- Implicit conversion with strings occurs when we combine strings with other data types, like numbers.
- Java can automatically convert these non-string data types into strings, so they play nicely together.

```
System.out.println(2+5*4)  //prints 22, as multiplication is executed
first.
```

Here, the Java compiler identified that we were mixing the variables of different data types; therefore, it converted int to string implicitly, concatenated with the string literals and stored it in 'sentence'.

```java
int n1 = 10;
double n2 = 3.5;

Double result = n1 + n2;

//int n1 implicitly got promoted to double 10.0 and added to n2.
```

Note - Implicit conversion is only allowed while converting shorter datatypes to larger data types as it is possible to accommodate smaller data types to larger ones, which is also known as widening.

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

       long a = 2.5;
       int b = 4;
       int c = a+b;

//This code will throw a compilation error.

    }
}
```

5. Type Casting:

Typecasting refers to converting one data type to another. Two common types of typecasting are implicit and explicit casting.

- **Implicit:** Implicit type casting refers to converting a small data type into a larger one by Java to prevent data loss. It is also known as widening or automatic type conversion.

```java
int a= 5;
double b= smaller; // Implicit casting from int to double
```

- **Explicit:** Explicit type casting refers to converting a large data type into a small one. Explicit type casting may result in data loss and requires the programmer to specify the type conversion explicitly. It is also known as narrowing or manual type conversion.

```
double a= 10.5;
int b= (int) larger; // Explicit casting from double to int
```

Key points to remember:

- Use implicit casting when converting to a larger data type to prevent data loss or perform operations involving mixed data types.
- Use explicit casting when converting to a smaller data type, but be cautious about potential data loss. Explicit casting allows you to control the conversion.
- Explicit casting requires manual intervention and may result in data loss.
- Choose the appropriate casting method based on your data type conversion needs.

6. Unary Operators:

In Java, unary operators are those operators that perform operations on a single operand used to manipulate values in different ways.

- **Unary Plus (+) Operator:** This operator doesn't change any value; it just states that the number is positive.

  int x = +5;

- **Unary Minus (-) Operator**: The minus operator negates the value. It turns positive numbers into negative ones and vice versa.

  int x = -5;

- **Increment Operator (++)**: The operator increases the value of a variable by 1. It can be used in two ways: pre-increment and post-increment.

```
int a = 10;
System.out.println(a++);

//prints initial value 10 and then increments value by 1.
```

```
int a = 10;
System.out.println(++a);

//increments the value of 'a' by 1 first and then prints 'a'  i.e. 11.
```

- **Decrement Operator (--)**: The decrement operator decrements the value by 1. Just like the increment operator, it can also be used in two ways, i.e. pre-decrement and post-decrement

```
int a = 10;
System.out.println(a--);

//prints initial value 10 and then decrements value by 1.
```

```
int a = 10;
System.out.println(--a);

//decrements the value of 'a' by 1 first and then prints 'a'  i.e. 9.
```

- **Logical Not Operator (!)**: The logical not operator reverses the value of a boolean expression. True becomes false and vice-versa.

```
boolean flag = true;
            System.out.print(!flag);

      //prints false
```

7. Bitwise Operators:

Bitwise operators are used in Java to manipulate individual data bits in integers or longs. These operators allow you to perform low-level operations on binary representations of data.

- **The Bitwise AND operator ('&'):** The bitwise AND operator (&) performs a bitwise AND operation between corresponding bits of two integers. It results in a 1 only if both bits being compared are 1. For example,

```
int a = 5;    // Binary: 0101
int b = 3;    // Binary: 0011
int result = a & b;  // Result: 0001 (Decimal 1)
```

- **Bitwise OR operator ('|'):** The bitwise OR operator (|) performs a bitwise OR operation between corresponding bits of two integers. It results in a 1 if at least one of the compared bits is 1. For example,

```
int a = 5;    // Binary: 0101
int b = 3;    // Binary: 0011
int result = a | b;  // Result: 0111 (Decimal 7)
```

- **Left shift operator (‘<<’):** The left shift operator (<<) shifts the bits of an integer to the left by a specified number of positions. It effectively multiplies the number by 2, raising the shift amount's power.

```
int a = 5;     // Binary: 0101
int result = a << 2;  // Result: 20
```

- **Right shift operator (‘<<’):** The right shift operator (>>) shifts the bits of an integer to the right by a specified number of positions. It effectively divides the number by 2, raised to the power of the shift amount.

```
int a = 16;   // Binary: 10000
int result = a >> 2;  // Result: 4
```

8. Relational Operators:

Relational operators in Java are used to compare two values and determine their relationship. These operators return a boolean result, indicating whether the specified condition is true or false.

- **Equality Operator (‘==’):** The equality operator compares two values for equality by checking if the values on both sides are equal and returns ‘true’ if the values are equal and ‘false’ otherwise.

```
int x = 5;
int y = 5;
boolean isEqual = (x == y); // isEqual = true
```

- **Inequality Operator(!=):** The inequality operator (!=) checks if two values are not equal and returns true if the values are different and false if they are equal.

```
int a = 10;
int b = 5;
boolean isNotEqual = (a != b); // isNotEqual = true
```

.

- **Greater Than Operator (>):** The greater than operator (>) checks if the value on the left is greater than the value on the right. Returns true if the condition is met; otherwise, it returns false.

```
int num1 = 8;
int num2 = 5;
boolean isGreaterThan = (num1 > num2); // isGreaterThan = true
```

- **Less Than Operator (‘<’):** The less than operator checks if the value on the left is less than the value on the right and returns true if the condition is true; otherwise, it returns false.

```
int score1 = 75;
int score2 = 90;
boolean isLessThan = (score1 < score2); // isLessThan = true
```

- **Greater Than or Equal To Operator (>=):** The greater than or equal to operator checks if the value on the left is greater than or equal to the value on the right and returns true if the condition is true. Otherwise, it returns false.

```
int age1 = 20;
int age2 = 20;
boolean isGreaterOrEqual = (age1 >= age2); // isGreaterOrEqual = true
```

- **Less Than or Equal To Operator (<=):** The less than or equal operator checks if the value on the left is less than or equal to the right and returns true if the condition is true; otherwise, it returns false.

```
double price1 = 45.0;
double price2 = 50.0;
boolean isLessOrEqual = (price1 <= price2); // isLessOrEqual = true
```

9. Logical Operators:

Logical operators in Java are used to perform logical operations on boolean values or conditions. They help make decisions and control the flow of your program based on the truth values of expressions.

1. Logical AND (&&) Operator:

- The logical AND operator combines two boolean expressions and returns true if both expressions on its left and right sides are valid.
- If any of the expressions is false, the result will be false.

```
boolean isSunny = true;
boolean isWarm = true;
boolean isGoodWeather = isSunny && isWarm; // isGoodWeather = true
```

2. Logical OR Operator(||):

- The logical OR operator combines two boolean expressions and returns true if at least one of the expressions on its left and right sides is true.
- If both expressions are false, the result will be false.

```
boolean hasCoffee = true;
boolean hasMilk = false;
boolean canMakeCoffee = hasCoffee || hasMilk; // canMakeCoffee = true
```

3. Combining Logical Operators: We can combine logical operators to create complex conditions for decision-making in our Java programs.

- Example for using Logical AND and Logical OR:

```
int age = 25;
boolean isStudent = true;
boolean isTeenagerOrStudent = (age < 20) || isStudent; //
isTeenagerOrStudent = true
```

Conclusion

In conclusion, Java provides a variety of operators for performing different operations within Java programs. Here's a summary of the operators we studied:

- Arithmetic Operators: Used for basic mathematical calculations such as addition, subtraction, multiplication, division, and modulus.
- Assignment Operators: Used to assign values to variables and concisely perform operations.
- Relational Operators: Used to compare values and determine their relationship, including equality, inequality, greater than, less than, and combinations.
- Logical Operators: Employed to evaluate logical conditions, combining boolean values using AND (&&) and OR (||) operators and negating conditions with NOT (!).
- Bitwise Operators: Used for low-level bit manipulation operations on integers.
- Conditional Operator (Ternary): Allows conditional value assignment based on a given condition.

These operators are fundamental to Java programming and are essential for performing calculations, making decisions, and controlling the flow of your code. Mastery of these operators is crucial for developing efficient and functional Java applications.

Control Flow- Conditionals

Introduction

Conditionals are fundamental constructs in Java that allow you to make decisions and control the flow of your program based on specified conditions. They enable you to execute different code blocks depending on whether certain conditions are met.

1. `if` statement

The if statement is used when you want to execute a block of code only if a specific condition evaluates to true. If the condition is false, the code inside the block is skipped.

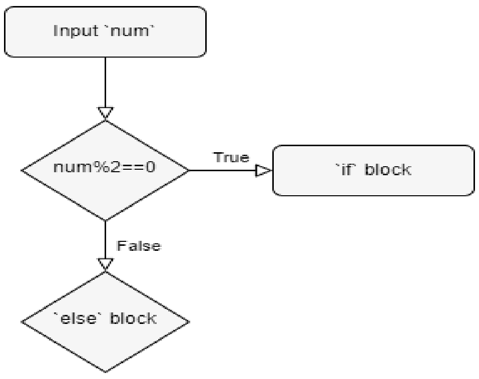
```
if (condition) {
    // Code to execute if the condition is true
}
```

2. `if-else` statement

The if-else statement allows you to provide an alternative code block executed when the condition is false. It provides a way to handle both cases.

```
 if (num%2==0) {
    System.out.println("Even Number");
}else {
    System.out.println("Odd Number");}
```

The above code can be visualised as below:

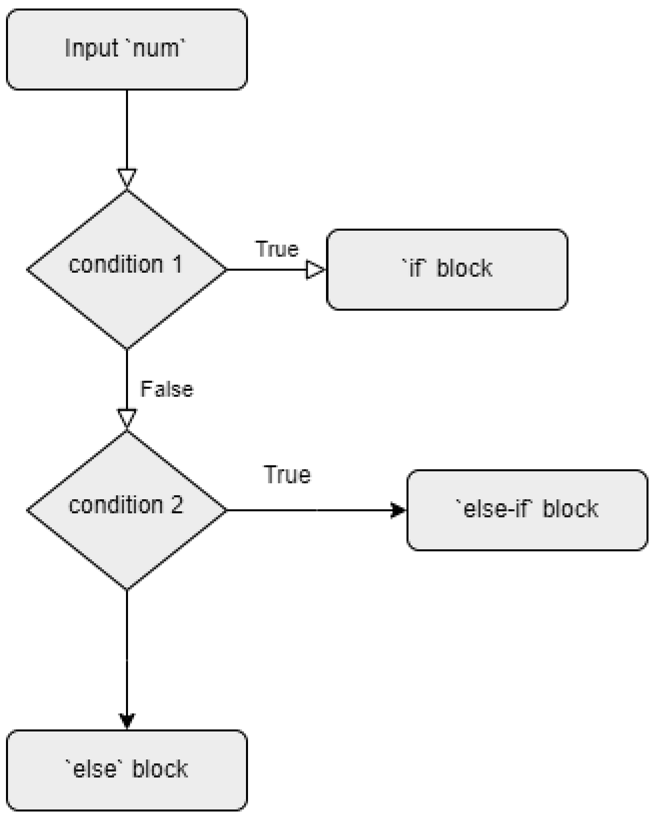


3. `if-else-if` statement

The *if-else-if statement* is used when you have multiple conditions to check sequentially. It allows you to specify different code blocks for each condition.

```
if (condition1) {
    // Code to execute if condition1 is true
} else if (condition2) {
    // Code to execute if condition2 is true
} else {
    // Code to execute if none of the conditions are true
}
```

The above code can be visualised as:



Key points to remember:

- Use meaningful variable and method names to enhance code readability.
- Always enclose code blocks within curly braces `{}` even for single statements to avoid ambiguity.
- Test your conditionals with various inputs to ensure they behave as expected.

4. `switch` statement

The `switch` statement is another conditional in Java that allows you to perform multi-way branching based on the value of an expression. It's designed for scenarios where you want to execute different code blocks depending on the value of a variable or expression.

```
switch (expression) {
    case value1:
        // Code to execute if expression equals value1
        break;
    case value2:
        // Code to execute if expression equals value2
        break;
    // Additional cases can follow...
    default:
        // Code to execute if expression doesn't match any case
}
```

Keywords in the code:

- **case**: Labels indicating the possible values of the expression.
- **break**: Terminates the switch block. Without it, execution continues to the next case.
- **default**: The default case is optional but serves as a catch-all if no other cases match the expression. It's commonly used to handle unexpected or undefined values.

Working:

- The expression is evaluated once.
- The value of the expression is compared to each case label.
- When a match is found, the corresponding code block is executed.
- Execution continues until a break statement is encountered or the switch block's end.
- If no match is found, the code block under default (if present) is executed.

5. Ternary Operators

- The ternary operator, also known as the conditional operator, provides a concise way to write conditional expressions in Java. It allows you to choose between two values or expressions based on a condition, all in a single line of code.
- **Basic Syntax**: condition ? valueIfTrue : valueIfFalse

Working:

- The condition is evaluated first.
- If the condition is true, the expression returns valueIfTrue.
- If the condition is false, the expression returns valueIfFalse.

```
int x = 10;
int y = 5;
int result = (x > y) ? x : y;

//if x is greater than y, the result will be assigned the value of x;
otherwise, it will be assigned the value of y.
```

Example of Nested Ternary Operator:

```
int x = 10;
int y = 20;
int z = 30;

// Find the maximum value among x, y, and z using a nested ternary
operator
int max = (x > y) ? ((x > z) ? x : z) : ((y > z) ? y : z);

System.out.println("The maximum value is: " + max);
```

Working:

- The first level of the ternary operator compares x and y using (x > y). If x is greater than y, it evaluates the expression ((x > z) ? x : z); otherwise, it considers the expression ((y > z) ? y : z).
- In the first expression (x > z) ? x : z, it checks if x is greater than z. If it is, it returns x; otherwise, it returns z.
- In the second expression (y > z) ? y : z, it checks if y is greater than z. If it is, it returns y; otherwise, it returns z.
- The result of the first-level ternary operator is the maximum value among x, y, and z, which is stored in the max variable.
- Ultimately, the program prints the maximum value, which is determined using the nested ternary operator.

Conclusion

In conclusion, learning about conditionals in Java gives us essential tools for controlling program flow and making decisions based on conditions. These constructs enhance code readability, flexibility, and efficiency, allowing us to create more intelligent and adaptable Java programs. We can write cleaner and more maintainable code by choosing the correct conditional construct for the task at hand and following best practices.

- if, if-else, and if-else-if-else statements offer ways to handle different conditions and scenarios.
- The switch statement is effective for managing multiple cases with distinct code paths.
- The ternary operator simplifies writing simple conditional expressions efficiently.

Iteration

Introduction

Iterations in Java are control structures that allow you to execute a block of code repeatedly once a specific condition is met to eliminate the need to rewrite code. Java provides several loops, each with its own use cases and syntax. Here are the main types of loops in Java:

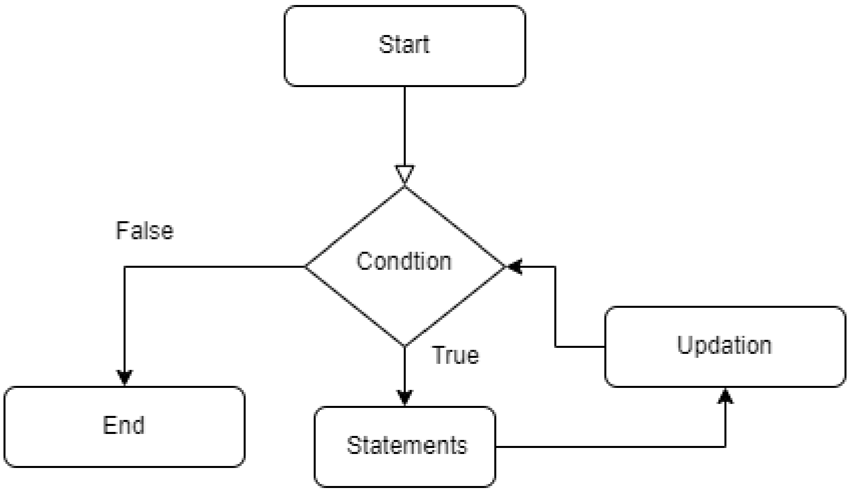
1. `for` Loop:

- The ‘for’ loop is the most commonly used loop in Java. It allows you to iterate over a sequence of values, typically numbers, for a specified number of times. The for loop consists of three steps: initialization, condition, and updation.
    1. Initialization: It's an expression used to initialise the loop variable.
    2. Condition: A boolean expression determining whether the loop should continue or terminate.
    3. Updation: It's an expression that updates the loop variable in each iteration.

The syntax is as follows:

```
for (initialization; condition; update) {
    // Code to be executed repeatedly
}
```

```
for (int i = 0; i < 5; i++) {
    System.out.println(i);
}
//Will print numbers 1 to 5.
```



- **`break` statement:** The break statement is used to exit a loop prematurely, even if the loop's condition is still true. When encountered, it terminates the innermost loop (where the break statement is located) and continues with the code after the loop.
- Example:

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break; // Exits the loop when i equals 5
    }
```

Note:- We can initialize variables outside the loop as well.

```
int i = 0;

for (; i < 5; i++) {
    System.out.println(i);
}
//Will print numbers 1 to 5.
```

2. `while` Loop

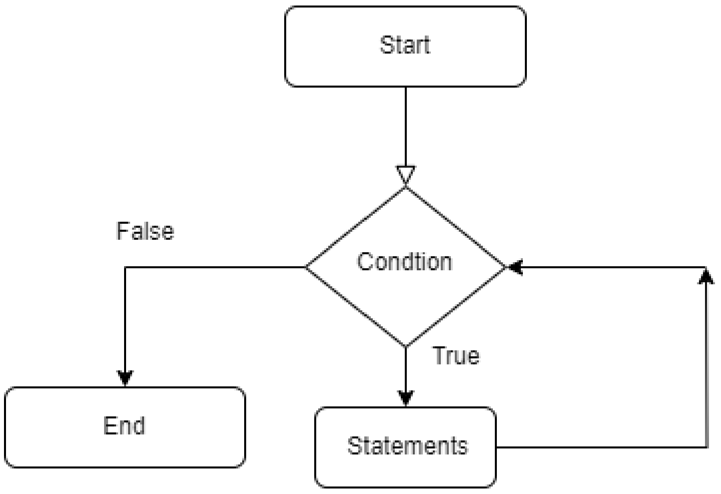
The `while` loop repeatedly executes a block of code as long as a specified condition is true. The syntax is as follows:

```
while (condition) {
    // Code to be executed repeatedly
}
```

Example:

```
int count = 0;
while (count < 5) {
    System.out.println(count);
    count++;
}

//will print numbers 0 to 4.
```

The above code can be visualized as below:



3. `do-while` Loop

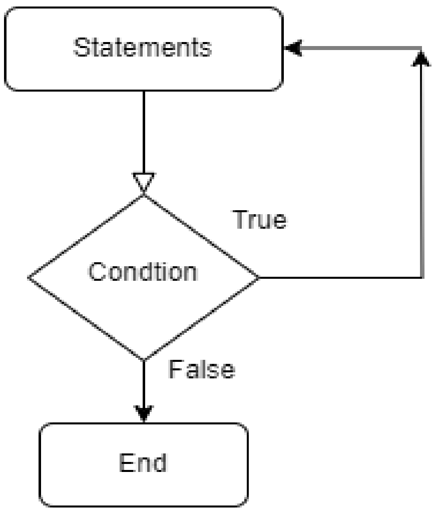
The do-while loop is similar to the while loop, but it guarantees that the code block is executed at least once before checking the condition. The syntax is as follows:

```
do {
    // Code to be executed repeatedly
} while (condition);
```

Example:

```
int number = 0;
do {
    System.out.println("Number: " + number);
    number++;
} while (number < 5);

//prints numbers 0 to 4.
```

The above code can be visualised as below:



4. Nested Loops

Nested loops in programming refer to placing one loop inside another loop. This technique is used to solve more complex problems involving iterating through multi-dimensional data structures, such as matrices, or performing repetitive tasks with multiple iteration levels. Here are some key points to understand about nested loops:

Nested loops are created by placing one loop inside another. Depending on your specific requirements, you can use any type of loop (e.g., for, while, while-while) as the outer loop or inner loop.

Example:

```
public class StarPattern {
    public static void main(String[] args) {
        int numRows = 5;

        // Outer loop for rows
        for (int i = 1; i <= numRows; i++) {
            // Inner loop for printing stars
            for (int j = 1; j <= i; j++) {
                System.out.print("* ");
            }
            System.out.println(); // Move to the next line after each row
        }
    }
}

//
Will print stars in pattern:
*
* *
* * *
* * * *
* * * * *
//
```

Key Points to Remember:

- Ensure that you initialise loop variables correctly and that the termination condition is well-defined. Failing to do so can result in infinite loops or loops that never execute.
- If a loop variable needs to be used outside the loop, declare it outside the loop and initialise it appropriately. This ensures that the variable retains its value after the loop exits.

Conclusion

Loops in Java are fundamental control structures that enable the repetition of code execution based on specific conditions. They are pivotal in automating tasks, iterating through data structures, and solving various programming problems. Here are key takeaways about loops in Java:

- Java offers different loop constructs such as for, while, and do-while, each tailored to specific looping scenarios.
- The `for` loop is commonly used for iterating over sequences and is ideal when the number of iterations is known in advance.
- The `while` and `do-while` loops are helpful when the loop's termination depends on dynamic conditions or user input.

Patterns

Introduction

Patterns in Java are a common programming exercise involving the creation of specific shapes or sequences of characters using loops. They serve as a valuable practice for understanding control structures and enhancing problem-solving skills. In this lesson, we'll explore three types of patterns: square patterns, triangle patterns, and pyramid patterns. We'll provide step-by-step explanations for each type, along with code examples using both stars and numbers.

Before printing any pattern, you must consider the following three things:

- The first step in printing any pattern is to figure out the number of rows that the pattern requires.
- Next, you should know how many columns are there in the i'th row.
- Once you have figured out the number of rows and columns, then focus on the pattern to print.

1. Square Patterns

- Printing a square pattern of stars.

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

Algorithm to follow:

Refer to the code block below simultaneously for easy understanding.

- Input the size of the square (e.g., size = 5).
- Use a nested loop structure with an outer loop controlling the rows and an inner loop controlling the columns.
- For each row, print asterisks (*) equal to the size of the square.
- Move to the next row by printing a newline character (System.out.println()).

```java
public class SquarePatternStars {
    public static void main(String[] args) {
        int size = 5; // Size of the square
        for (int i = 1; i <= size; i++) {
            for (int j = 1; j <= size; j++) {
                System.out.print("*"+” ”);
            }
            System.out.println();
        }
    }
}
```

- Printing a square pattern using numbers.
  Expected output:

```
1 1 1 1 1
2 2 2 2 2
3 3 3 3 3
4 4 4 4 4
5 5 5 5 5
```

Algorithm to follow:

Refer to the code block below simultaneously for easy understanding.

- Input the size of the square (e.g., size = 5).
- Use a nested loop structure with an outer loop controlling the rows and an inner loop controlling the columns.
- For each row, print the value of i, where i is the row number, in each column.
- Move to the next row by printing a newline character (System.out.println()).

```java
public class SquarePatternNumbers {
    public static void main(String[] args) {
        int size = 5; // Size of the square
        for (int i = 1; i <= size; i++) {
            for (int j = 1; j <= size; j++) {
                System.out.print(i + " ");
            }
            System.out.println();
        }
    }
}
```

2. Triangular Patterns

- To print a triangle pattern with a given number of rows, using asterisks.

  Expected output:

```
*
* *
* * *
* * * *
* * * * *
```

Algorithm to follow:

Refer to the code block below simultaneously for easy understanding.

- Input the number of rows for the triangle (e.g., size = 5).
- Use a nested loop structure with an outer loop controlling the rows and an inner loop controlling the columns.

- In each row, print asterisks (*) equal to the row number. For the first row, print one asterisk, for the second row, print two asterisks, and so on.
- Move to the next row by printing a newline character (System.out.println()).

```java
public class TrianglePatternStars {
    public static void main(String[] args) {
        int size = 5; // Number of rows
        for (int i = 1; i <= size; i++) {
            for (int j = 1; j <= i; j++) {
                System.out.print("* ");
            }
            System.out.println();
        }
    }
}
```

- Similarly for printing numbers with the pattern shown below:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Algorithm to follow:

(Refer to the code block below simultaneously for easy understanding).

- Input the number of rows for the triangle (e.g., size = 5).
- Use a nested loop structure with an outer loop controlling the rows and an inner loop controlling the columns.
- In each row, print numbers from 1 to i, where i is the row number. For the first row, print 1, for the second row, print 1 2, and so on.
- Move to the next row by printing a newline character (System.out.println()).

```java
public class TrianglePatternNumbers {
    public static void main(String[] args) {
        int size = 5; // Number of rows
        for (int i = 1; i <= size; i++) {
            for (int j = 1; j <= i; j++) {
                System.out.print(j + " ");
            }
            System.out.println();
        }
    }
}
```

- Same can be done for numbers with decreasing order:

```
5
5 4
5 4 3
5 4 3 2
5 4 3 2 1
```

Solution:

```java
public class TrianglePatternNumbersDescending {
    public static void main(String[] args) {
        int size = 5; // Number of rows
        for (int i = 1; i <= size; i++) {
            for (int j = size; j >= i; j--) {
                System.out.print(j + " ");
            }
            System.out.println();
        }
    }
}
```

3. Pyramid Pattern

- Pyramid pattern with a given number of rows using asterisks (*).
  Expected output:

```
        *
      * * *
    * * * * *
  * * * * * * *
* * * * * * * * *
```

Algorithm to follow:

(Refer to the code block below simultaneously for easy understanding.)

- Input the number of rows for the pyramid (e.g., size = 5).
- Use a nested loop structure with an outer loop controlling the rows and an inner loop controlling the columns.
- In each row, start by printing spaces for indentation. The number of spaces is equal to size - row number.
- After printing the spaces, print asterisks (*). The number of asterisks in each row is equal to 2 * row number - 1.
- Move to the next row by printing a newline character (System.out.println()).

```
public class PyramidPatternStars {
    public static void main(String[] args) {
        int size = 5; // Number of rows
        for (int i = 1; i <= size; i++) {
            // Print spaces for indentation
            for (int j = 1; j <= size - i; j++) {
                System.out.print("  ");
            }
            // Print asterisks in an ascending order
            for (int k = 1; k <= 2 * i - 1; k++) {
                System.out.print("* ");
            }
            System.out.println();
        }
    }
}
```

- Pyramid pattern with a given number of rows using numbers.
  Expected Output:

```
        1
      1 2 3
    1 2 3 4 5
  1 2 3 4 5 6 7
1 2 3 4 5 6 7 8 9
```

Algorithm to follow:

(Refer to the code block below simultaneously for easy understanding.)

- Input the number of rows for the pyramid (e.g., size = 5).
- Use a nested loop structure with an outer loop controlling the rows and an inner loop controlling the columns.
- In each row, start by printing spaces for indentation. The number of spaces is equal to (size - row) number.
- After printing the spaces, print numbers in an ascending order. The numbers start from 1 and go up to 2 * row number - 1.
- Move to the next row by printing a newline character (System.out.println()).

```
public class PyramidPatternNumbers {
    public static void main(String[] args) {
        int size = 5; // Number of rows
        for (int i = 1; i <= size; i++) {
            // Print spaces for indentation
            for (int j = 1; j <= size - i; j++) {
                System.out.print("  ");
            }
            // Print numbers in an ascending order
            for (int k = 1; k <= 2 * i - 1; k++) {
                System.out.print(k + " ");
            }
            System.out.println();
        }
    }
}
```

Conclusion

Patterns in Java offer an engaging and instructive programming exercise that involves using loops to create visually appealing shapes and sequences of characters. In this exploration of patterns, we've covered square patterns, triangle patterns, and pyramid patterns, each providing unique challenges and opportunities for enhancing programming skills. Here are key takeaways:

- **Fundamental Control Structures**: Creating patterns reinforces the understanding of fundamental control structures, including loops and conditional statements, which are essential in Java programming.
- **Algorithmic Thinking**: Patterns encourage algorithmic thinking, as designers must strategize on how to print characters or numbers in specific arrangements to achieve the desired pattern.
- **Nested Loops**: Nested loops play a crucial role in creating patterns. Understanding how to control inner and outer loops is fundamental to achieving complex designs.
- **Variety of Patterns**: Java allows for an array of patterns, from simple squares and triangles to more intricate pyramids and custom designs. Patterns can be adapted to suit various requirements and aesthetics.

Patterns II

Introduction

In addition to basic patterns, advanced patterns in Java offer more complex and visually interesting designs that can be created using loops and control structures. In this guide, we'll explore several advanced patterns, provide step-by-step explanations, and offer code examples.

1. Character Patterns

- Printing characters as shown below:

```
aaaa
bbbb
cccc
```

Solution :

```java
import java.util.Scanner;

public class CharacterPatternNRows {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the value of N: ");
        int N = scanner.nextInt();

        for (int i = 0; i < N; i++) {
            char currentChar = (char) ('a' + i);
            for (int j = 0; j < N; j++) {
                System.out.print(currentChar);
            }
            System.out.println();
        }
    }
}
```

Algorithm to follow:

- Import the Scanner class for user input.
- Create a Scanner object to read input from the user.
- Prompt the user to enter the value of N.
- Read the user's input into the variable N.
- Use a nested loop with two variables, i and j, to control the rows and columns.

- In the outer loop (i), iterate from 0 to N-1. For each value of i, calculate the currentChar as the character 'a' plus i. This will give us 'a' for i = 0, 'b' for i = 1, and so on.
- In the inner loop (j), iterate from 0 to N-1. For each value of j, print the currentChar.
- After printing N characters, move to the next line using System.out.println() to create rows of characters.
- Repeat steps 6-8 for each value of i to create N rows of characters.

- Printing characters as shown below:

```
abcd
bcde
cdef
```

Solution:

```
import java.util.Scanner;

public class CharacterPatternContinuous {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the value of N: ");
        int N = scanner.nextInt();

        for (int i = 0; i < N; i++) {
            char currentChar = (char) ('a' + i);
            for (int j = 0; j < N; j++) {
                System.out.print(currentChar);
                currentChar++;
            }
            System.out.println();
        }
    }
}
```

Algorithm to follow:

- Instead of printing the same character currentChar repeatedly, we increment currentChar inside the inner loop to print consecutive characters ('abcd', 'bcde', 'cdef', etc.).
- The outer loop (i) still controls the rows and the inner loop (j) controls the columns. Each row starts with the character corresponding to the current value of i.
- This code takes a user-defined value of N and prints character patterns 'abcd', 'bcde', 'cdef', and so on in rows.

2. Inverted Triangle Patterns

- Printing an inverted triangle pattern of asterisks(*), for example, if the input is ‘4’ the output should be

```
****
***
**
*
```

Solution:

```
import java.util.Scanner;

public class InvertedTrianglePatternAsterisks {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number of rows (N): ");
        int N = scanner.nextInt();

        for (int i = N; i >= 1; i--) {
            for (int j = 1; j <= i; j++) {
                System.out.print("*");
            }
            System.out.println();
        }
    }
}
```

Algorithm to follow:

- Prompt the user to enter the number of rows (N).
- Read the user's input into the variable N.
- Use a loop with the variable i to control the number of rows. The loop starts from N and goes down to 1.
- In each row, use another loop with the variable j to print asterisks ("). The inner loop iterates from 1 to i, printing " for each iteration.
- Use System.out.println() to move to the next line after completing each row.

- Printing an inverted triangle pattern of numbers. For example, if N = 5, the output should be

```
1 2 3 4 5
  1 2 3 4
    1 2 3
      1 2
        1
```

Solution:

```
import java.util.Scanner;

public class InvertedTrianglePattern {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number of rows (N): ");
        int N = scanner.nextInt();

        for (int i = 1; i <= N; i++) {
            // Print spaces for indentation
            for (int j = 1; j < i; j++) {
                System.out.print("  ");
            }

            // Print numbers in descending order
            for (int k = 1; k <= N - i + 1; k++) {
                System.out.print(k + " ");
            }
            System.out.println();
        }
    }
}
```

Algorithm to follow:

- Prompt the user to enter the number of rows (N).
- Read the user's input into the variable N.
- Use a nested loop with three variables, i, j, and k, to control the rows, spaces for indentation, and numbers.
- In the outer loop (i), iterate from 1 to N. This loop controls the number of rows in the inverted triangle.
- In the first inner loop (j), iterate from 1 to i - 1. This loop is responsible for printing spaces for indentation. The number of spaces increases with each row.

- After printing the spaces, enter another inner loop (k). This loop is responsible for printing the numbers in ascending order, starting from 1. The number of numbers in each row decreases from the top row to the bottom row.
- Use System.out.println() to move to the next line after completing each row.

3. Mirror Image Pattern

- Mirror image pattern with asterisks ('*'). Expected output:

```
        *
      * *
    * * *
  * * * *
* * * * *
```

Solution:

```
import java.util.Scanner;

public class MirrorImagePattern {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number of rows (N): ");
        int N = scanner.nextInt();

        for (int i = 1; i <= N; i++) {
            // Print spaces for indentation
            for (int j = 1; j <= N - i; j++) {
                System.out.print("  ");
            }

            // Print asterisks in ascending order
            for (int k = 1; k <= i; k++) {
                System.out.print("* ");
            }

            System.out.println(); // Move to the next line
        }
    }
}
```

Algorithm to follow:

- Prompt the user to enter the number of rows (N).
- Read the user's input into the variable N.

- Use a loop with the variable i to control the number of rows. The loop iterates from 1 to N.
- In each row, use another loop with the variable j to print spaces for indentation. The number of spaces decreases as we move to the right side of the pattern. We calculate the number of spaces as N - i.
- After printing the spaces, use a third loop with the variable k to print asterisks ('*') in ascending order. The number of asterisks increases from left to right, starting from 1 and going up to i.
- Use System.out.println() to move to the next line after completing each row.

4. Isosceles Triangle Pattern

- . For example, if N = 5, the output should be:

```
        1
      1 2
    1 2 3
  1 2 3 4
1 2 3 4 5
```

Solution:

```
import java.util.Scanner;

public class MirrorImagePatternNumbers {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number of rows (N): ");
        int N = scanner.nextInt();

        for (int i = 1; i <= N; i++) {
            // Print spaces for indentation
            for (int j = 1; j <= N - i; j++) {
                System.out.print("  ");
            }

            // Print numbers in ascending order
            for (int k = 1; k <= i; k++) {
                System.out.print(k + " ");
            }

            System.out.println(); // Move to the next line
        }
    }
}
```

Algorithm to follow:

- Prompt the user to enter the number of rows (N).
- Read the user's input into the variable N.
- Use a loop with the variable i to control the number of rows. The loop iterates from 1 to N.
- In each row, use another loop with the variable j to print spaces for indentation. The number of spaces decreases as we move to the right side of the pattern. We calculate the number of spaces as N - i.
- After printing the spaces, use a third loop with the variable k to print numbers in ascending order. The numbers start from 1 and go up to i.
- Use System.out.println() to move to the next line after completing each row.

Conclusion

In this exploration of advanced patterns in Java, we've covered a variety of fascinating patterns that enhance our understanding of programming logic and aesthetics. Let's summarize what we've learned from character patterns, inverted triangle patterns, mirror image patterns, and isosceles triangle patterns.

Mastering these advanced patterns not only enhances your programming skills but also develops your ability to visualize and create intricate designs in code. These patterns are valuable for both problem-solving and artistic expression in programming. As you continue your journey in programming, you'll find these pattern-building techniques applicable in various contexts, from creating engaging user interfaces to solving algorithmic challenges.

Methods In Java

Introduction

In Java, a method is a block of code that performs a specific task or action. Methods are used to organize and modularize code, making it more readable, reusable, and maintainable. Methods are an essential part of object-oriented programming (OOP) and play a crucial role in Java programs.

1. Method Declaration

A method is declared using the following syntax:

```
returnType methodName(parameters) {
    // Method body
}
```

- **returnType**: It specifies the type of value that the method returns. Use void if the method doesn't return anything.
- **methodName**: This is the name of the method, following Java naming conventions.
- **parameters**: These are the input values (arguments) that the method expects, if any.

Method Call: To execute a method, you call it by its name followed by parentheses:

```
public static void greet() {
    System.out.println("Hello, world!");
}

public static void main(String[] args) {
    greet(); // Method call
}
```

2. Methods With Arguments

Methods can accept input parameters, also known as arguments. These parameters allow you to pass data into the method for processing. Here's a basic structure of a method with arguments:

```java
public static void add(int num1, int num2) {
    int sum = num1 + num2;
    System.out.println("Sum: " + sum);
}

public static void main(String[] args)
    add(5, 3); // Method call with arguments
}
```

- Methods can have zero or more parameters (arguments) to accept input values.
- You can have multiple parameters separated by commas.
- Arguments passed to a method must match the parameter types and order.

3. Return Type of Methods

Methods can have return types, indicating the type of value they return after performing their tasks. A method can return a value using the return keyword. Here's the structure of a method with a return type:

```java
public static int add(int num1, int num2) {
    return num1 + num2; // Return an int
}

public static void main(String[] args) {
    int result = add(5, 3); // Method call with return value
    System.out.println("Result: " + result);
}
```

- Methods can have various return types, including primitive data types, objects, or custom types.
- Use the return statement to return a value from a method.

4. Pass By Value

Java uses "pass by value" when passing arguments to methods. This means that a copy of the argument's value is passed to the method, rather than the actual variable. Any modifications made to the parameter within the method do not affect the original variable. Here's an example:

```java
public static void modifyValue(int num) {
    num = num * 2;
}
```

```java
public static void main(String[] args) {
    int x = 5;
    modifyValue(x);
    System.out.println(x); // Output: 5 (unchanged)
}
```

In this example, the **modifyValue** method takes an int parameter, but when it modifies num, it doesn't affect the value of **x** in the main method because Java passes the value of **x (5)** to **modifyValue**.

5. Method Calling Method

In Java, it's possible for one method to call another method. It allows you to create modular and reusable code by breaking down complex tasks into smaller, more manageable parts.

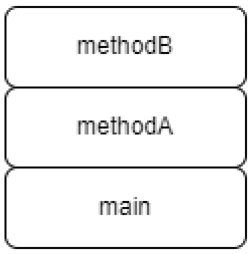
```java
public class Math{
    public static int add(int a, int b) {
        return a + b;
    }

    public static int multiply(int x, int y) {
        return x * y;
    }

    public static void main(String[] args) {
        int sum = add(5, 3); // Calling add method
        int product = multiply(4, 2); // Calling multiply method

        System.out.println("Sum: " + sum);
        System.out.println("Product: " + product);
    }
}
```

6. Call Stack In Java

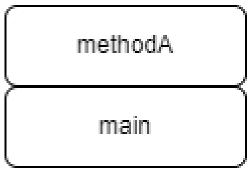
- The call stack is a fundamental concept in Java that keeps track of method calls during program execution.
- Each time a method is called, a new frame is added to the call stack.
- When a method completes, its frame is removed from the stack.
- This stack ensures that methods are executed in a last-in, first-out (LIFO) order.

```
public class CallStackExample {
    public static void main(String[] args) {
        methodA();
    }

    public static void methodA() {
        System.out.println("Method A is called.");
        methodB();
    }

    public static void methodB() {
        System.out.println("Method B is called.");
    }
}
```

When **main** calls **methodA**, a frame for **methodA** is added to the call stack. Inside **methodA**, **methodB** is called, and a frame for **methodB** is added. The stack looks like this -



As **methodB** finishes, its frame is removed, and the call stack returns to -



Finally, as **methodA** completes, its frame is removed, leaving only **main** on the stack.

7. Method Overloading

Method overloading in Java refers to the ability to define multiple methods within the same class with the same name but different parameter lists. The parameter lists can differ in the following ways:

- **Number of Parameters**: Overloaded methods can have a different number of parameters. For example, one method might have two parameters, while another has three.
- **Data Types of Parameters**: Overloaded methods can have parameters of different data types. For example, one method might take two integers as parameters, while another takes two doubles.
- **Order of Parameters**: The order of parameters in overloaded methods can be different. For example, one method might have parameters (int x, double y), while another has (double y, int x).

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

    public static double add(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        int sum1 = add(5, 3); // Calls the int version of add
        double sum2 = add(4.5, 2.3); // Calls the double version of add

        System.out.println("Sum 1: " + sum1);
        System.out.println("Sum 2: " + sum2);
    }
}
```

In this example, we have two add methods with different parameter types (int and double). The appropriate method is called based on the argument type.

- Method overloading allows you to define multiple methods in the same class with the same name but different parameter lists.
- The compiler differentiates between overloaded methods based on the number or types of parameters.
- Overloaded methods must have distinct parameter lists to avoid ambiguity.

Conclusion

In Java, methods play a pivotal role in organizing and structuring code for modularity and reusability. They are declared with a specific return type, a method name, and parameters. Methods can be invoked by their names with appropriate arguments, and their return types specify the data they yield. It's crucial to understand that Java employs "pass by value," meaning that when arguments are passed to methods, they are copies, and changes within the method remain local.

Additionally, the call stack in Java tracks method calls during program execution, operating on a last-in, first-out (LIFO) basis. As methods are invoked, frames are pushed onto the stack, and when a method completes, its frame is popped off the stack. This mechanism ensures proper execution flow.

Method overloading further enhances code flexibility by allowing multiple methods with the same name but distinct parameter lists. Overloaded methods are differentiated based on the number, data types, or order of parameters, making it easier to create intuitive and versatile APIs in Java. Together, these concepts form the foundation for building efficient, organized, and modular Java applications.

Arrays

Introduction

In cases where there is a need to use several variables of the same type for storing, for example, names or marks of ‘n’ students, we use a data structure called arrays.
Arrays are collections of fixed numbers of elements of a single type. Using arrays saves us from the time and effort required to declare each of the features of the array individually.

The length of an array is established when the array is created. After creation, its length is. For example, {1,2,3,4,5} is an array of integers.

Similarly, an array can be a collection of characters, Boolean, and Double.

Declaring Array Variables

To use an array in a program, you must declare a variable to refer to the array and specify the type (which, once specified, can’t be changed) of the array the variable can reference. Here is the syntax for declaring an array variable –

Syntax:

```
datatype [] arrayRefVar;
datatype arrayRefVar []
```

```
Int [] arr;
Int arr []
```

Creating Array:

Declaring an array variable does not create an array (i.e. no space is reserved for an array). Here is the syntax for creating an array –

arrayRefVar = new datatype [array Size];
Example:
arr=new int [20];

The above statement does two things –

It creates an array using the new keyword [array Size].

It assigns the reference of the newly created array to the variable arrayRefVar.

Combining the declaration of an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below–

```
datatype [] arrayRefVar = new datatype [array Size];
```

Array Indexes

To access different elements in an array -- all elements in the array are indexed, and indexing starts from 0. So if the array has five elements, the first index will be 0, and the last one will be 4. Similarly, if we have to store n values in an array, then indexes will range from 0 to n - 1. Trying to retrieve an element from an invalid index will give an **ArrayIndexOutOfBondsException**.

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

arr[0] = 1   arr[1] = 2   arr[2] = 3   arr[3] = 4   arr[4] = 5

Initialising an Array

In Single Line

Syntax of creating and initialising an array in a single line, dataType [] arrayRefVar =

{value0, value1, ..., value};

```
int [] arr= {1,2,3,4,5,6,7};
```

Loops

You can use loops to iterate through arrays. You can use several loops: for loop, while loop, and for each loop. Here's how you can use each of them to iterate through an array:

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

```
{
int [] arr = new int [20];
Scanner Scan = new Scanner(System. in);
for(int i = 0; i < arr.length; i++){
      arr[i]=Scan.nextInt();
      }
    }
```

For Each Loop

This is a special type of loop to access array elements of an array. But we can use this loop only to traverse an array. Nothing can be changed in the array using this loop.
The syntax is as follows:

```
for (datatype variable: array) {
    // code to be executed for each element in the array
}
```

Here, the data type is the data type of the elements in the array, the variable is a variable of that data type, and the array is the array you want to iterate through. The for-each loop automatically iterates through the array, assigning each element to the variable, and executes the code inside the loop for each element.

Here's an example of how you can use the enhanced for loop to iterate through an array:

```
public class Solutions {
public static void main (String [] args){
int [] arr= {10,20,30,40,50};
for (int i: arr) {
System.out.print(i+" ");
}
  }
```

How are Arrays Stored?

Arrays in Java store one of two things: primitive values (int, char) or references (a.k.a pointers).

When an object is created using “new”, memory is allocated on the heap, and a reference is returned. This is also true for arrays since arrays are objects.

```
int arr [] = new int [10]; //here arr is a reference to the array
```

Reassigning references and Garbage collector

All the reference variables (not final) can be reassigned again and again, but the data type to whom they will refer is fixed at the time of their declaration.

```
public class Solutions {

public static void main (String [] args) {

int [] arr = new int [20]; // HERE arr is a reference, not array name...
int [] arr1 = new int [10];

arr = arr1; // We can re-assign arr to the arrays which is referred by arr1.
Both arr and arr1 refer to the same arrays now.
}
}
```

In the above example, we create two reference variables and arr1. So now there are also two objects in the garbage collection heap.

Suppose you assign the arr1 reference variable to arr. In that case, no reference will be present for the 20 integer space created earlier, so the Garbage Collector can now free this memory block.

Passing Arrays to Functions

Passing Array as a function parameter

In Java programming language, the parameter passing is always made by value. Whenever we create a type variable, we copy its value to the method.

Passing Reference Type

As we studied in the lecture, we store references of Non-Primitive data types and access them via references. In such cases, the references of Non-Primitives are passed to function.

Arrays are objects, and when you pass an array to a method, you're actually passing the reference to the array. This means any modifications made to the array within the method will affect the original array because it's the same underlying data structure being accessed.

```
public class Solutions {
public static void print(int [] arr)
{

for (int i=0; i<5;i++) {

          System.out.print(arr[i]+" ");
      }
  }

public static void main (String [] args) {

int [] arr= {1,2,3,4,5};
    print(arr);
  }
}
```

In the provided code:

- The print method takes an array (which is a reference type) as an argument (pass-by-value), allowing it to print the contents of the array.
- The main method creates an array and calls the print method, passing the array as an argument (pass-by-value, where the value is the reference to the array).
- In summary, while Java is a "pass-by-value," the distinction is essential when working with reference types like arrays. You're passing a copy of the reference (address), and modifications to the object (or array) inside the method affect the original object because it's the same underlying object referenced.

Similarly, when we pass an array to the increment function shown below, the reference(address) to the array is passed, not the array itself.

```
public class Solutions {
public static void increment (int [] arr){
for (int i=0; i<5;i++){
arr[i]++;
}
}
public static void main (String [] args) {
int [] arr= {1,2,3,4,5};
increment(arr);
for (int i=0; i<5;i++){
System.out.print(arr[i]+" ");
}
 }
```

Output:

```
2 3 4 5 6
```

Here the reference to the array was passed. Thus, the inside increment function arr refers to the same array created in the main. Hence, the changes by increment function are performed on the same array and will reflect in the main.

Arrays II

Insertion:

Insertion in an array is typically needed in various scenarios where you need to add elements to the array at a specific position. Here are some common cases where array insertion is necessary:

Dynamic Data Storage:

- Arrays provide a way to store and manage data dynamically. Insertion allows for expanding the array dynamically by adding new elements at specific positions.

Ordered Data:

- When maintaining an ordered collection of data, you might need to insert elements while keeping the order intact. For example, maintaining a sorted list of names and inserting a new name in its correct position.

User Input:

- When dealing with user input, you may need to insert elements at specific positions based on user interactions or requests. For instance, inserting an item at a user-defined position in a shopping cart.

Moving Data:

- Shifting elements by inserting can be required when reorganizing or restructuring data in an array, such as when performing sorting or rearranging elements for efficient searching.

Efficient Memory Management:

- Sometimes, when working with limited memory, you may need to carefully manage data by inserting elements at specific positions to optimize memory usage and access patterns.

These cases highlight the need for array insertion, whether for managing data dynamically or maintaining order. The ability to insert elements into an array is fundamental for solving a wide range of programming problems effectively.

Let's Look at one of the problem statements:

Problem Statement: We need to insert an element at a specified position b in the array by shifting the existing elements to the right.

- We start by checking if the given position b is valid. If it's less than 0 or greater than the array size, it's an invalid position.
- We then iterate through the array in reverse starting from the last element up to the position b. For each element, we shift it one position to the right.
- After creating space for the new element, we insert the new value at the specified position b.

Now, let's look at the pseudo-code:

```java
public class ArrayInsertion {
    public static void insertElement(int[] arr, int n, int b, int
newValue) {
        if (b < 0 || b > n) {
            System.out.println("Invalid position for insertion.");
            return;
        }
        for (int i = n - 1; i >= b; i--) {
            arr[i + 1] = arr[i];
        }
        arr[b] = newValue;
    }
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int n = 5;  // Current size of the array
        int b = 2;  // Position to insert the new value
        int newValue = 10;  // Value to insert
        insertElement(arr, n, b, newValue);
        // Print the updated array
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
    }
}
```

Reversing an Array:

Reversing an array is a versatile operation that finds applications in various domains. It allows for efficient handling of data and enables different operations based on the reversed order of the elements.

Problem Statement:

The goal is to reverse an array using two different approaches: one by using extra space to store the reversed array and the other by using the two-pointer approach to reverse the array in place.

Approach 1: Using Extra Space

```
public class ArrayReversal {
    public static int[] reverseArrayWithExtraSpace(int[] arr, int n) {
        int[] reversedArray = new int[n];

        for (int i = n - 1; i >= 0; i--) {
            reversedArray[n - 1 - i] = arr[i];
        }
        return reversedArray;
    }
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int n = arr.length;
        int[] reversedArray = reverseArrayWithExtraSpace(arr, n);
        // Print the original and reversed arrays
        System.out.println("Original array: " + Arrays.toString(arr));
        System.out.println("Reversed array:"+
Arrays.toString(reversedArray));
    }
}
```

Pseudo Code Explanation:

- We create a new array called reversedArray to store the reversed elements.
- We iterate through the original array in reverse order and copy each element to the reversedArray.
- Finally, we return the reversedArray.

Approach 2: Using Two-Pointers

```java
public class ArrayReversal {

    public static void reverseArrayTwoPointers(int[] arr, int n) {
        int start = 0;
        int end = n - 1;

        while (start < end) {
            // Swap arr[start] and arr[end]
            int temp = arr[start];
            arr[start] = arr[end];
            arr[end] = temp;
            // Move the pointers towards the center of the array
            start++;
            end--;
        }
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int n = arr.length;

        // Reverse the array using two pointers
        reverseArrayTwoPointers(arr, n);

        // Print the reversed array
        System.out.println("Reversed array: " + Arrays.toString(arr));
    }
}
```

Pseudo Code Explanation:

- We use two pointers, start and end, initially pointing to the beginning and end of the array, respectively.
- We swap elements at the start and end positions and then move the pointers towards the center until they meet.
- This process effectively reverses the array in place.

Push Zeroes

Pushing zeroes to the end of an array is a fundamental operation that has many applications in data. It allows for efficient handling of data by segregating zero and non-zero elements, enabling various operations based on this segregation.

Problem Statement:

You are given an array arr of integers. Your task is to push all the zeros in the array to the end while maintaining the order of non-zero elements.

```
public class ArrayZeroesToEnd {

    public static void pushZeroesToEnd(int[] arr, int n) {
        int leftPointer = 0;
        int rightPointer = 0;

        while (rightPointer < n) {
            if (arr[rightPointer] != 0) {
                // If the element is non-zero, swap arr[leftPointer]
with arr[rightPointer]
                int temp = arr[leftPointer];
                arr[leftPointer] = arr[rightPointer];
                arr[rightPointer] = temp;
                leftPointer++;
            }
            rightPointer++;
        }
    }

    public static void main(String[] args) {
        int[] arr = {0, 1, 0, 3, 12};
        int n = arr.length;
        // Push zeroes to the end of the array
        pushZeroesToEnd(arr, n);
        // Print the modified array
        System.out.println("Array after pushing zeroes to the end: " +
Arrays.toString(arr));
    }
}
```

Explanation:

Initialization:

- leftPointer = 0: Initialize a variable left pointer to 0, which will point to the current position where a non-zero element can be placed.
- rightPointer = 0: Initialize a variable rightPointer to 0, which will traverse the array.

Traverse the Array using a Single Loop with Two Pointers:

- while rightPointer < n do: Continue the loop while rightPointer is less than the array size.
- if arr[rightPointer] != 0 then: Check if the current element is not zero (a non-zero element).
- swap arr[leftPointer] with arr[rightPointer]: Swap the element at leftPointer with the non-zero element at rightPointer, effectively moving all non-zero elements to the front of the array.
- leftPointer = leftPointer + 1: Increment the leftPointer to indicate the addition of a non-zero element.
- rightPointer = rightPointer + 1: Increment rightPointer to move both pointers together.

Array - Rotation:

Good Going !!
Now, Let's cover the problem statement, pseudo code, and explanation for array rotation using the brute force approach, both left and right rotations.

Problem Statement:

You are given an array arr of integers and an integer x. Your task is to rotate the array by x positions either to the left or to the right.

Right Rotation:

- For right rotation, use the formula: (i + x) % length, where i is the current index and length is the length of the array.

```java
public class ArrayRotation {

    public static void rotateRight(int[] arr, int n, int x) {
        x = x % n;  // Handle cases where x > n
        int[] rotatedArr = new int[n];
        for (int i = 0; i < n; i++) {
            rotatedArr[(i + x) % n] = arr[i];
        }
        // Copy rotated array back to the original array
        for (int i = 0; i < n; i++) {
            arr[i] = rotatedArr[i];
        }
    }
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int n = arr.length;
        int x = 2;  // Number of positions to rotate
        // Rotate the array to the right by x positions
        rotateRight(arr, n, x);
        // Print the rotated array
        System.out.println("Array after rotating " + x + " positions to the
right: " + Arrays.toString(arr));
    }
}
```

Explanation:

- x = x % n: Normalize the rotation amount x to ensure it's less than the array n's length.
- rotatedArr[(i + x) % n] = arr[i]: For each element in the original array, place it at the (i + x) % n position in the rotated array, effectively performing a right rotation.
- Copying Back to Original Array: (Optional): After rotation, copy the elements from the rotated array back to the original array to reflect the rotation.

Left Rotation:

For left rotation, use the formula: (i + x - length) % length, where i is the current index and length is the length of the array.

```java
import java.util.Arrays;

public class ArrayRotation {

    public static void rotateLeft(int[] arr, int n, int x) {
        x = x % n;  // Handle cases where x > n
        int[] rotatedArr = new int[n];
        for (int i = 0; i < n; i++) {
            rotatedArr[(i - x + n) % n] = arr[i];
        }

        // Copy rotated array back to the original array
        for (int i = 0; i < n; i++) {
            arr[i] = rotatedArr[i];
        }
    }
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int n = arr.length;
        int x = 2;  // Number of positions to rotate
        // Rotate the array to the left by x positions
        rotateLeft(arr, n, x);
        // Print the rotated array
        System.out.println("Array after rotating " + x + " positions to the 
left: " + Arrays.toString(arr));
    }
}
```

Explanation

- x = x % n: Normalize the rotation amount x to ensure it's less than the array n's length.
- rotatedArr[(i + x - n) % n] = arr[i]: For each element in the original array, place it at the (i + x - n) % n position in the rotated array, effectively performing a left rotation.
- Copying Back to Original Array: (Optional): After rotation, copy the elements from the rotated array back to the original array to reflect the rotation.

These pseudocodes outline the steps for rotating an array to the right and to the left using the brute force approach. The rotation is achieved by placing the elements in the new positions calculated based on the given rotation amount x.

Rotate Array: Optimized Approach:

The optimized approach for array rotation involves a reversal technique. Let's cover the problem statement, pseudo code, and explanation for array rotation using this optimized approach, both left and right rotations.

Problem Statement:

You are given an array arr of integers and an integer x. Your task is to rotate the array by x positions either to the left or to the right.

Optimized Approach for Array Rotation:

Right Rotation:

- Reverse the entire array.
- Reverse the subarray from index 0 to x-1.
- Reverse the subarray from index x to length-1.

```
public class ArrayRotation {

    public static void rotateRight(int[] arr, int n, int x) {
        x = x % n;  // Handle cases where x > n
        reverseArray(arr, 0, n - 1);
        reverseArray(arr, 0, x - 1);
        reverseArray(arr, x, n - 1);
    }

    public static void reverseArray(int[] arr, int start, int end) {
        while (start < end) {
            // Swap arr[start] and arr[end]
            int temp = arr[start];
            arr[start] = arr[end];
            arr[end] = temp;
            start++;
            end--;
        }
    }
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int n = arr.length;
```

```java
        int x = 2;  // Number of positions to rotate
        // Rotate the array to the right by x positions
        rotateRight(arr, n, x);
        // Print the rotated array
        System.out.println("Array after rotating " + x + " positions to 
the right: " + Arrays.toString(arr));
    }
}
```

Left Rotation:

- Reverse the entire array.
- Reverse the subarray from index 0 to length-x-1.
- Reverse the subarray from index length-x to length-1.

```java
public class ArrayRotation {
    public static void rotateLeft(int[] arr, int n, int x) {
        x = x % n;  // Handle cases where x > n
        reverseArray(arr, 0, n - 1);
        reverseArray(arr, 0, n - x - 1);
        reverseArray(arr, n - x, n - 1);
    }

    public static void reverseArray(int[] arr, int start, int end) {
        while (start < end) {
            // Swap arr[start] and arr[end]
            int temp = arr[start];
            arr[start] = arr[end];
            arr[end] = temp;
            start++;
            end--;
        }
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int n = arr.length;
        int x = 2;  // Number of positions to rotate

        // Rotate the array to the left by x positions
        rotateLeft(arr, n, x);

        // Print the rotated array
        System.out.println("Array after rotating " + x + " positions to
the left: " + Arrays.toString(arr));
    }
}
```

Explanation:

- x = x % n: Normalize the rotation amount x to ensure it's less than the array n's length.
- Reverse Entire Array: reverseArray(arr, 0, n-1): Reverse the entire array.

Reverse Subarrays:

- For right rotation:
  - Reverse the subarray from index 0 to x-1.
  - Reverse the subarray from index x to length-1.
- For left rotation:
  - Reverse the subarray from index 0 to length-x-1.
  - Reverse the subarray from index length-x to length-1.

Reverse Array Procedure:

- reverseArray procedure swaps elements in the given range to reverse the array or subarray.

By reversing the entire array and then reversing appropriate subarrays, we achieve the desired rotation in an optimized manner. This approach avoids multiple rotations and provides a more efficient solution.

Strings

Character Arrays

Character arrays in Java are used to store sequences of characters. They are declared similarly to other arrays in Java but specifically hold characters.

Declaration and initialization of character array:

```
char[] charArray = {'a', 'b', 'c', 'd'}; // Initializing a character array
```

Here, we declare a reference variable charArray of type char[] (a character array). Then, we initialize charArray by creating a new character array and assigning it the values 'a', 'b', 'c', 'd'.

Strings in Java:

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects(we study about objects in detail in the OOPS lecture). The Java platform provides the String class to create and manipulate strings. The most direct and easiest way to create a string is to write: String str = "Hello world"; In the above example, "Hello world!" is a string literal—a series of characters in code that is enclosed in double quotes. Whenever it encounters a string literal in code, the compiler creates a String object with its value—in this case, Hello world!. Note: Strings in Java are immutable thus, we cannot modify their value. If we want to create a mutable string, we can use StringBuffer and StringBuilder classes.

A String can be constructed by either:

Directly assigning a string literal to a String reference - just like a primitive, or via the "new" operator and constructor, similar to any other classes(like arrays and scanner). However, this is not commonly used and is not recommended.

For example,

```
String str1 = "Java is Amazing!"; // Implicit construction via string
literal String str2 = new String("Java is Cool!"); // Explicit
construction via new
```

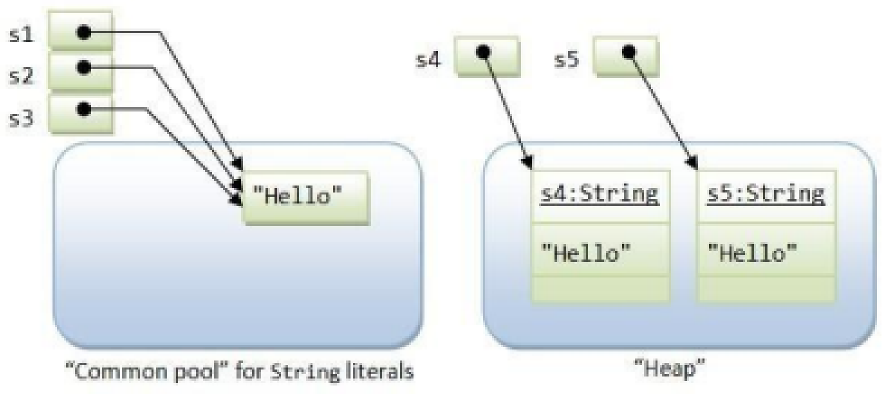
In the first statement, str1 is declared as a String reference and initialized with a string literal "Java is Amazing." In the second statement, str2 is declared as a String reference and initialized via the new operator to contain "Java is Cool." String literals are stored in a common pool called String pool. This facilitates the sharing of storage for strings with the same contents to conserve storage. String objects allocated via the new operator are stored in the heap memory(all non-primitives created via the new operator are stored in heap memory), and there is no sharing of storage for the same contents.

Creating strings:

As mentioned, there are two ways to construct a string:
Implicit construction by assigning a string literal or explicitly creating a String object via the new operator and constructor.

For example,

```
String s1 = "Hello"; // String literal String s2 = "Hello"; // String

literal String s3 = s1; // same reference String s4 = new

String("Hello"); // String object String s5 = new String("Hello"); //
String object
```



Java has provided a special mechanism for keeping the String literals -- in a so-called string common pool. If two string literals have the same contents, they will share the same storage inside the common pool. This approach is adopted to conserve storage for frequently--used strings. On the other hand, String objects created via the new operator and constructor are kept in the heap memory. Each String object in the heap has its own storage just like any other object. There is no sharing of storage in the heap even if two String objects have the same contents.

Important Java Methods :

1 . String "Length" Method :

String class provides an inbuilt method to determine the length of the Java String.

For example

```
String str1 = "test string"; //Length of a String   System.out.println("Length of String: " +
str.length());
```

2 . String "indexOf" Method:

String class provides an inbuilt method to get the index of a character in Java String.

For example

```
String str1 = "the string";
System.out.println("Index of character 's': "  + str_Sample.indexOf('s')); // returns 5
```

3 . String "charAt" Method:

Similar to the above question, given the index, how do I know the character at that location? Simple one again!! Use the “charAt” method and provide the index whose character you need to find.

For example :

```
String str1 = "test string";
System.out.println("char at index 3 : " + str.charAt()); // output - 't'
```

4. String "CompareTo" Method :

This method is used to compare two strings. Use the method “compareTo” and specify the String that you would like to compare. Use “compareToIgnoreCase” in case you don’t want the result to be case-sensitive. The result will have the value 0 if the argument string is equal to this string, a value less than 0 if this string is lexicographically less than the string argument and a value greater than 0 if this string is lexicographically greater than the string argument.

For example :

```
String str = "test";

System.out.println("Compare To "test": " + str.compareTo("test"));

//Compare to -- Ignore case System.out.println("Compare To "test": --

Case Ignored: " + str.compareToIgnoreCase("Test"));
```

5. String "Contain" Method :

Use the method “contains” to check if a string contains another string and specify the characters you need to check. Returns true if and only if this string contains the specified sequence of char values

For example:

```
String str = "test string";
System.out.println("Contains sequence ing: " + str.contains("ing"));
```

6. String "endsWith" Method :

This method is used to find whether a string ends with a particular prefix or not. Returns true if the character sequence represented by the argument is a suffix of the character sequence represented by this object

For example:

```
String str = "star";
System.out.println("EndsWith character 'r': " + str.endsWith("r"));
```

7. String "replaceAll" & "replaceFirst" Method:

Java String Replace, replaceAll and replaceFirst methods. You can specify the part of the String you want to replace and the replacement String in the arguments.

For example:

```
String str = "sample string"; System.out.println("Replace sample with test: " +
str.replace("sample", "test"));
```

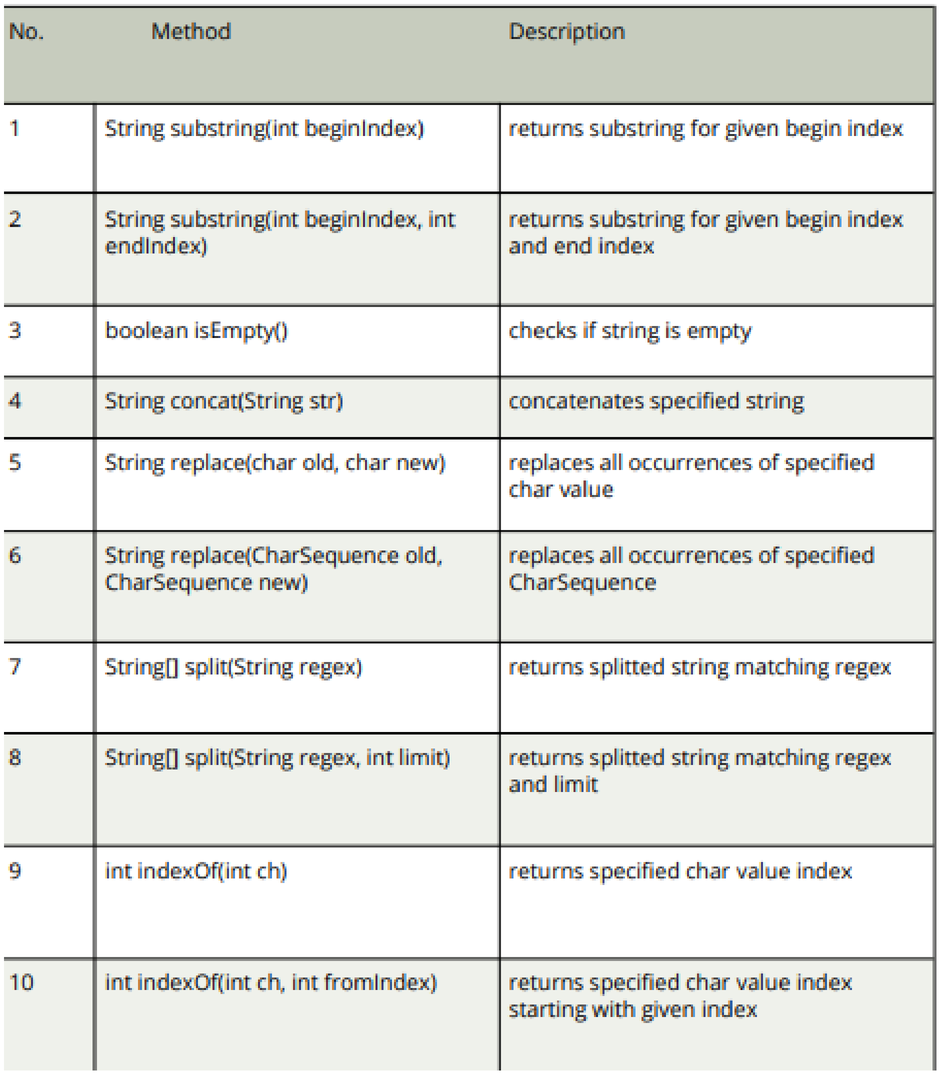
8. String Java "tolowercase" & Java "touppercase":

Use the “toLowercase()” or “ToUpperCase()” methods against the Strings that need to be converted

For example

```
String str = "TEST string";
System.out.println("Convert to LowerCase: " + str.toLowerCase());
System.out.println("Convert to UpperCase: " + str.toUpperCase());
```

Other Important Java Strings Methods:



No.	Method	Description
1	String substring(int beginIndex)	returns substring for given begin index
2	String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index
3	boolean isEmpty()	checks if string is empty
4	String concat(String str)	concatenates specified string
5	String replace(char old, char new)	replaces all occurrences of specified char value
6	String replace(CharSequence old, CharSequence new)	replaces all occurrences of specified CharSequence
7	String[] split(String regex)	returns splitted string matching regex
8	String[] split(String regex, int limit)	returns splitted string matching regex and limit
9	int indexOf(int ch)	returns specified char value index
10	int indexOf(int ch, int fromIndex)	returns specified char value index starting with given index

Strings are Immutable

Since string literals with the same contents share storage in the common pool, Java's String is designed to be immutable. That is, once a String is constructed, its contents cannot be modified. Otherwise, the other String references sharing the same storage location will be affected by the change, which can be unpredictable and, therefore, undesirable. Methods such as toUpperCase() might appear to modify the contents of a String object. In fact, a completely new String object is created and returned to the caller. The original String object will be deallocated once there are no more references and subsequently garbage-collected. Because

String is immutable, it is not efficient to use String if you need to modify your string frequently (that would create many new Strings occupying new storage areas).

For Example,

// inefficient code

```
String str = "Hello";
for (int i = 1; i < 1000; ++i)
 {
 str = str + i;
 }
```

StringBuffer

As explained earlier, Strings are immutable because String literals with the same content share the same storage in the string common pool. Modifying the content of one String directly may cause adverse side effects to other Strings sharing the same storage.

StringBuffer object is just like any ordinary object, which is stored in the heap and not shared, and therefore, can be modified without causing adverse side effects to other objects. The StringBuilder class was introduced in JDK 1.5. It is the same as the StringBuffer class, except that StringBuilder is not synchronized for multi-thread operations(you can read more about multi-threading). However, for a single-thread program, StringBuilder, without the synchronization overhead, is more efficient.

2D Arrays

Introduction:

- A 2D array is a combination of many 1D arrays.
- It represents a table/matrix with rows and columns of data.
- In this type of array, the position of a data element is referred to by two indices instead of one.

Consider an example of recording temperatures 4 times a day, 4 days in a row. Such data can be presented in a 2D array as below.

```
Day 1 - 11, 12, 5, 2
Day 2 - 15, 6, 10, 6
Day 3 - 10, 8, 12, 5
Day 4 - 12, 15, 8, 6
```

The above data can be represented as a 2D array as below.

```
int[][] T = {{11, 12, 5, 2}, {15, 6, 10, 6}, {10, 8, 12, 5}, {12, 15, 8,
6}};
```

How are 2D arrays Stored?

When we define this:

```
int[][] T = {{11, 12, 5, 2}, {15, 6, 10, 6}, {10, 8, 12, 5}, {12, 15,
8, 6}};
```

1. Here, T is a variable that stores the address of the 1D array of type array.
2. Further each index of this 1D also contains the address of arrays of type int. And finally these int arrays will contain the integers which are stored by the user.

Accessing Values in a Two-Dimensional Array

The data elements in a two-dimensional array can be accessed using two indices. One index refers to the main or parent array(inner array) and another index refers to the position of the data element in the inner array. If we mention only one index then it means we are talking about the entire inner array for that index position.

The example below illustrates how it works :

```
int[][] T = {{11, 12, 5, 2}, {15, 6, 10, 6}, {10, 8, 12, 5}, {12, 15, 8, 6}};

System.out.println(T[0]); //array at index 0 in T

System.out.println(T[1][2]); //Element at index 2 in array, at index // 1 in T
```

When the above code is executed, it produces the following result –

```
[I@6d06d69c // address where the 1D array is stored 10
```

Updating Values in Two-Dimensional Array

We can update the entire inner list or some specific data elements of the inner list by reassigning the values using the list index.

```
int[][] T = {{11, 12, 5, 2}, {15, 6, 10, 6}, {10, 8, 12, 5}, {12, 15, 8,
6}};
```

This can be done the same way as in 1D arrays. Consider the examples given below:

```
T[0][1]= 1 // Update element at index 1 of the array at index 0 of T to 1
T[1][1]= 7 // similarly at 1,1 to 7
```

This would modify the list as follows:

```
int[][] T = {{11, 1, 5, 2}, {15, 7, 10, 6}, {10, 8, 12, 5}, {12, 15, 8,
6}};
```

Printing a Two Dimensional Array

We can use nested for loops to print a 2D List. The outer loop iterates over the inner lists and the inner nested loop prints the elements of the lists. This can be shown as follows:

```
int[][] T= {{11,12,5,2},{15,6,10,6},{0,5,11,13},{10,8,12,5},{12,15,8,6}};

for(int i=0; i<T.length(); i++){ //Every row in T is an array
for(int j=0; j<T[i].length(); j++){ //Every col of the array row is element
System.out.print(T[i][j] + " "); //Print element
}
System.out.println(); //New Line
}
```

The above code will print T as:

```
11 12 5 2
15 6 10 6
0 5 11 13
10 8 12 5
12 15 8 6
```

Input Of Two Dimensional Arrays

For this, we will require the user to specify the number of rows (say row) and columns (say col) of the 2D array. The user will then enter all the elements of the 2D list in a single line. The java code for this can be written as follows:

```
public static void main(String[] args) {
 Scanner s = new Scanner(System.in);
 int row = s.nextInt();
 int col = s.nextInt();
 int[][] input = new int[row][col];
 for(int i=0; i<row; i++){ //Every row in T is an array
 for(int j=0; j<col; j++){ // each col of row
 input[i][j] = s.nextInt(); //take input
 }
 }
 }
```

User Input:

```
4
5
11 12 5 2 15
6 10 6 0 5
11 13 10 8 12
5 12 15 8 6
```

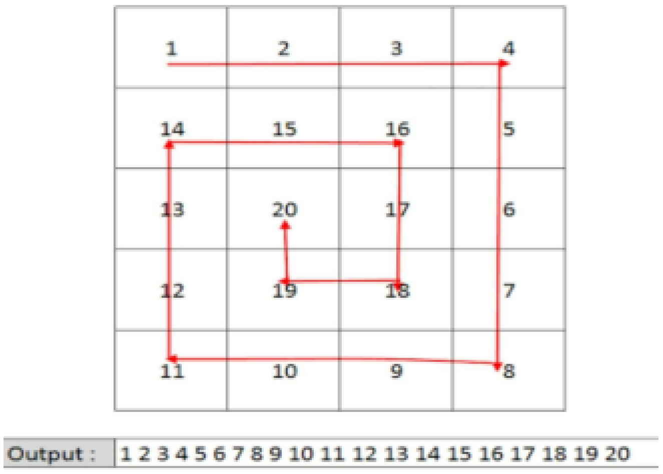
Now, this given input can be printed using the previous code for printing the 2D array.

Problem Statement - Spiral print

Given an NxM 2D array, you need to print it in a spiral fashion. See the below shown diagram for a better understanding.

Problem Statement:

You are given an array arr of integers and an integer x. Your task is to rotate the array by x positions either to the left or to the right.



Approach

- There needs to be a total of NxM elements printed.
- The problem can be solved by dividing the matrix into loops or squares or boundaries.

- It can be seen that the elements of the outer loop are printed first in a clockwise manner then the elements of the inner loop are printed. So, printing the elements of a loop can be solved using four loops which prints all the elements.

- Every ‘for’ loop defines a single-direction movement along with the matrix. The first for loop represents the movement from left to right, whereas the second crawl represents the movement from top to bottom, the third represents the movement from right to left, and the fourth represents the movement from bottom to up.

- Since there are NxM elements that need to be printed, so, we can wrap all the above code in one single for loop and let it run for until the loop prints NxM elements.

Problem Statement - Wave print (Assignment Question)

Given a 2D array, you need to print in this 2D array in a wave fashion. Wave fashion means in a sine wave order, that means, 1st column needs to be printed from top to bottom, then 2nd column needs to be printed from bottom to top, then again the next column from top to bottom and so on until we reach the last column.

Approach

1. Traverse the 2D array column-wise.
2. Try to keep track of the column which you are printing:
 - If the column number is even, move from top to bottom, i.e., start row from 0 to end.
 - Else, if the column is odd, move from bottom to top, i.e., start row number from end and decrement it until it reaches zero.

Jagged Arrays

Till now, we have seen 2D arrays with an equal number of columns in all rows. Jagged arrays are two-dimensional arrays with a variable number of columns in various rows. Some examples of jagged arrays are shown below:

```
int[][] T1 = {{11,12,5,2},{10,6},{10,8,12},{12}};
int[][] T2 = {{1,2,3}, {1}, {3}, {1,2}};
int[][] T3 = {{1},{},{1,2,1},{0}};
```

The elements in a jagged list can be accessed the same way as shown above using valid indices. Note: Keep in mind the range of columns for various rows. Indexes out of the range can produce an indexOutOfBound error.

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

 Scanner s = new Scanner(System.in);
 int rows = s.nextInt();
 int[][] input = new int[row][]; // we don't define columns here as this will
vary for each column
 for(int i=0; i<rows; i++){ //Every row in T is an array
 int cols = s.nextInt(); // number of columns for this row
 for(int j=0; j<cols; j++){ // each col of row
 input[i][j] = s.nextInt(); //take input
}
}
}
```

You can now define the total number of rows and for each row , you first need to define the number of columns before entering the elements in that particular column.

Searching And Sorting

Searching

Searching means to find out whether a particular element is present in a given sequence or not. There are commonly two types of searching techniques:

- Linear search (We have studied about this in **Arrays**)
- Binary search

In this module, we will be discussing binary search.

Binary Search

Binary Search is a searching algorithm for finding an element's position in a sorted array. In a nutshell, this search algorithm takes advantage of a collection of elements being already sorted by ignoring half of the elements after just one comparison.

Prerequisite: Binary search has one pre-requisite; unlike the linear search where elements could be any order, the array in **binary search**must be sorted,

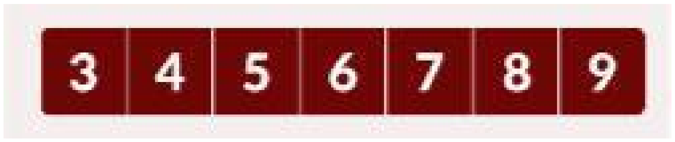
The algorithm works as follows:

1. Let the element we are searching for, in the given array/list is X.
2. Compare X with the middle element in the array.
3. If X matches with the middle element, we return the middle index.
4. If X is greater than the middle element, then X can only lie in the right (greater) half subarray after the middle element, then we apply the algorithm again for the right half.

5. If X is smaller than the middle element, then X must lie in the left (lower) half, this is because the array is sorted. So we apply the algorithm for the left half

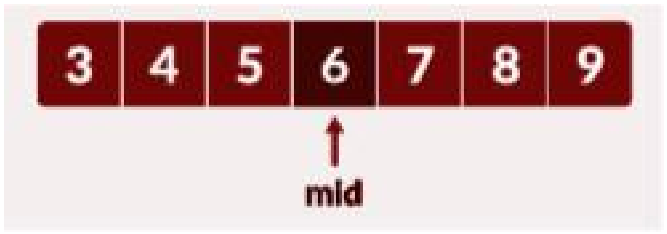
Example Run

- Let us consider the array to be:

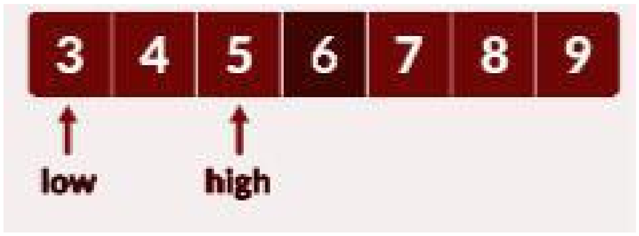


- Let x = 4 be the element to be searched.

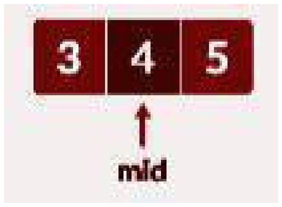
- Set two pointers **low**and **high**at the first and the last element respectively.
- Find the middle element **mid**of the array ie. **arr[(low+high)/2]** = 6.



- If **x == mid**, then return **mid**. Else, compare the element to be searched with **m**.
- If **x > mid**, compare x with the middle element of the elements on the right side of mid. This is done by setting low to **low = mid + 1**.
- Else, compare x with the middle element of the elements on the left side of mid. This is done by setting high to **high = mid - 1**.



- Repeat these steps until low meets high. We found 4:



Java Code

```
// Function to implement Binary Search Algorithm
public static int binarySearch(int arr[], int n, int x) { int start = 0, end = n -
        1;
    // Repeat until the pointers start and end meet each other
        while(start <= end) {

                int mid = (start + end) / 2; // Middle Index if(arr[mid] == x) { //
                element found
                        return mid;
                }
                else if(x < arr[mid]) {                        // x is on the left side
                        end = mid - 1;
                }
                else {                                          // x is on the right side
                        start = mid + 1;
                }
        }

        return -1;            // Element is not found
}

public static void main(String[] args) {

        int[] input = {3, 4, 5, 6, 7, 8, 9};

        int x = 4;

        System.out.print(binarySearch(input, n, x)); // print index
}
```

We will get the **output**of the above code as: *// Element found at index*

Advantages of Binary search:

- This searching technique is faster and easier to implement.
- Requires no extra space.
- Reduces the time complexity of the program to a greater extent. (The term **time complexity** might be new to you, you will get to understand this when you will be studying algorithmic analysis. For now, just consider it as the time taken by a particular algorithm in its execution, and time complexity is determined by the number of operations that are performed by that algorithm i.e. time complexity is directly proportional to the number of operations in the program).

Sorting

Sorting is a permutation of a list of elements such that the elements are either in increasing ***(ascending)***order or decreasing***(descending)***order.

There are many different sorting techniques. The major difference is the amount of **space** and **time**they consume while being performed in the program.

For now, we will be discussing the following sorting techniques:

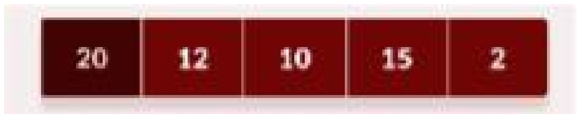
- Selection sort
- Bubble sort
- Insertion sort

Let us now discuss these sorting techniques in detail.

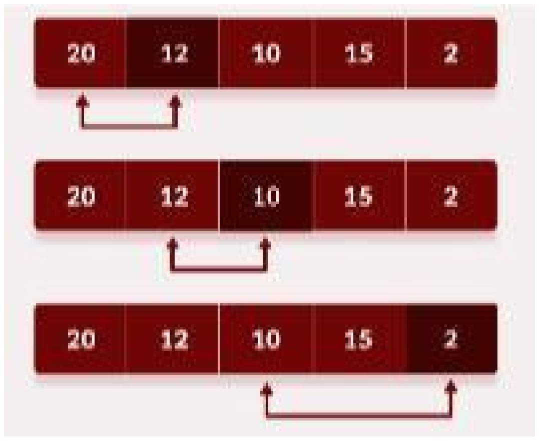
Selection Sort

Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list. The detailed algorithm is given below.

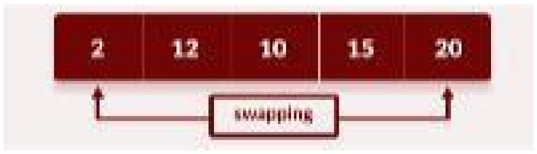
- Consider the given unsorted array to be:



- Set the first element as **minimum**.
- Compare **minimum**with the second element. If the second element is smaller than **minimum**, assign the second element as **minimum**.
- Compare **minimum**with the third element. Again, if the third element is smaller, then assign **minimum**to the third element otherwise do nothing. The process goes on until the last element.

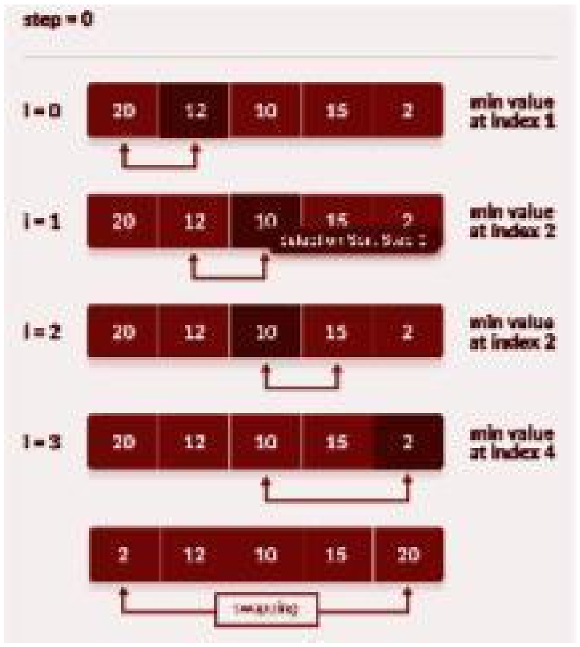


- After each iteration, **minimum**is placed in the front of the unsorted list.

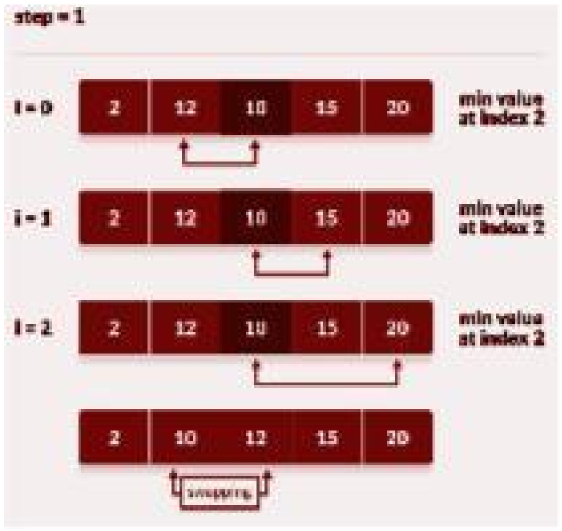


- For each iteration, indexing starts from the first unsorted element. These steps are repeated until all the elements are placed at their correct positions.

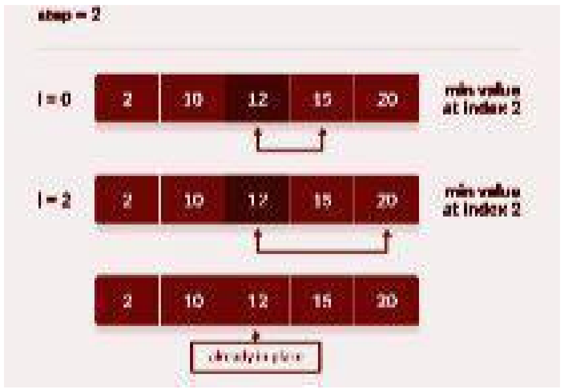
First Iteration



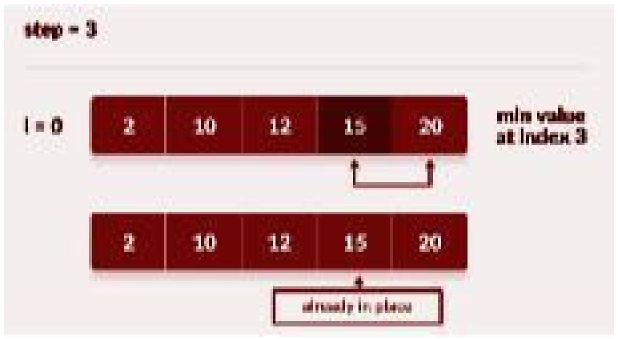
Second Iteration:



Third Iteration



Fourth Iteration



Java Code

```
public static void selectionSort(int input[], int n) { for(int i = 0; i < n-1; i++ ) {
                    // Find min element in the array
                    int min = input[i], minIndex = i;
                    for(int j = i+1; j < n; j++) {
                        // to sort in descending order, change < to > in this // line select the minimum element in
                          each loop
                        if(input[j] < min) {
                              min = input[j];

                              minIndex = j;

                        }

                    }
                    // Swap
                    int temp = input[i];


                    input[minIndex] = temp;

            }

}

public static void main(String[] args) { int input[] =
            {20, 12, 10, 15, 2}; selectionSort(input, 6);

            for(int i = 0; i < 6; i++) { System.out.print(input[i]
                  + " ");

            }

}
```

We will get the **output** of the above code as:

```
 2 10 12 15 20                                        // sorted array
```

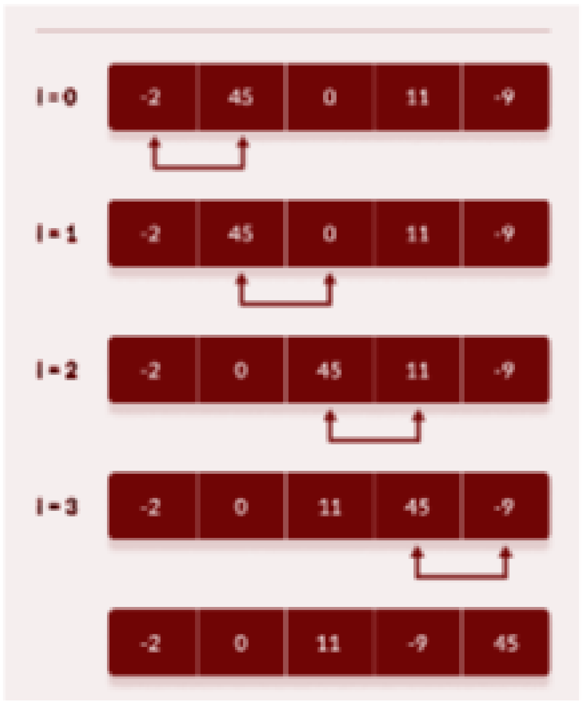
Bubble Sort

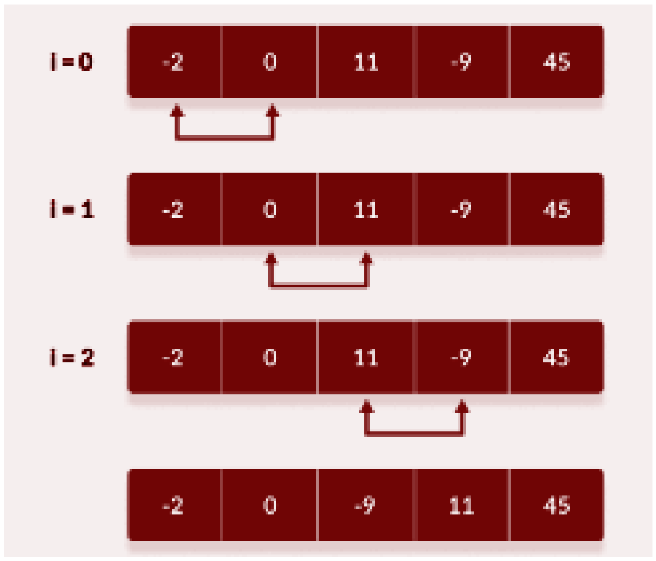
Bubble sort is an algorithm that compares the adjacent elements and swaps their positions if they are not in the intended order. The order can be ascending or descending.

How does Bubble Sort work?

- Starting from the first index, compare the first and the second elements. If the first element is greater than the second element, they are swapped.
- Now, compare the second and third elements. Swap them if they are not in order.
- The above process goes on until the last element.
- The same process goes on for the remaining iterations. After each iteration, the largest element among the unsorted elements is placed at the end.
- In each iteration, the comparison takes place up to the last unsorted element.
- The array is sorted when all the unsorted elements are placed at their correct positions.

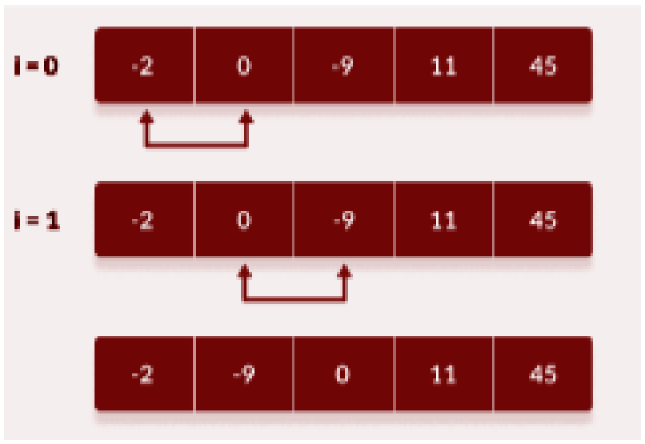
Let the array be [-2, 45, 0, 11, -9

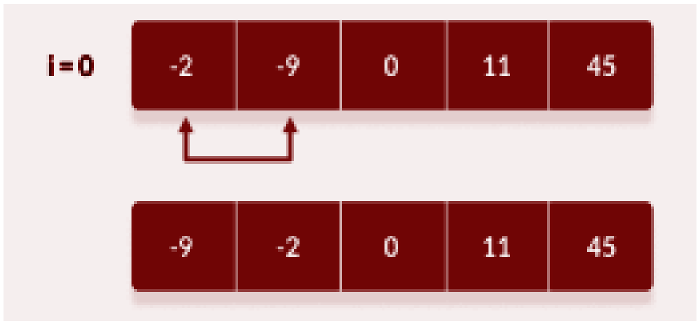




First Iteration

Second Iteration





Third Iteration

Fourth Iteration

Java Code

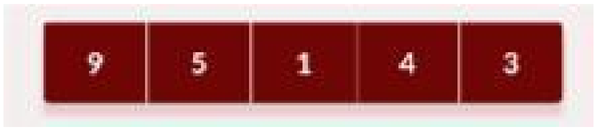
```java
public class BubbleSort {
    public static void bubbleSort(int[] array) {
        int n = array.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                // Swap if the element found is greater than the next
element
                if (array[j] > array[j + 1]) {
                    // Swap array[j] and array[j+1]
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
            }
        }
    }

    public static void printArray(int[] array) {
        int n = array.length;
        for (int i = 0; i < n; ++i) {
            System.out.print(array[i] + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        int[] array = {64, 34, 25, 12, 22, 11, 90};

        System.out.println("Original array:");
        printArray(array);

        bubbleSort(array);

        System.out.println("Sorted array:");
        printArray(array);
    }
}
```

Insertion Sort

- Insertion sort works similarly as we sort cards in our hand in a card game.
- We assume that the first card is already sorted
- Then, we select an unsorted card.
- If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left.
- In the same way, other unsorted cards are taken and put in the right place. A similar approach is used by insertion sort.
- Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration

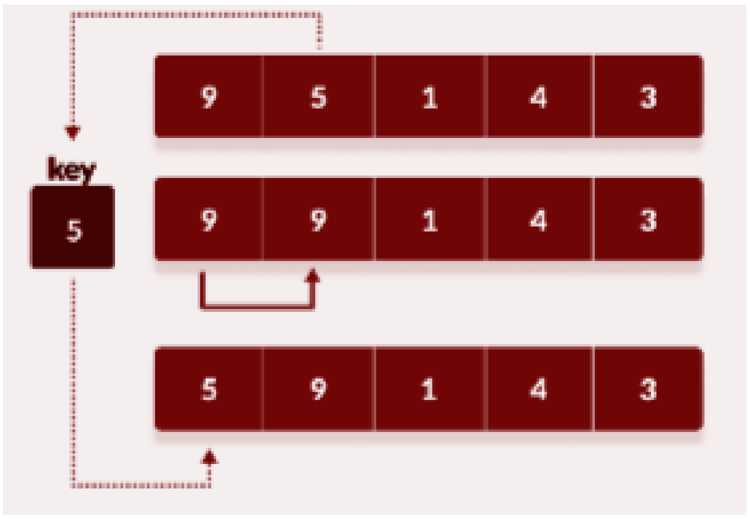
Algorithm

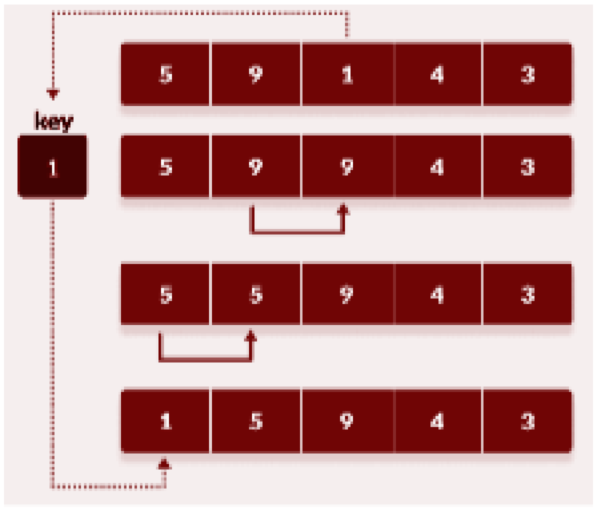
- Suppose we need to sort the following array.



- The first element in the array is assumed to be sorted. Take the second element and store it separately in**key**.
- Compare the key with the first element. If the first element is greater than **key**, then **key**is placed in front of the first element.
- If the first element is greater than **key**, then **key**is placed in front of the first element.
- Now, the first two elements are sorted.
- Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.
- Similarly, place every unsorted element at its correct position.

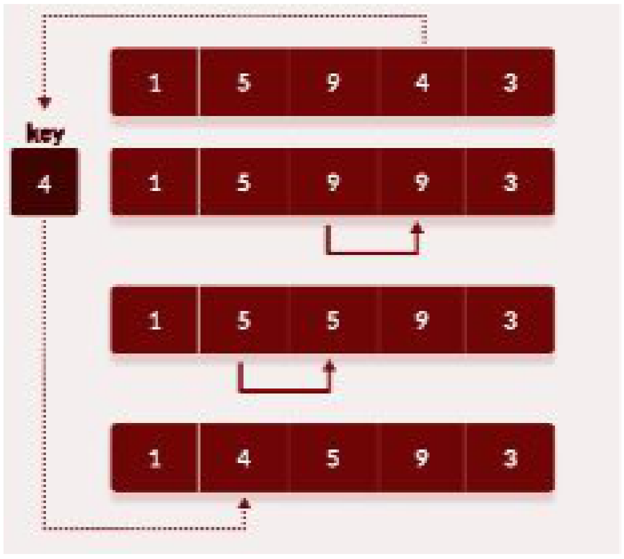
The various iterations are depicted below:

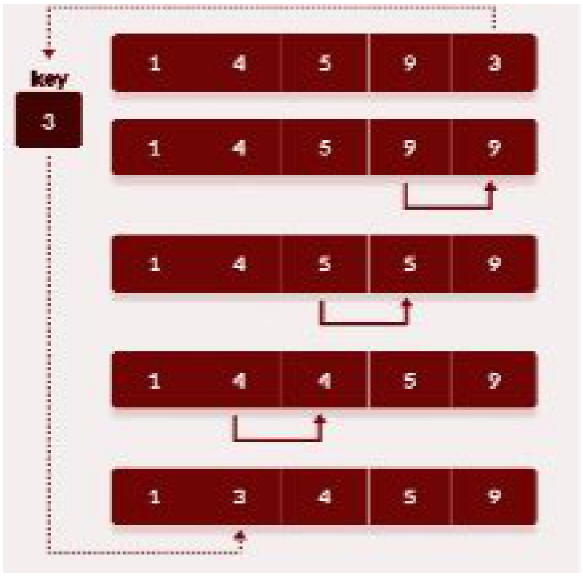




First Iteration

Second Iteration





Third Iteration

Fourth Iteration

Java Code

```java
public class InsertionSort {
    public static void insertionSort(int[] array) {
        int n = array.length;
        for (int i = 1; i < n; ++i) {
            int key = array[i];
            int j = i - 1;

            // Move elements of array[0..i-1] that are greater than
key to one position ahead of their current position
            while (j >= 0 && array[j] > key) {
                array[j + 1] = array[j];
                j = j - 1;
            }
            array[j + 1] = key;
        }
    }

    public static void printArray(int[] array) {
        int n = array.length;
        for (int i = 0; i < n; ++i) {
            System.out.print(array[i] + " ");
        }
        System.out.println();
    }
```

Now, practice different questions to get more familiar with the concepts. In the advanced course, you will study more types of sorting techniques.

Time Complexity

What you will learn in this lecture?

- Algorithm Analysis
- Type of Analysis
- Big O Notation
- Determining Time Complexities Theoretically
- Time complexity of some common algorithms

Introduction

An important question while programming is: How efficient is an algorithm or piece of code?

Efficiency covers lots of resources, including:

1. CPU (time) usage
2. memory usage
3. disk usage
4. network usage

All are important but we are mostly concerned about CPU time. Be careful to differentiate between:

1. **Performance:** how much time/memory/disk/etc. is actually used when a program is run. This depends on the machine, compiler, etc. as well as the code we write.
2. **Complexity:** how do the resource requirements of a program or algorithm scale, i.e. what happens as the size of the problem being solved by the code gets larger. Complexity affects performance but not vice-versa. The time required by a function/method is proportional to the number of "basic operations" that it performs.

Algorithm Analysis

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Why Analysis of Algorithms?

- To predict the behavior of an algorithm without implementing it on a specific computer.
- It is much more convenient to have simple measures for the efficiency of an algorithm than to implement the algorithm and test the efficiency every time a certain parameter in the underlying computer system. changes.
- It is impossible to predict the exact behavior of an algorithm. There are too many influencing factors.
- The analysis is thus only an approximation; it is not perfect.
- More importantly, by analyzing different algorithms, we can compare them to determine the best one for our purpose.

Types of Analysis

To analyze a given algorithm, we need to know, with which inputs the algorithm takes less time (i.e. the algorithm performs well) and with which inputs the algorithm takes a long time.

Three types of analysis are generally performed:

• **Worst-Case Analysis:** The worst-case consists of the input for which the algorithm takes the longest time to complete its execution.

• **Best Case Analysis:** The best case consists of the input for which the algorithm takes the least time to complete its execution.

• **Average case:** The average case gives an idea about the average running time of the given algorithm.

There are two main complexity measures of the efficiency of an algorithm:

- **Time complexity** is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm.
- **Space complexity** is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm.

Big-O notation

We can express algorithmic complexity using the **big-O** notation. For a problem of size N:

- A constant-time function/method is *"order 1"*: **O(1)**
- A linear-time function/method is *"order N"*: **O(N)**
- A quadratic-time function/method is *"order N squared"*: **O(N^2)**

Definition: Let g and f be functions from the set of natural numbers to itself. The function f is said to be **O(g)** (read big-oh of g), if there is a constant **c** and a natural **n_0** such that **$f(n) \leq cg(n)$** for all **$n > n_0$.**

Note: O(g) is a set!

Abuse of notation: f = O(g) does not mean f ∈ O(g).

Examples:

- **$5n^2 + 15 = O(n^2)$**, since **$5n^2 + 15 \leq 6n^2$**, for all **$n > 4$**.
- **5n2 + 15 = $O(n^3)$**, since **$5n^2 + 15 \leq n^3$**, for all **$n > 6$**.
- **O(1)** denotes a constant.

Although we can include constants within the big-O notation, there is no reason to do that. Thus, we can write **O(5n + 4) = O(n)**.

Note: The **big-O** expressions do not have constants or low-order terms. This is because, when N gets large enough, constants and low-order terms don't matter (a constant-time function/method will be faster than a linear-time function/method, which will be faster than a quadratic-time function/method).

Determining Time Complexities Theoretically

In general, how can you determine the running time of a piece of code? The answer is that it depends on what kinds of statements are used.

1. Sequence of statements

```
statement 1;
statement 2;
...
statement k;
```

The total time is found by adding the times for all statements:

```
totalTime = time(statement1) + time(statement2) +..+
time(statementk)
```

2. `if-else` statements

```
if (condition):
    #sequence of statements 1
else:
    #sequence of statements 2
```

Here, either **sequence 1** will execute, or **sequence 2** will execute. Therefore, the worst-case time is the slowest of the two possibilities:

`max(time(sequence 1), time(sequence 2))`

For example, if sequence 1 is O(N) and sequence 2 is O(1) the worst-case time for the whole if-then-else statement would be **O(N)**.

3. `for` loops

```
for i in range N:
    #sequence of
statements
```

Here, the loop executes **N** times, so the sequence of statements also executes **N** times. Now, assume that all the statements are of the order of **O(1)**, then the total time for the **for** loop is **N * O(1)**, which is **O(N)** overall.

4. Nested loops

```
for i in range N:
    for i in range
M:
        #statements
```

The outer loop executes **N** times. Every time the outer loop executes, the inner loop executes **M** times. As a result, the statements in the inner loop execute a total of **N * M** times. Assuming the complexity of the statement inside the inner loop to be **O(1)**, the overall complexity will be **O(N * M).**

Sample Problem:

What will be the Time Complexity of following while loop in terms of 'N' ?

```
while N>0:
    N =
N//8
```

We can write the iterations as:

Iteration Number	**Value of N**
1	N
2	N//8
3	N//64
...	...
k	N//8^k

We know, that in the last i.e. the **kth** iteration, the value of **N** would become **1**, thus, we can write:

```
N//8^k = 1
=> N = 8^k
=> log(N) = log(8^k)
=> k*log(8) =
log(N)
=> k =
log(N)/log(8)
=> k = log_8(N)
```

Now, clearly the number of iterations in this example is coming out to be of the order of **`log₈(N)`**. Thus, the time complexity of the above while loop will be **`O(log₈(N))`**.

Qualitatively, we can say that after every iteration, we divide the given number by 8, and we go on dividing like that, till the number remains greater than 0. This gives the number of iterations as **`O(log₈(N))`**.

Time Complexity Analysis of Some Common Algorithms

Linear Search

Linear Search time complexity analysis is done below-

Best case- In the best possible case:

- The element being searched will be found in the first position.

- In this case, the search terminates in success with just one comparison.
- Thus in the best case, the linear search algorithm takes **O(1)** operations.

Worst Case- In the worst possible case:

- The element being searched may be present in the last position or may not present in the array at all.
- In the former case, the search terminates in success with **N** comparisons.
- In the latter case, the search terminates in failure with **N** comparisons.
- Thus in the worst case, the linear search algorithm takes **O(N)** operations.

Binary Search

Binary Search time complexity analysis is done below-

- In each iteration or each recursive call, the search gets reduced to half of the array.
- So for **N** elements in the array, there are **$\log_2 N$** iterations or recursive calls.

Thus, we have-

- Time Complexity of the Binary Search Algorithm is **$O(\log_2 N)$**.
- Here, **N** is the number of elements in the sorted linear array.

This time complexity of binary search remains unchanged irrespective of the element position even if it is not present in the array.

Big-O Notation Practice Examples

Example-1 Find upper bound for **$f(n) = 3n + 8$**

Solution: $3n + 8 \leq 4n$, for all $n \geq 8$

$\therefore 3n + 8 = O(n)$ with $c = 4$ and $n_0 = 8$

Example-2 Find upper bound for **$f(n) = n^2 + 1$**

Solution: $n^2 + 1 \leq 2n^2$, for all $n \geq 1$

$\therefore$ $n^2 + 1 = O(n^2)$ with $c = 2$ and $n_0 = 1$

Example-3 Find upper bound for **$f(n) = n^4 + 100n^2 + 50$**

Solution: $n^4 + 100n^2 + 50 \leq 2n^4$, for all $n \geq 11$

$\therefore$ $n^4 + 100n^2 + 50 = O(n^4)$ with $c = 2$ and $n_0 = 11$

Recursion-1

Introduction

The process in which a function calls itself is called **recursion** and the corresponding function is called a **recursive function**.

Since computer programming is a fundamental application of mathematics, so let us first try to understand the mathematical reasoning behind recursion.

In general, we all are aware of the concept of functions. In a nutshell, functions are mathematical equations that produce an output on providing input. **For example:** Suppose the function **F(x)** is a function defined by:

$$F(x) = x^2 + 4$$

We can write the **Java Code** for this function as:

```
public static int F(int x){
    return (x * x + 4);
}
```

Now, we can pass different values of x to this function and receive our output accordingly.

Before moving onto the recursion, let's try to understand another mathematical concept known as the **Principle of Mathematical Induction (PMI)**.

Principle of Mathematical Induction (PMI) is a technique for proving a statement, a formula, or a theorem that is asserted about a set of natural numbers. It has the following three steps:

1. **Step of the trivial case:** In this step, we will prove the desired statement for a base case like **n = 0** or **n = 1**.
2. **Step of assumption:** In this step, we will assume that the desired statement is valid for **n = k**.
3. **To prove step:** From the results of the assumption step, we will prove that, **n = k + 1** is also true for the desired equation whenever **n = k** is true.

For Example: Let's prove using the **Principle of Mathematical Induction** that:

```
S(N): 1 + 2 + 3 + ... + N = (N * (N + 1))/2
```

(The sum of first N natural numbers)

Proof:

Step 1: For N = 1, S(1) = 1 is true.

Step 2: Assume, the given statement is true for N = k, i.e.,

```
1 + 2 + 3 + .... + k = (k * (k + 1))/2
```

Step 3: Let's prove the statement for N = k + 1 using step 2.

To Prove: `1 + 2 + 3 + ... + (k+1) = ((k+1)*(k+2))/2`

Proof:

Adding (k+1) to both LHS and RHS in the result obtained on step 2:

```
1 + 2 + 3 + ... + (k+1) = (k*(k+1))/2 + (k+1)
```

Now, taking (k+1) common from RHS side:

```
1 + 2 + 3 + ... + (k+1) = (k+1)*((k + 2)/2)
```

According the statement that we are trying to prove:

```
1 + 2 + 3 + ... + (k+1) = ((k+1)*(k+2))/2
```

Hence proved.

One can think, why are we discussing these over here. To answer this question, we need to know that these three steps of PMI are related to the three steps of recursion, which are as follows:

1. **Induction Step and Induction Hypothesis:** Here, the Induction Step is the main problem which we are trying to solve using recursion, whereas the Induction Hypothesis is the sub-problem, using which we'll solve the induction step. Let's define the Induction Step and Induction Hypothesis for our running example:

   Induction Step: Sum of first n natural numbers - F(n)

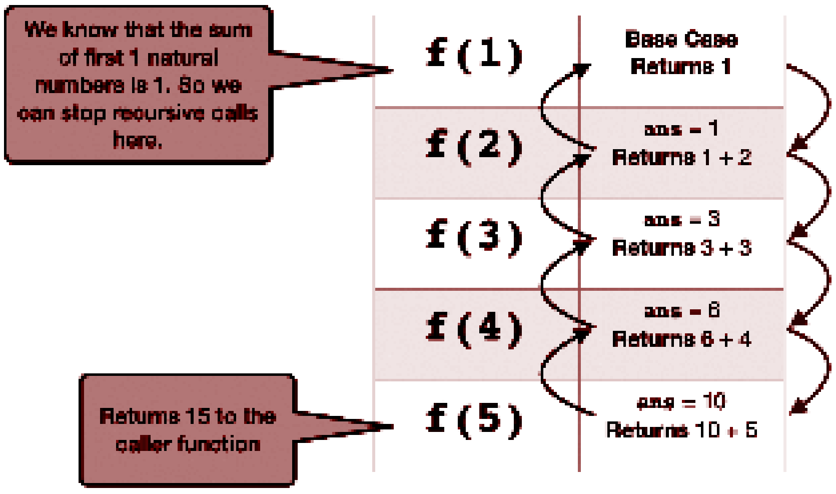
   Induction Hypothesis: This gives us the sum of the first n-1 natural numbers - F(n-1)
2. Express F(n) in terms of F(n-1) and write code:

```
F(N) = F(N-1)+ N
```

   Thus, we can write the Java code as:

```
public static int f(int N){
    int ans = f(N-1); //Induction Hypothesis step
    return ans + N;  //Solving problem from result in previous step
}
```

3. The code is still not complete. The missing part is the base case. Now we will dry run to find the case where the recursion needs to stop.



4. After the dry run, we can conclude that for N equals 1, the answer is 1, which we already know. So we'll use this as our base case. Hence the final code becomes:

```
public static int f(int N){
    if(N == 1)   // Base Case
        return 1;
    int ans = f(N-1);
    return ans + N;
}
```

This is the main idea to solve recursive problems. To summarize, we will always focus on finding the solution to our starting problem and tell the function to compute the rest for us using the particular hypothesis. This idea will be studied in detail in further sections with more examples.

Now, we'll learn more about recursion by solving problems which contain smaller subproblems of the same kind. Recursion in computer science is a method where the solution to the question depends on solutions to smaller instances of the same problem. By the exact nature, it means that the approach that we use to solve the original problem can be used to solve smaller problems as well. So, in other words, in recursion, a function calls itself to solve smaller problems. **Recursion** is a popular approach for solving problems because recursive solutions are generally easier to think than their iterative counterparts, and the code is also shorter and easier to understand.

Working of recursion

We can define the steps of the recursive approach by summarizing the above three steps:

- **Base case:** A recursive function must have a terminating condition at which the process will stop calling itself. Such a case is known as the base case. In the absence of a base case, it will keep calling itself and get stuck in an infinite loop. Soon, the **recursion depth*** will be exceeded and it will throw an error.
- **Recursive call:** The recursive function will invoke itself on a smaller version of the main problem. We need to be careful while writing this step as it is crucial to correctly figure out what your smaller problem is.
- **Small calculation:** Generally, we perform a calculation step in each recursive call. We can achieve this calculation step before or after the recursive call depending upon the nature of the problem.

Note*: Recursion uses an in-built stack which stores recursive calls. Hence, the number of recursive calls must be as small as possible to avoid memory-overflow. If the number of recursion calls exceeded the maximum permissible amount, the **recursion depth*** will be exceeded.
Now, let us see how to solve a few common problems using Recursion.

Problem Statement - Find Factorial of a Number

We want to find out the factorial of a natural number.

Approach: Figuring out the three steps of PMI and then relating the same using recursion.

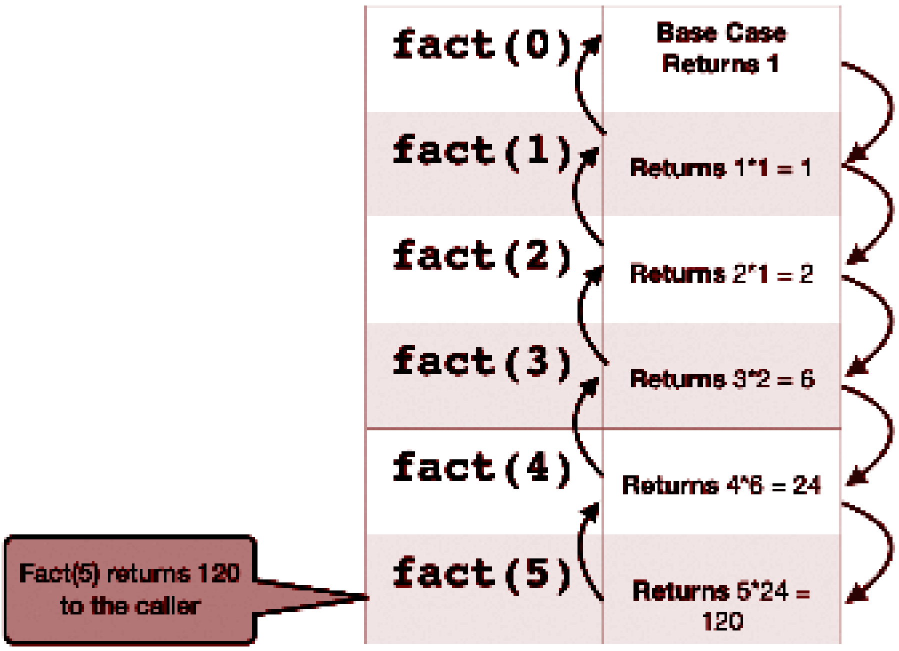
1. **Induction Step:** Calculating the factorial of a number n - **F(n)**

Induction Hypothesis: We have already obtained the factorial of n-1 - **F(n-1)**

2. Expressing F(n) in terms of F(n-1): `F(n)=n*F(n-1)`. Thus we get:

```
public static int fact(int n){
     int ans = fact(n-1); #Assumption step
     return ans * n; #Solving problem from assumption step
}
```

3. The code is still not complete. The missing part is the base case. Now we will dry run to find the case where the recursion needs to stop. Consider **n = 5**:



As we can see above, we already know the answer of n = 0, which is 1. So we will keep this as our base case. Hence, the code now becomes:

```
public static int factorial(int n){
  if (n == 0) // base case
      return 1;
  else
      return n*factorial(n-1); // recursive case
}
```

Problem Statement - Fibonacci Number

Write a function *int fib(int n)* that returns nth fibonacci number. For example, if *n* = 0, then *fib(int n)* should return 0. If n = 1, then it should return 1. For n > 1, it should return F(n-1) + F(n-2), i.e., fibonacci of n-1 + fibonacci of n-2.

Function for Fibonacci series:

F(n) = F(n-1) + F(n-2), F(0) = 0 and F(1) = 1

Approach: Figuring out the three steps of PMI and then relating the same using recursion.

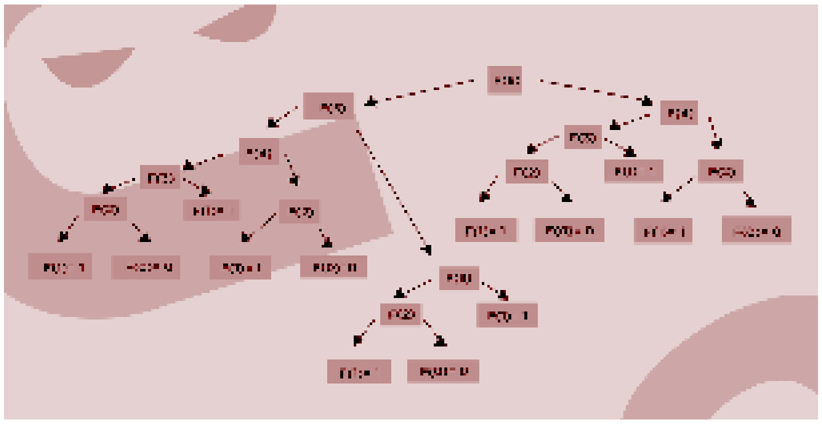
1. **Induction Step:** Calculating the n**th** Fibonacci number n.

   Induction Hypothesis: We have already obtained the (n-1)**th** and (n-2)**th** Fibonacci numbers.

2. Expressing F(n)in terms of F(n-1) and F(n-2): $\mathbf{F_n = F_{n-1} + F_{n-2}}$.

```
public static int f(int n){
     int ans = f(n-1) + f(n-2); //Assumption step
     return ans; //Solving problem from assumption step
}
```

3. Let's dry run the code for achieving the base case: (Consider n= 6)



From here we can see that every recursive call either ends at 0 or 1 for which we already know the answer: **F(0) = 0 and F(1) = 1**. Hence using this as our base case in the code below:

```
public static int fib(int n){
  if (n <= 1)
      return n;
  else
      return (fib(n-1) + fib(n-2));
}
```

Recursion and array

Let us take an example to understand recursion on arrays.

Problem Statement - Check If Array Is Sorted.

We have to tell whether the given array is sorted or not using recursion.

For example:

- If the array is {2, 4, 8, 9, 9, 15}, then the output should be **YES.**
- If the array is {5, 8, 2, 9, 3}, then the output should be **NO.**

Approach: Figuring out the three steps of PMI and then relating the same using recursion.

1. **Induction hypothesis or Assumption step:** We assume that we have already obtained the answer to the array starting from index 1. In other words, we assume that we know whether the array (starting from the first index) is sorted.
2. **Solving the problem from the results of the "Assumption step":** Before going to the assumption step, we must check the relation between the first

two elements. Find if the first two elements are sorted or not. If the elements are not in sorted order, then we can directly return false. If the first two elements are in sorted order, then we will check for the remaining array through recursion.

```
public static int isSorted(int[][] a, int size){
    if (a[0] > a[1]) // Small Calculation
        return false
    int isSmallerSorted = isSorted(a+1, size-1); //Assumption step
    return isSmallerSorted;
}
```

3. We can see that in the case when there is only a single element left or no element left in our array, the array is always sorted. Let's check the final code now:

```
public static int isSorted(int[][] a, int size){
    if (size == 0 or size == 1) // Base case
        return true;

    if (a[0] > a[1]) // Small calculation
        return false;

    int isSmallerSorted = isSorted(a+1, size-1); //Recursive call
    return isSmallerSorted;
}

// driver code
public static void main(String[] args) {
    int arr[] = {2, 3, 6, 10, 11};
    if(isSorted(arr, 5))
        System.out.println("Yes");
    else
        System.out.println("No");
}
```

Problem Statement - First Index of Number

Given an array of length **N** and an integer **x**, you need to find and return the first index of integer **x** present in the array. Return **-1** if it is not present in the array. The first index means that if **x** is present multiple times in the given array, you have to return the index at which **x** comes first in the array.

To get a better understanding of the problem statement, consider the given cases:

Case 1: `Array = {1,4,5,7,2}, Integer = 4`

Output: 1

Explanation: 4 is present at 1st position in the array.

Case 2: `Array = {1,3,5,7,2}, Integer = 4`

Output: -1

Explanation: 4 is not present in the array

Case 3: `Array = {1,3,4,4,4}, Integer = 4`

Output: 2

Explanation: 4 is present at 3 positions in the array; i.e., [2, 3, 4]. But as the question says, we have to find out the first occurrence of the target value, so the answer should be 2.

Approach:

Now, to solve the question, we have to figure out the following three elements of the solution:

1. **Base case**
2. **Recursive call**
3. **Small calculation**

Small calculation part:

Let the array be: [5, 5, 6, 2, 5] and x = 6. Now, if we want to find 6 in the array, then first we have to check with the startIndex which we will increment in each recursive call.

```
if(arr[sI] == x)
    return sI;
```

Recursive Call step:

- Since, in the running example, the startIndex element is not equal to 6, so we will have to make a recursive call for the remaining array: `[5,6,2,5], x=6`. Though we will pass the same array but startIndex will be incremented.
- The recursive call will look like this:

```
f(arr, sI+1, x);
```

- In the recursive call, we are incrementing the startIndex pointer.
- We have to assume that the answer will come from the recursive call. The answer will come in the form of an integer.
- If the answer is -1, this denotes that the element is not present in the remaining array.
- If the answer is any other integer (other than -1), then this denotes that the element is present in the remaining array.

Base case step:

- The base case for this question can be identified by dry running the case when you are trying to find an element that is not present in the array.
- **For example:** Consider the array [5, 5, 6, 2, 5] and x = 10. On dry running, we can conclude that the base case will be the one when the startIndex exceeds size of the array.
- Therefore , then we will return -1. This is because if the base case is reached, then this means that the element is not present in the entire array.
- We can write the base case as:

```
if(sI == arr.length) // Base Case
    return -1;
```

Note: The code written from the above insights can be accessed in the solution tab in the question itself.

Problem Statement - Last Index of Number

Given an array of length **N** and an integer **x**, you need to find and return the first index of integer **x** present in the array. Return **-1** if it is not present in the array. The last index means that if **x** is present multiple times in the given array, you have to return the index at which **x** comes last in the array.

Case 1: `Array = {1,4,5,7,2}, Integer = 4`
Output: 1 (**Explanation:** 4 is present at 1st position in the array, which is the last and the only place where 4 is present in the given array.)

Case 2: `Array = {1,3,5,7,2}, Integer = 4`
Output: -1 (**Explanation:** 4 is not present in the array.)

Case 3: `Array = {1,3,4,4,4}, Integer = 4`
Output: 4 (**Explanation**: 4 is present at 3 positions in the array; i.e., [2, 3, 4], but as the question says, we have to find out the last occurrence of the target value, so the answer should be 4.)

Approach:

Now, to solve the question, we have to figure out the following three elements of the solution.

1. **Base case**
2. **Recursive call**
3. **Small calculation**

Let the array be: [5, 5, 6, 2, 5] and x = 6. Now, if we want to find 6 in the array, then first we have to check with the first index. This is the **small calculation part**.

Code:

```
if(arr[sI] == x)
    return sI;
```

Since, in the running example, the 0th index element is not equal to 6, so we will have to make a recursive call for the remaining array: [5, 6, 2, 5] and x = 6. This is the **recursive call step**. We will start with startIndex from the last index of the array.

The recursive call will look like this:

```
f(arr, sI-1, x);
```

- In the recursive call, we are decrementing the startIndex pointer..
- We have to assume that the answer will come for a recursive call. The answer will come in the form of an integer.

Base Case:

- The base case for this question can be identified by dry running the case when you are trying to find an element that is not present in the array.
- **For example:** [5, 5, 6, 2, 5] and x = 10. On dry running, we can conclude that the base case will be the one when the startIndex passes beyond left of index 0 as we started from the rightmost index.

- When startIndex becomes -1, then we will return -1.
- This is because if the startIndex reaches -1, then this means that we have traversed the entire array and we were not able to find the target element.

Note: The code written from the above insights can be accessed in the solution tab in the question itself.

All Indices of A Number

Here, given an array of length N and an integer x, you need to find all the indexes where x is present in the input array. Save all the indexes in a new array (in increasing order) and return that array.

Case 1: `Array = {1,4,5,7,2}, Integer = 4`
Output: [1], the size of the array will be 1 (as 4 is present at 1st position in the array, which is the only position where 4 is present in the given array).

Case 2: `Array = {1,3,5,7,2}, Integer = 4`
Output: [], the size of the array will be 0 (as 4 is not present in the array).

Case 3: `Array = {1,3,4,4,4}, Integer = 4`
Output: [2, 3, 4], the size of the array will be 3 (as 4 is present at three positions in the array; i.e., [2, 3, 4]).

Now, let’s think about solving this problem…

Approach:

Now, to solve the question, we have to figure out the following three elements of the solution:

1. **Base case**
2. **Recursive call**
3. **Small calculation**

Let us assume the given array is: **[5, 6, 5, 5, 6]** and the target element is **5**, then the output array should be **[0, 2, 3]** and for the same array, let's suppose the target element is **6**, then the output array should be **[1, 4]**.

To solve this question, the base case should be the case when the startIndex reaches the size of the array. In this case, we should simply return an empty array, i.e., an array of size 0, since there are no elements left.

The next two components of the solution are Recursive call and Small calculation. Let us try to figure them out using the following images:

So, the following are the recursive call and small calculation components of the solution:

Recursive Call

```
int[][] output = fun(arr, startIndex+1, size, x);
```

Small Calculation:

1. If the element at startIndex of array is equal to the x, then create a new array of size of output+1. Now copy paste all the elements of the output array to the new array starting from 1st index and at the 0th index add the element of startIndex in original array. Finally return this new output.
2. Else is the case when element at startIndex do not match with x. So simply return the output.

Note: The code written from the above insights can be accessed in the solution tab in the question itself.

Using the same concept, other problems can be solved using recursion, just remember to apply PMI and three steps of recursion intelligently.

Recursion 2

In this module, we are going to understand how to solve different kinds of problems using recursion.

Recursion With Strings

Recursion in strings is not a very different logic, it is the same as we apply in arrays, in fact it becomes more easy to pass a complete new string using substring method.

Problem Statement - Replace pi

You are given a string of size n containing characters a-z. The task is to write a recursive function to replace all occurrences of pi with 3.14 in the given string and print the modified string.

Approach

1. **Step of the trivial case:** In this step, we will prove the desired statement for a base case like **size of string = 0** or **1**.
2. **Small calculation and recursive part interlinked:** In this step, you will first check character at 0th index and character at 1st index of the string.
    - If it comes out to be 'p' and 'i' then we make a recursive call passing the string from index 2. And thereafter "3.14" needs to be concatenated with the recursive answer and this updated string will be the answer.

- Else we will just make a recursive call passing the string from index 1. Thereafter the character at 0th index needs to be concatenated with the recursive answer and return the same.

Binary Search Using Recursion

In a nutshell, this search algorithm takes advantage of a collection of elements that are already sorted by ignoring half of the elements after just one comparison.

You are given a target element X and a sorted array. You need to check if X is present in the array. Consider the algorithm given below:

- Compare X with the middle element in the array.
- **If** X is the same as the middle element, we return the index of the middle element.
- **Else if** X is greater than the mid element, then X can only lie in the right (greater) half subarray after the mid element. Thus, we apply the algorithm, recursively, for the right half. `#Condition1`
- **Else if** X is smaller, the target X must lie in the left (lower) half. So we apply the algorithm, recursively, for the left half. `#Condition2`

```
// Returns the index of x in arr if present, else -1
public static int binary_search(arr, low, high, x){
    if (high >= low){ // Check base case
        mid = (high + low) / 2;
        if (arr[mid] == x) //If element is at the middle itself
            return mid;
        else if (arr[mid] > x)   //Condition 2
            return binary_search(arr, low, mid - 1, x);
        else //Condition 1
            return binary_search(arr, mid + 1, high, x);
    }
    else
        return -1;  // Element is not present in the array
}
```

Sorting Techniques Using Recursion - Merge Sort

Merge sort requires dividing a given list into equal halves until it can no longer be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

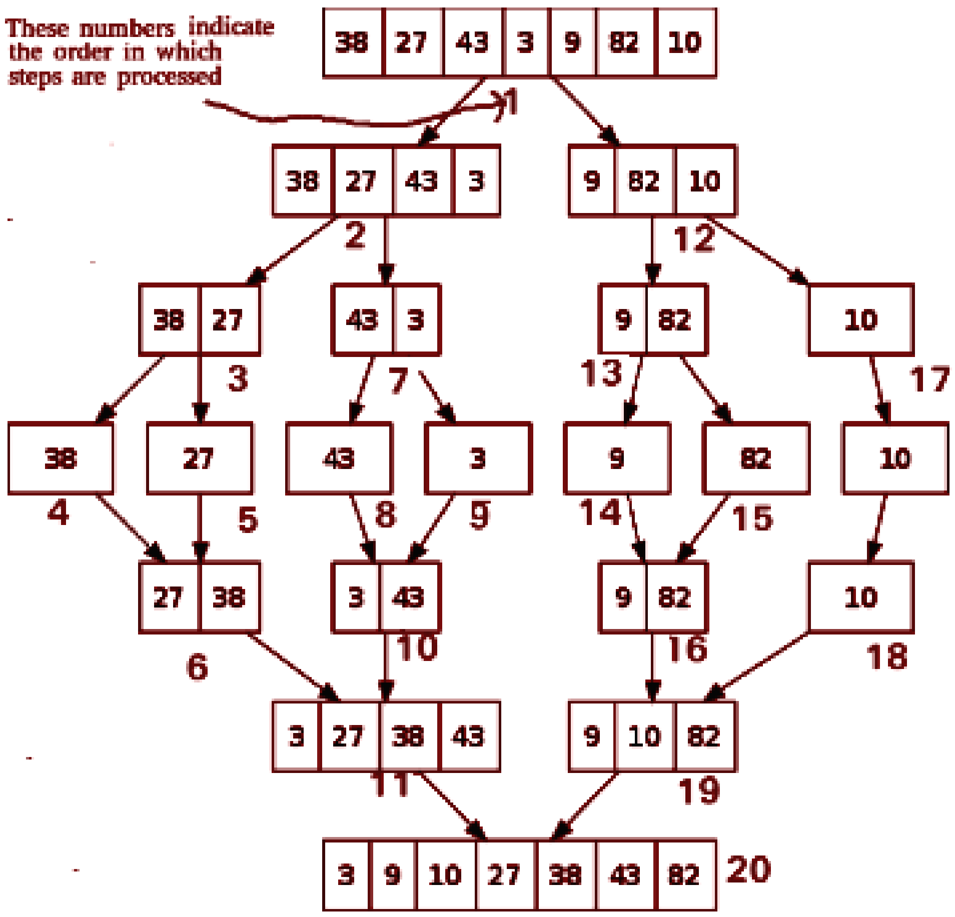
- **Step 1 −** If it is only one element in the list it is already sorted, return.
- **Step 2 −** Divide the list recursively into two halves until it can't be divided further.
- **Step 3 −** Merge the smaller lists into a new list in sorted order.

It has just one disadvantage and that is it's **not an in-place sorting** technique i.e. it creates a copy of the array and then works on that copy.

Pseudo-Code

```
public static void mergeSort(arr[], l,  r){
   if (r > l){
     1. Find the middle point to divide the array into two halves:
          middle m = (l+r)/2
     2. Call mergeSort for the first half:
          Call mergeSort(arr, l, m)
     3. Call mergeSort for the second half:
          Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
          Call merge(arr, l, m, r)
   }
}
```

The following diagram shows the complete merge sort process for an example array `[38,27,43,3,9,82,10]`. If we take a closer look at the diagram, we can see that the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.



Quick Sort

Quick-sort is based on the **divide-and-conquer approach**. It works along the lines of choosing one element as a pivot element and partitioning the array around it such that:

- The left side of the pivot contains all the elements that are less than the pivot element

- The right side contains all elements greater than the pivot.

Algorithm for Quick Sort:

Based on the **Divide-and-Conquer** approach, the quicksort algorithm can be explained as:

- **Divide:** The array is divided into subparts taking pivot as the partitioning point. The elements smaller than the pivot are placed to the left of the pivot and the elements greater than the pivot are placed to the right side.
- **Conquer:** The left and right sub-parts are again partitioned using the by selecting pivot elements for them. This can be achieved by recursively passing the subparts into the algorithm.
- **Combine:** This part does not play a significant role in quicksort. The array is already sorted at the end of the conquer step.

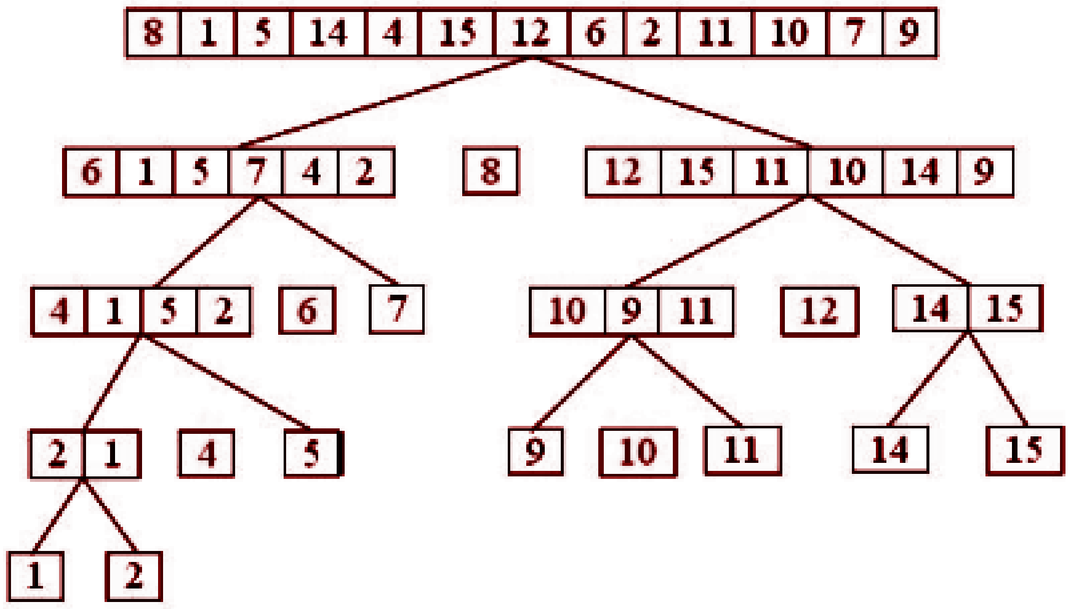
The advantage of quicksort over merge sort is that it does not require any extra space to sort the given array, that it is an in-place sorting technique.

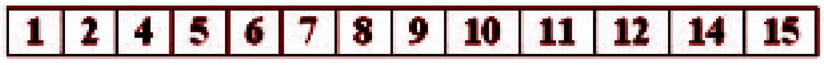
There are many ways to pick a pivot element:

1. Always pick the first element as the pivot.
2. Always pick the last element as the pivot.
3. Pick a random element as the pivot.
4. Pick the middle element as the pivot.

Given below is a pictorial representation of how this algorithm sorts the given array:

```
[8,1,5,14,4,15,12,6,2,11,10,7,9]
```





1	2	4	5	6	7	8	9	10	11	12	14	15

- In **step 1**, 8 is taken as the pivot.
- In **step 2**, 6 and 12 are taken as pivots.
- In **step 3**, 4, and 10 are taken as pivots.
- We keep dividing the list about pivots till there are only 2 elements left in a sublist.

Problem Statement - Tower Of Hanoi

Tower of Hanoi is a **mathematical puzzle** where we have **3** rods and **N** disks. The objective of the puzzle is to move all disks from **source rod** to **destination rod** using a **third rod (say auxiliary)**. The rules are :

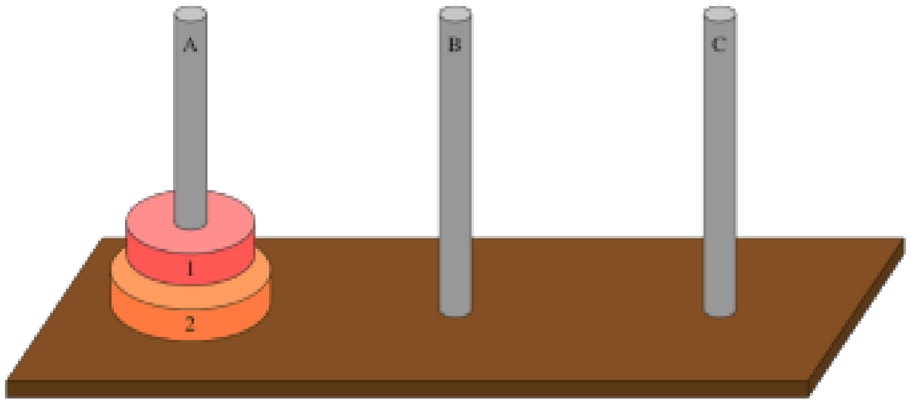
- Only one disk can be moved at a time.
- A disk can be moved only if it is on the top of a rod.
- No disk can be placed on the top of a smaller disk.

Print the steps required to move **N** disks from source rod to destination rod. Source Rod is named as **'A'**, the destination rod as **'B'**, and the auxiliary rod as **'C'**.

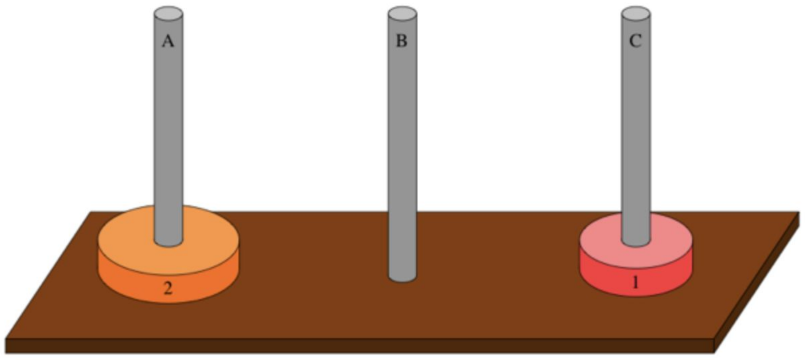
Let's see how to solve the problem recursively. We'll start with a really easy case **N=1**. We just need to move one disk from source to destination.

- You can always move **disk 1** from peg **A** to peg **B** because you know that any disks below it must be larger.
- There's nothing special about pegs **A** and **B**. You can move disk 1 from peg **B** to peg **C** if you like, or from peg **C** to peg **A**, or from any peg to any peg.
- Solving the Towers of Hanoi problem with one disk is trivial as it requires moving only the one disk one time.

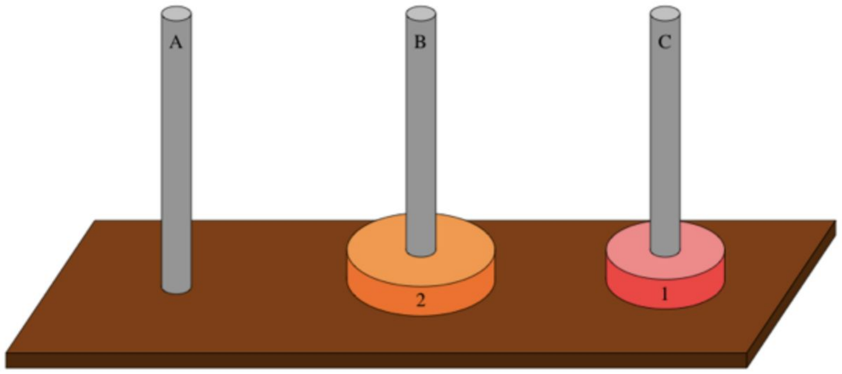
Now consider the case **N=2**. Here's what it looks like at the start:



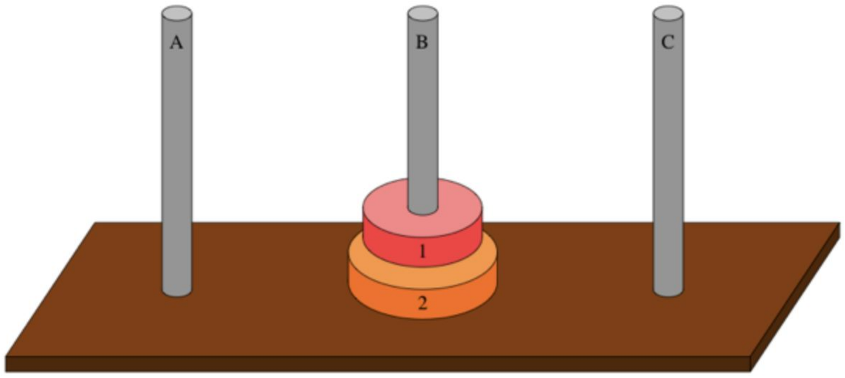
First, move **disk 1** from peg **A** to peg **C**:



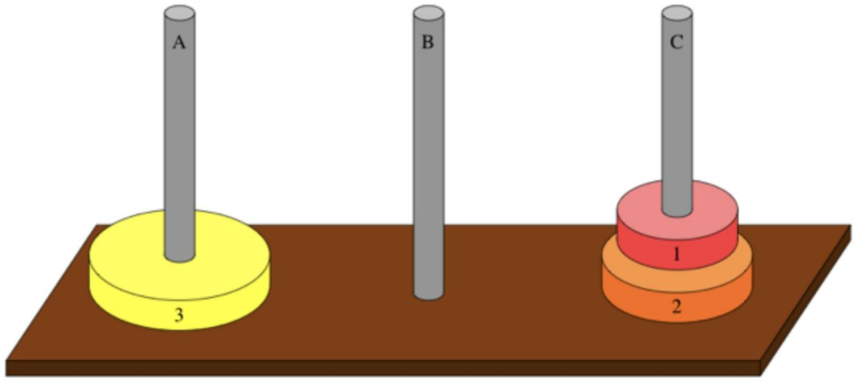
Notice that we're using peg **C** as a spare peg, a place to put **disk 1** so that we can get at **disk 2**. Now that **disk 2**—the bottommost disk—is exposed, move it to peg **B**:



Finally, move **disk 1** from peg **C** to peg **B**:

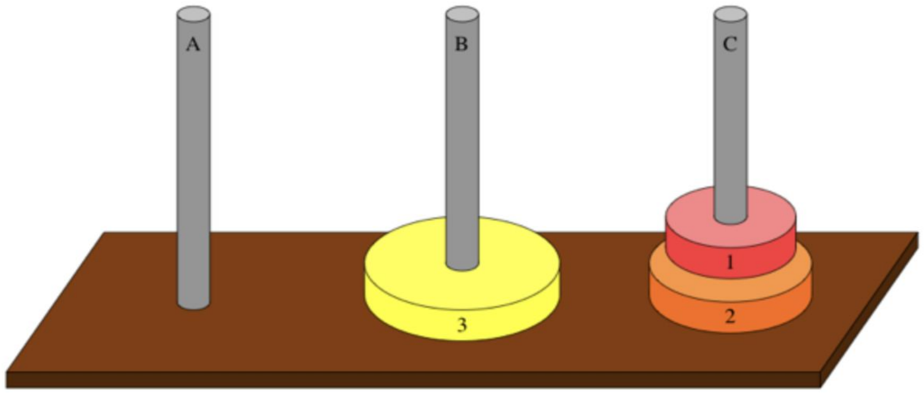


Now let us solve this problem for **3** disks. You need to expose the bottom disk (**disk 3**) so that you could move it from peg **A** to peg **B**. To expose **disk 3**, you needed to move disks 1 and 2 from peg **A** to the spare peg, which is peg **C**:



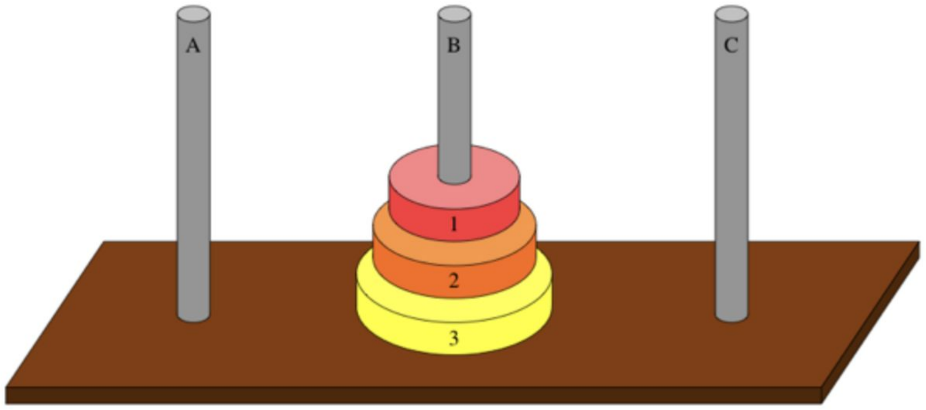
Wait a minute—it looks like two disks moved in one step, violating the first rule. But they did not move in one step. You agreed that you can move disks 1 and 2 from any peg to any peg, **using three steps.** The situation above represents what you have after three steps. (Move **disk 1** from peg **A** to peg **B**; move **disk 2** from peg **A** to peg **C**; move **disk 1** from peg **B** to peg **C**.)

More to the point, by moving disks 1 and 2 from peg **A** to peg **C**, you have recursively solved a subproblem: move disk **1 through n-1** (remember that n = 3)



from peg **A** to peg **C**. Once you've solved this subproblem, you can move **disk 3** from peg **A** to peg **B**:

Now, to finish up, you need to recursively solve the subproblem of moving disks **1 through n-1**, from peg **C** to peg **B**. Again, you agreed that you can do so in three steps. (Move **disk 1** from peg **C** to peg **A**; move **disk 2** from peg **C** to peg **B**; move **disk 1** from peg **A** to peg **B**.) And you're done:



At this point, you might have picked up the pattern. The **algorithm** can be summarised as:

If **n == 1**, just move **disk 1**. Otherwise, when n >= 2, solve the problem in three steps:

- Recursively solve the subproblem of moving disks **1 through n-1** from whichever peg they start on, to the spare peg.
- Move disk **N** from the peg it starts on, to the peg it's supposed to end up on.
- Recursively solve the subproblem of moving disks **1 through n-1**, from the spare peg to the peg they're supposed to end up on.

Java Code

```java
// Recursive Java function to solve the Tower of Hanoi
public static void TowerOfHanoi(int n, int src, int dest, int aux){
    if (n==1){ // Base Case
        System.out.println("Move disk 1 from source",src,"to destination",dest);
        return;
    }
    TowerOfHanoi(n-1, src, aux, dest); // Recursive Call 1
    System.out.println("Move disk",n,"from source",src,"to destination",dest);
    TowerOfHanoi(n-1, aux, dest, src); // Recursive Call 2
}

// driver code
public static void main(String[] args) {
    int n = 4;
    TowerOfHanoi(n,'A','B','C'); // A, B, C are the name of rods
}
```

The output of this code will be:

```
Move disk 1 from source A to destination C
Move disk 2 from source A to destination B
Move disk 1 from source C to destination B
Move disk 3 from source A to destination C
Move disk 1 from source B to destination A
Move disk 2 from source B to destination C
Move disk 1 from source A to destination C
Move disk 4 from source A to destination B
Move disk 1 from source C to destination B
Move disk 2 from source C to destination A
Move disk 1 from source B to destination A
Move disk 3 from source C to destination B
Move disk 1 from source A to destination C
Move disk 2 from source A to destination B
Move disk 1 from source C to destination B
```

Space Complexity Analysis

Introduction

- The space complexity of an algorithm represents the amount of extra memory space needed by the algorithm in its life cycle.
- Space needed by an algorithm is equal to the sum of the following two components:
    - A fixed part is a space required to store certain data and variables (i.e. simple variables and constants, program size, etc.), that are not dependent on the size of the problem.
    - A variable part is a space required by variables, whose size is dependent on the size of the problem. For example, recursion stack space, dynamic memory allocation, etc.
- Space complexity **S(p)** of any algorithm **p** is **S(p) = A + Sp(I)** Where **A** is treated as the fixed part and **S(I)** is treated as the variable part of the algorithm which depends on instance characteristic **I**.

Note: It's necessary to mention that space complexity depends on a variety of things such as the programming language, the compiler, or even the machine running the algorithm.
To get warmed up, let's consider a simple operation that sums two integers (numbers without a fractional part):

```
public static int difference(int a, int b){
    return a + b;
}
```

In this particular method, three variables are used and allocated in memory:

The first integer argument, a; the second integer argument, b; and the returned sum which is also an integer.

In Java, these three variables point to three different memory locations. We can see that the space complexity is constant, so it can be expressed in big-O notation as **O(1)**.

Next, let's determine the space complexity of a program that sums all integer elements in an array:

```
public static int sumArray(int[] array){
    int sum = 0;
    for(int i=0; i<array.length; i++)
        sum += array[i];
    return sum;
}
```

Again, let's list all variables present in the above code:

- **array**
- **size**
- **sum**
- **iterator**

The space complexity of this code snippet is **O(n),** which comes from the reference to the array that was passed to the function as an argument.

Let us now analyze the space complexity for a few common sorting algorithms. This will give you deeper insight into complexity analysis.

Quick-Sort Space Complexity Analysis

Let us consider the various scenarios possible :

Best case scenario: The best-case scenario occurs when the partitions are as evenly balanced as possible, i.e their sizes on either side of the pivot element are either equal or have a size difference of 1 of each other.

- **Case 1:** The case when the sizes of the sublist on either side of the pivot become equal occurs when the subarray has an odd number of elements and the pivot is right in the middle after partitioning. Each partition will have **(n-1)/2** elements.
- **Case 2:** The size difference of 1 between the two sublists on either side of pivot happens if the subarray has an even number, **n**, of elements. One partition will have **n/2** elements with the other having **(n/2)-1**.
- In either of these cases, each partition will have at most **n/2** elements, and the tree representation of the subproblem sizes will be as below:

Worst case scenario:

This happens when we encounter the most unbalanced partitions possible, then the original call takes place **n** times, the recursive call on **n-1** elements will take place **(n-1)** times, the recursive call on **(n-2)** elements will take place **(n-2)** times, and so on.

Based on the above-mentioned cases we can conclude that:

- The space complexity is calculated based on the space used in the recursion stack. The worst-case space used will be **O(n)**.
- The average case space used will be of the order **O(log n).**
- The worst-case space complexity becomes **O(n)** when the algorithm encounters its worst-case when we need to make **n** recursive calls for getting a sorted list.

Practice Problems

Problem 1: What is the time & space complexity of the following code:

```
int a = 0
int b = 0
for(int i=0; i<n; i++){
    a = a + i;
}

for(int j=0; j<m; j++){
    b = b + j;
}
```

Problem 2: What is the time & space complexity of the following code:

```
int a = 0;
int b = 0;
for(int i=0; i<n; i++){
    for(int j=0; j<n; j++){
        a = a + j;
    }
}

for(int k=0; k<n; k++){
    b = b + k
}
```

Problem 3: What is the time and space complexity of the following code:

```
int a = 0;
for(int i=0; i<n; i++){
    int j = n;
    while (j>i){
        a = a + i + j;
        j = j-1;
    }
}
```

Object-Oriented Programming (OOPS-1)

Introduction to OOPS

Object-oriented programming System(OOPs) is a programming paradigm based on the concept of *"objects"* and *"classes"* that contain data and methods. The primary purpose of OOP is to increase the flexibility and maintainability of programs. It is used to structure a software program into simple, reusable pieces of code **blueprints** (called *classes*) which are used to create individual instances of *objects*.

What is an Object?

The object is an entity that has a state and a behavior associated with it. It may be any real-world object like the mouse, keyboard, chair, table, pen, etc.
Objects have states and behaviors. Arrays are objects. You've been using objects all along and may not even realize it. Apart from primitive data types, objects are all around in java.

What is a Class?

A class is a **blueprint** that defines the variables and the methods (Characteristics) common to all objects of a certain kind.

Example: If **Car** is a class, then **Maruti 800** is an object of the **Car** class. All cars share similar features like 4 wheels, 1 steering wheel, windows, breaks etc. Maruti 800 (The **Car** object) has all these features.

Classes vs Objects (Or Instances)

Classes are used to create user-defined data structures. Classes define functions called **methods**, which identify the behaviors and actions that an object created from the class can perform with its data.

In this module, you'll create a **Car** class that stores some information about the characteristics and behaviors that an individual **Car** can have.

A class is a blueprint for how something should be defined. It doesn't contain any data. The **Car** class specifies that a name and a top-speed are necessary for defining a **Car**, but it doesn't contain the name or top-speed of any specific **Car**.

While the class is the blueprint, an instance is an object that is built from a class and contains real data. An instance of the **Car** class is not a blueprint anymore. It's an actual car with a **name**, like Creta, and with a **top speed** of 200 Km/Hr.

Put another way, a class is like a form or questionnaire. An **instance** is like a form that has been filled out with information. Just like many people can fill out the same form with their unique information, many instances can be created from a single class.

Defining a Class in Java

All class definitions start with the `class` keyword, which is followed by the name of the class.

Here is an example of a **Car** class:

```
class Car{

}
```

Note: Java class names are written in *CapitalizedWords* notation by convention. **For example**, a class for a specific model of Car like the Bugatti Veyron would be written as **BugattiVeyron**. The first letter is capitalized. This is just a good programming practice.

The **Car** class isn't very interesting right now, so let's spruce it up a bit by defining some properties that all Car objects should have. There are several properties that we can choose from, including **color**, **brand**, and **top-speed**. To keep things simple, we'll just use **color** and **top-speed**.

Constructor

- Constructors are generally used for instantiating an object.
- The task of a constructor is to initialize(assign values) to the data members of the class when an object of the class is created.
- In Java, constructor for a class must be of the same name as of class.
- In Java, constructors don't have a return type.

Syntax of Constructor Declaration

```
public Car(){
    // body of the constructor
}
```

Types of constructors

- **Default Constructor:** The default constructor is a simple constructor that doesn't accept any arguments.
- **Parameterized Constructor:** A constructor with parameters is known as a parameterized constructor. The parameterized constructor takes its arguments provided by the programmer.

Class Attributes or Data Members

Class attributes are attributes that may or may not have the same value for all class instances. You can define a class attribute by assigning a value to a variable name outside of `constructor.`

For example, the following **Car** class has a class attribute called `name` and `topSpeed` with the value `"Black"`:

```
class Car{
    String name;
    int topSpeed;

    public Car(String carName, int speed){
        name = carName;
        topSpeed = speed;
    }
}
```

- Class attributes are defined directly beneath the first line of the class name.
- When an instance of the class is created, the class attributes are automatically created.

The `this` Parameter

- The `this` parameter is a reference to the current instance of the class and is used to access variables that belong to the class.
- We can use `this` every time but the main use of `this` comes in picture when the attributes and data members share the same name.

Let's update the Car class with the `Car` method that creates `name` and `topSpeed` attributes:

```
class Car{
    String name;
    int topSpeed;

    public Car(String name, int topSpeed){
        this.name = name;
        this.topSpeed = topSpeed;
    }
}
```

In the body of `constructor`, two statements are using the self variable:

1. `this.name = name` assigns the value of the `name` parameter to `name` attribute.
2. `this.topSpeed= topSpeed` assigns the value of the `topSpeed` parameter to `topSpeed` attribute.

All **Car** objects have a `name` and a `topSpeed`, but the values for the `name` and `topSpeed` attributes will vary depending on the **Car** instance. Different objects of the **Car** class will have different names and top speeds.

Now that we have a **Car** class, let's create some cars!

Instantiating an Object in Java

Creating a new object from a class is called instantiating an object. Consider the previous simpler version of our **Car** class:

```
Car c1 = new Car("Creta", 200);
```

You can instantiate a new **Car** object by typing the name of the class, followed by opening and closing parentheses:

Now, instantiate a second **Car** object:

```
Car c2 = new Car("i 10", 190);
```

The new **Car** instance is located at a different memory address. That's because it's an entirely new instance and is completely different from the first **Car** object that you instantiated.

Even though **c1** and **c2** are both instances of the **Car** class, they represent two distinct objects in memory. So they are not equal.

After you create the **Car** instances, you can access their instance attributes using dot notation:

```
System.out.println(c1.name);
System.out.println(c2.topSpeed);
```

Output will be:

```
Creta
190
```

One of the biggest advantages of using classes to organize data is that instances are guaranteed to have the attributes you expect. The values of these attributes ***can*** be changed dynamically:

```
c1.topSpeed= 250
c2.name = "Jeep"
```

In this example, you change the **`topSpeed`** attribute of the **`c1`** object to **250**. Then you change the **`name`** attribute of the **`c2`** object to `"Jeep"`.

Note:

- The key takeaway here is such custom objects are **mutable** by default i.e. their states can be modified.

Access Modifiers

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package

We write the type of modifier before every method or data member.

Final Keyword

If you make any variable final, you cannot change the value of the final variable (It will be constant).

```
class Pen{
    final int price = 15;
}

public class MCQs {
    public static void main(String[] args) {
        Pen p = new Pen();
        p.price = 20;
        System.out.println(p.price);
    }

}
```

There is a final variable price, we are going to change the value of this variable, but it can't be changed because the final variable once assigned a value can never be changed. Therefore this will give a **compile time error**.

Static Keyword

The static variable gets memory only once in the class area at the time of class loading.

The **static keyword** in Java is used for memory management mainly. We can apply static keyword with variables, methods, blocks and nested classes. The static keyword belongs to the class rather than an instance of the class.

```
class Car{
    static int year;
    String company_name;

}

class NewCar{
    public static void main (String[] args) {
        Car c=new Car();
        Car.year=2018;
        c.company_name="KIA";
        Car d=new Car();
        System.out.print(d.year);

    }

}
```

Now in this code, when we look carefully, even when the new instance of Car is created, the year is defined by the first instance of the Car and it tends to remain the same for all instances of the object. But here's the catch, we can change the value of this static variable from any instance. Here the output will be 2018 for every instance as long as it is not changed.

Static Functions

If you apply a static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data members and can change the value of it.

```
class Test{
    static int a = 10;
    static int b;
    static void fun(){
        b = a * 4;
    }
}

class NewCar{
    public static void main(String[] args) {
      Test t=new Test();
      t.fun();
      System.out.print(t.a + t.b);
    }

}
```

When t.fun() is called, it will simply change the value of b to 40.

Therefore the output of this code will be 50.

Object-Oriented Programming (OOPS-2)

What you will learn in this lecture?

- Components of OOPs.
- Access modifiers with inheritance and protected modifiers.
- All about exception handling.

Encapsulation

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is, it is a protective shield that prevents the data from being accessed by the code outside this shield.

- Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of its own class in which they are declared.
- As in encapsulation, the data in a class is hidden from other classes using the data hiding concept which is achieved by making the members or methods of class as private and the class is exposed to the end user or the world without providing any details behind implementation using the abstraction concept, so it is also known as combination of data-hiding and abstraction..

- Encapsulation can be achieved by: Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.

Inheritance

- Inheritance is a powerful feature in Object-Oriented Programming.
- Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance, the information is made manageable in a hierarchical order.
- The class which inherits the properties of the other is known as **subclass** *(derived class or child class)* and the class whose properties are inherited is known as **superclass** *(base class, parent class)*.

Super Keyword:

The super keyword in Java is a reference variable which is used to refer to an immediate parent class object.

Whenever you create an instance of a subclass, an instance of the parent class is created implicitly which is referred to by a super reference variable.

Let us take a real-life example to understand inheritance. Let's assume that **Human** is a class that has properties such as **height**, **weight**, **age**, etc and functionalities (or methods) such as **eating()**, **sleeping()**, **dreaming()**, **working()**, etc.

Now we want to create **Male** and **Female** classes. Both males and females are humans and they share some common properties (like **height**, **weight**, **age**, etc) and behaviors (or functionalities like **eating()**, **sleeping()**, etc), so they can inherit these properties and functionalities from the **Human** class. Both males and females also have some characteristics specific to them (like men have short hair and females have long hair). Such properties can be added to the **Male** and **Female** classes separately.

This approach makes us write less code as both the classes inherited several properties and functions from the superclass, thus we didn't have to re-write them. Also, this makes it easier to read the code.

Java Inheritance Syntax

```
class SuperClass{
    // Body of parent class
}

class SubClass extends SuperClass{
    // Body of derived class
}
```

To inherit properties of the parent class, **extends** keyword is used followed by the name of the parent class.

Example of Inheritance in Java

To demonstrate the use of inheritance, let us take an example.

A polygon is a closed figure with 3 or more sides. Say, we have a class called Polygon defined as follows.

```
class Polygon{
    int n;
    int[] sides;

    public Polygon(int no_of_sides){ //Constructor
        this.n = no_of_sides;
        this.sides = new int[no_of_sides];
    }

    void inputSides(){ //Take user input for side lengths
       Scanner s = new Scanner(System.in);
       for (int i=0; i<this.n; i++){
           System.out.println("Enter side: ");
           this.sides[i] = s.nextInt();
       }
    }

    void displaySides(): //Print the sides of the polygon
        for (int i=0; i<this.n; i++){
            System.out.println("Side " + i+1 +" is" + this.sides[i]);
        }
}
```

This class has **data attributes** to store the number of sides **n** and magnitude of each side as a list called **sides**.

The `inputSides()` method takes in the magnitude of each side and `dispSides()` displays these side lengths.

Now, a triangle is a polygon with 3 sides. So, we can create a class called **Triangle** which inherits from **Polygon**. In other words, we can say that every triangle is a polygon. This makes all the attributes of the **Polygon** class available to the **Triangle** class.

Constructor in Subclass

The constructor of the subclass must call the constructor of the superclass using super keyword:

```
super.Polygon(<Parameter1>,<Parameter2>,...)
```

Note: The parameters being passed in this call must be the same as the parameters being passed in the superclass' constructor/ function, otherwise it will throw an error.

The **Triangle** class can be defined as follows.

```
class Triangle extends Polygon{
    public Triangle(){
        super.Polygon(3); //Calling constructor of superclass
    }

    void findArea(){
        int a = super.sides[0];
        int b = super.sides[1];
        int c = super.sides[2];
        // calculate the semi-perimeter
        int s = (a + b + c) / 2;
        int area = Math.sqrt(s*(s-a)*(s-b)*(s-c));
        print('The area of the triangle is ' + area);
    }
}
```

However, the class **Triangle** has a new method `findArea()` to find and print the area of the triangle. This method is only specific to the **Triangle** class and not **Polygon** class.

Here, even though we did not define methods like `inputSides()` or `displaySides()` for class Triangle separately, we will be able to use them. If an attribute is not found in the subclass itself, the search continues to the superclass.

Access Modifiers

Various object-oriented languages like C++, Java, Python control access modifications which are used to restrict access to the variables and methods of the class. There are four types of access modifiers available in java, which are **Public**, **Private,** and **Protected** in a class, then there is a **default** case (we don't write any keyword in this case), which lies somewhere in between public and private.

Public Modifier

The public access modifier is specified using the keyword public.

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

Consider the given example:

```
// Package 1
public class Student{
    public String name; // public member
    public int age;   // public member

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

---------------------------------------------------------------------

// Package 2
```

```
class Test{
    public static void main(String[] args) {
        Student obj = Student("Boy", 15)
        System.out.println(obj.age); //calling public member of class
        System.out.println(obj.name); //calling public member
    }
}
```

We will get the output as:

```
10
Boy
```

We will be able to access both **name** and **age** of the object from outside the class and package as they are **public**. However, this is not a good practice due to *security concerns.*

Private Modifier

The members of a class that are declared **private** are accessible within the class only. A private access modifier is the most secure access modifier. Data members of a class are declared private by adding a private keyword before the data member of that class. Consider the given example:

```
// Package 1
public class Student{
    private String name; // private member
    public int age;   // public member

     // constructor
    public void Student(String name, int age){
        this.name = name;
        this.age = age;
    }
}
---------------------------------------------------------------------
```

```
// Package 2
class Test{
    public static void main(String[] args) {
        Student obj = Student("Boy", 15)
        System.out.println(obj.age); //calling public member of class
        System.out.println(obj.name); //calling private member
    }
}
```

We will get the output as:

```
10
AttributeError: 'Student' object has no attribute 'name'
```

We will get an **AttributeError** when we try to access the **name** attribute. This is because **name** is a **private** attribute and hence it cannot be accessed from outside the class.

Note: *We can even have **public** and **private** methods.*

Private and Public modifiers with Inheritance

- The subclass will be able to access any **public** method or instance attribute of the superclass.
- The subclass will not be able to access any **private** method or instance attribute of the superclass.

Protected Modifier

The members of a class that are declared protected are only accessible to a class derived from it. Data members of a class are declared **protected** by adding a protected keyword before the data member of that class.

The given example will help you get a better understanding:

```
// superclass
public class Student{
    protected String name; // private member

     // constructor
    public void Student(String name){
        this.name = name;
    }
}
```

This is the parent class **Student** with a **protected** instance attribute **name**. Now consider a subclass of this class:

```
class Display extends Student{
       // constructor
       public Display(String name){
                super.Student(name);
       }
       public void displayDetails(){
                // accessing protected data members of the superclass
                System.out.println("Name: ", super.name);
       }
}

class Test{
    public static void main(String[] args) {
        Display obj = Student("Boy");   // creating objects of the
                                        // derived class
        obj.displayDetails();     // calling public member functions
                                  // of the class
        System.out.println(obj.name);    // trying to access
                                         // protected attribute
    }
}
```

This class **Display** inherits the **Student** class. The method `displayDetails()` accesses the **protected** attribute **_name**. Further, we try to access it again outside this class.

Output:

```
Name: Boy
AttributeError: 'Display' object has no attribute 'name'
```

You can observe that we were able to access the **protected** attribute **_name** from inside the `displayDetails()` method in the subclass. However, we were not able to access it outside the subclass and we got an **AttributeError**. This justifies the definition of the **protected** modifier.

Polymorphism

Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word "poly" means many and "morphs" means forms, So it means many forms.

In Java polymorphism is mainly divided into two types:

- Compile time Polymorphism
- Runtime Polymorphism

1. **Compile-time polymorphism**: It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading. But **Java supports the Operator Overloading with only the '+' symbol**. '+' symbol in java works for adding two integer numbers and it can also be used for string concatenation.

Function/ Method Overloading: When there are multiple functions with the same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments.

Example 1: Polymorphism in addition(+) operator

We know that the **+** operator is used extensively in Java programs. But, it does not have a single usage. For integer data types, the **+** operator is used to perform arithmetic addition operation.

```
int num1 = 1;
int num2 = 2;
System.out.println(num1+num2);
```

Hence, the above program outputs **3**.

Similarly, for string data types, the **+** operator is used to perform concatenation.

```
String str1 = "Java"
String str2 = "Programming"
print(str1+" "+str2)
```

As a result, the above program outputs `"Java Programming"`.

Here, we can see that a single operator + has been used to carry out different operations for distinct data types. This is one of the most simple occurrences of **polymorphism** in Python.

Example 2: Polymorphism with methods/ functions in Java

Let's look at an example.

```
// Java program for Method overloading
```

```
class MultiplyFun {
    // Method with 2 parameter
    static int Multiply(int a, int b){
        return a * b;
    }
    // Method with the same name but 3 parameter
    static int Multiply(int a, int b, int c){
        return a * b * c;
    }
}

class Test{
    public static void main(String[] args) {
        System.out.println(MultiplyFun.Multiply(2, 4));
        System.out.println(MultiplyFun.Multiply(2, 7, 3));
    }
}
```

Output

```
8
42
```

2. **Runtime Polymorphism:** It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding.

   It occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

   Let us see see this in code:

```
// Java program for Method overriding

class Parent {
      void Print() {
            System.out.println("parent class");
      }
}

class subclass1 extends Parent {
      void Print() {
            System.out.println("subclass1");
      }
}

class subclass2 extends Parent {
      void Print() {
            System.out.println("subclass2");
      }
}

class TestPolymorphism3 {
      public static void main(String[] args) {
            Parent a;

            a = new subclass1();
            a.Print();

            a = new subclass2();
            a.Print();
      }
}
```

Output

```
subclass1
subclass2
```

Exception Handling

Error in Java can be of two types i.e. normal unavoidable errors and Exceptions.

- Errors are the problems in a program due to which the program will stop the execution.
- On the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program.

Difference between Syntax Errors and Exceptions

Error: An Error "indicates serious problems that a reasonable application should not try to catch."

Both Errors and Exceptions are the subclasses of java.lang.Throwable class. Errors are the conditions which cannot get recovered by any handling techniques. It surely causes termination of the program abnormally. Errors belong to unchecked type and mostly occur at runtime. Some of the examples of errors are Out of memory error or a System crash error. Also, there are syntax errors that are caused by the wrong syntax in the code. It leads to the termination of the program in compile time itself.

Example:

When you are using recursion to solve any problem, you must have seen errors which say "Stack overflow". In your case, this might have arised due to the incorrect or absence of base case. But this has a deeper explanation. This stack overflow error may also arise when the input is huge and to solve the problem you need too many recursive calls one above the other, this will lead to overflow of the main stack space provided. So there comes the need to solve this problem iteratively. You will practically experience these errors in Dynamic Programming lecture.

For a 64 bits Java 8 program with minimal stack usage, the maximum number of nested method calls is about 7 000. Generally, we don't need more, except in very specific cases. You can

Exceptions: Exceptions are raised when the program is syntactically correct but the code resulted in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.

Example:

```
int marks = 10000;
int a = marks / 0;
System.out.println(a);
```

Output:

```
ZeroDivisionError: division by zero
```

The above example raised the **ZeroDivisionError** exception, as we are trying to divide a number by 0 which is not defined.

Exceptions in Java

- Java has many built-in exceptions that are raised when your program encounters an error (something in the program goes wrong).
- When these exceptions occur, the Java interpreter stops the current process and passes it to the calling process until it is handled.
- If not handled, the program will crash.
- For example, let us consider a program where we have a function A that calls function B, which in turn calls function C. If an exception occurs in function C but is not handled in C, the exception passes to B and then to A.
- If never handled, an error message is displayed and the program comes to a sudden unexpected halt.

Some Common Exceptions

A list of common exceptions that can be thrown from a standard Java program is given below.

- **ArithmeticException**
  It is thrown when an exceptional condition has occurred in an arithmetic operation.

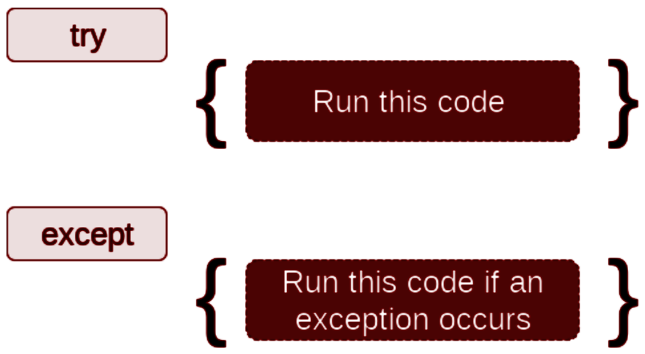
- **ArrayIndexOutOfBoundsException**
  It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
- **ClassNotFoundException**
  This Exception is raised when we try to access a class whose definition is not found
- **FileNotFoundException**
  This Exception is raised when a file is not accessible or does not open.
- **IOException**
  It is thrown when an input-output operation failed or interrupted
- **InterruptedException**
  It is thrown when a thread is waiting , sleeping , or doing some processing, and it is interrupted.
- **NoSuchFieldException**
  It is thrown when a class does not contain the field (or variable) specified
- **NoSuchMethodException**
  It is thrown when accessing a method which is not found.
- **NullPointerException**
  This exception is raised when referring to the members of a null object. Null represents nothing

- **NumberFormatException**
  This exception is raised when a method could not convert a string into a numeric format.
- **RuntimeException**
  This represents any exception which occurs during runtime.
- **StringIndexOutOfBoundsException**
  It is thrown by String class methods to indicate that an index is either negative than the size of the string

Catching Exceptions

In Java, exceptions can be handled using `try-catch` blocks.

- If the Java program contains suspicious code that may throw the exception, we must place that code in the `try` block.
- The `try` block must be followed by the `catch` statement, which contains a block of code that will be executed in case there is some exception in the `try` block.
- We can thus choose what operations to perform once we have caught the exception.



- Here is a simple example:

```
int[] arr = {1, 0, 2};
for (int ele : arr){
    try{ //This block might raise an exception while executing
        System.out.println("The entry is" + ele);
        int r = 1/int(ele);
    }
    catch(Exception e) { //This block executes in case of an
                          // exception in "try"
        System.out.println("Oops! An error occurred: "+e.toString());
    }
    System.out.println();
}
```

We get the output to this code as:

```
The entry is 1

The entry is 0
Oops! An error occurred: java.lang.ArithmeticException: / by zero

The entry is 2
```

- In this program, we loop through the values of an array arr.
- As previously mentioned, the portion that can cause an exception is placed inside the try block.
- If no exception occurs, the catch block is skipped and normal flow continues.
- But if any exception occurs, it is caught by the catch block (second value of the array).
- Here, we print the name of the exception using the `e.toString()` function.
- We can see that element 0 causes ZeroDivisionError.

Every exception in Java inherits from the base **Exception** class.

Catching Specific Exceptions in Java

- In the above example, we did not mention any specific exception in the `catch` clause.
- This is not a good programming practice as it will catch all exceptions and handle every case in the same way.
- We can specify which exceptions a `catch` clause should catch.
- A try clause can have any number of `catch` clauses to handle different exceptions, however, only one will be executed in case an exception occurs.
- You can use multiple `catch` blocks for different types of exceptions.

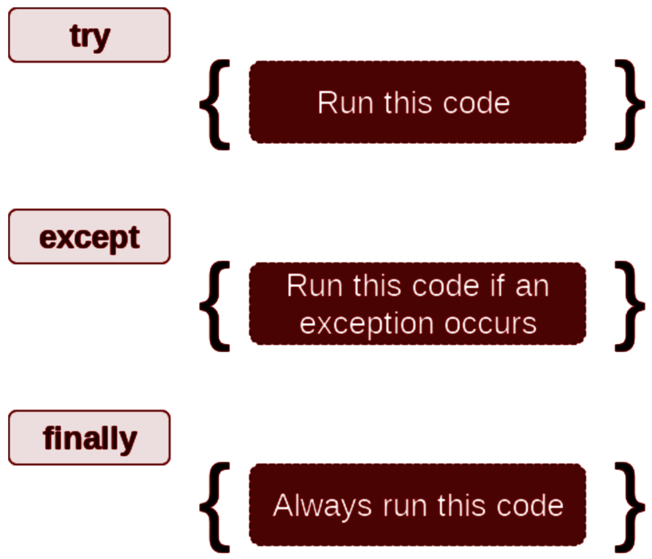
Here is an example to understand this better:

```
try{
    a=10/0;
}
catch(ArithmeticError e){
    System.out.println("Arithmetic Exception");
}
catch(IOException e){
    System.out.println("input output Exception");
}
```

Output:

```
Arithmetic Exception
```

`finally` Statement



The `try` statement in Java can have an optional `finally` clause. This clause is executed no matter what and is generally used to release external resources. Here is an example of file read and close to illustrate this:

```
FileReader f = null;
try{
    f = new FileReader(file);
    BufferedReader br = new BufferedReader(f);
    String line = null;
}
catch (FileNotFoundException fnf) {
    fnf.printStackTrace();
}
finally {
    if( f != null)
        f.close();
}
```

This type of construct makes sure that the file is closed even if an exception occurs during the program execution.

Object-Oriented Programming (OOPS-3)

What you will learn in this lecture?

- Important keywords and their use.
- Abstraction.
- Interfaces.

Final Keyword

- When a variable is declared with a final keyword, its value can't be modified, essentially, a constant. This also means that you must initialize a final variable.
- If the final variable is a reference, this means that the variable cannot be re-bound to reference another object, but the internal state of the object pointed by that reference variable can be changed i.e. you can add or remove elements from the final array or final list.
- Final keywords can be used to initialise constants.

Initializing a final variable:

```
final int {name_of_variable} = {value};
```

Example:

```
final int pi = 3.14;
```

Refer to the course videos to see the use case and more about the final keyword.

Abstract Classes

An abstract class can be considered as a blueprint for other classes. Abstract classes are classes that contain one or more abstract methods. An abstract method is a method that has a declaration but does not have an implementation. This set of methods must be created within any child classes which inherit from the abstract class. *A class that contains one or more abstract methods is called an **abstract class**.*

Creating Abstract Classes in Java

- By default, Java does not provide abstract classes.
- A method becomes abstract when decorated with the keyword **`abstract`**.
- An abstract class cannot be directly instantiated i.e. we cannot create an object of the abstract class.
- However, the subclasses of an abstract class that have definitions for all the abstract methods declared in the abstract class, can be instantiated.
- While declaring abstract methods in the class, it is not mandatory to use the **`abstract`** decorator (i.e it would not throw an exception). However, it is considered a good practice to use it as it notifies the compiler that the user has defined an abstract method.

The given Java code uses the **ABC** class and defines an abstract base class:

```
 abstract class ABC{
    int value;
    Abstract int do_something(){ //Our abstract method declaration
        // TO_DO
    }
}
```

We will do it in the following example, in which we define two classes inheriting from our abstract class:

```
class add extends ABC{
    int do_something(){
        return value + 42;
    }
}

class mul extends ABC{
    int do_something(){
        return value * 42;
    }
}

class Test{
    public static void main(String[] args) {
        add x = new add(10);
        mul y = new mul(10);

        System.out.println(x.do_something());
        System.out.println(y.do_something());
    }
}
```

We get the output as:

```
52
420
```

Thus, we can observe that a class that is derived from an abstract class cannot be instantiated unless all of its abstract methods are overridden.

Note: Concrete classes contain only concrete (normal) methods whereas abstract classes may contain both concrete methods and abstract methods.

- An abstract method can have an implementation in the abstract class.

- However, even if they are implemented, this implementation shall be overridden in the subclasses.
- If you wish to invoke the method definition from the abstract superclass, the abstract method can be invoked with `super()` call mechanism. (*Similar to cases of "normal" inheritance*).
- Similarly, we can even have concrete methods in the abstract class that can be invoked using `super()` call. Since these methods are not abstract it is not necessary to provide their implementation in the subclasses.
- Consider the given example:

```
abstract class ABC{

    abstract int do_something(){ //Abstract Method
        System.out.println("Abstract Class AbstractMethod");
    }

    int do_something2(){ //Concrete Method
        System.out.println("Abstract Class ConcreteMethod");
    }
}

class AnotherSubclass extends ABC{
    int do_something(){
        //Invoking the Abstract method from super class
        super().do_something();
    }

    //No concrete method implementation in subclass
}

class Test{
    public static void main(String[] args) {
        AnotherSubclass x = new AnotherSubclass()
        x.do_something() //calling abstract method
        x.do_something2() //Calling concrete method
    }
}
```

We will get the output as:

```
Abstract Class AbstractMethod
Abstract Class ConcreteMethod
```

Another Example

The given code shows another implementation of an abstract class.

```
// Java program showing how an abstract class works
abstract class Animal{ //Abstract Class
    abstract move();
}

class Human extends Animal{ //Subclass 1
    void move(){
        System.out.println("I can walk and run");
    }
}

class Snake extends Animal{ //Subclass 2
    void move(){
        System.out.println("I can crawl")
}

class Dog extends Animal{ //Subclass 3
    void move(){
        System.out.println("I can bark")
    }
}

// Driver code
```

```
class Test{
    public static void main(String[] args) {
        Animal R = new Human();
        R.move();
        Animal K = Snake();
        K.move();
        R = Dog();
        R.move();
    }
}
```

We will get the output as:

```
I can walk and run
I can crawl
I can bark
```

Interfaces

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

Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.

An interface is different from a class in several ways:

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.

- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Declaring Interface

```
public interface Name_of_interface {
      // body
}
```

Example:

```
public interface VehicleInterface {

      public final static double PI = 3.14;

      public int getMaxSpeed();
      public void print();
}
```

Now we need to implement this interface using a different class. A class uses the implements keyword to implement an interface. The **implements** keyword appears in the class declaration following the extends portion of the declaration.

```
public class Vehicle implements CarInterface{

      @Override
      public void print() {
            // TODO Auto-generated method stub
            // We can implement this function further.
      }

      @Override
      public int getMaxSpeed() {
```

```
		// TODO Auto-generated method stub
		return 0;
	}

	@Override
	public String getCompany() {
		// TODO Auto-generated method stub
		return null;
	}
}
```

@**Override** annotation informs the compiler that the element is meant to **override** an element declared in an interface.

We can implement the given overridden functions and instantiate an object of Vehicle class.

OOPs - 4

Introduction

In this lecture we will try to create game logics by playing around different classes. We will create a Tic-Tac-Toe game and we will discuss Othello game. It will be a java console application. You can refer to course videos to understand the game rules.

Tic-Tac-Toe

In this game, two players will be played and you have one print board on the screen where from 1 to 9 numbers will be displayed or you can say it box number. Now, you have to choose X or O for the specific box number. For example, if you have to select any number then for X or O will be shown on the print board, and turn for next will be there. The task is to create a Java program to implement a 3×3 Tic-Tac-Toe game for two players.

How to Play the Game :

- Both the players choose either **X** or **O** to mark their cells.
- There will be a 3×3 grid with numbers assigned to each of the 9 cells.
- The player who chose **X** begins to play first.
- He enters the cell number where he wishes to place **X**.
- Now, both **O** and **X** play alternatively until any one of the two wins.
- **Winning criteria:** Whenever any of the two players has fully filled one row/ column/ diagonal with his symbol (X/ O), he wins and the game ends.

- If neither of the two players wins, the game is said to have ended in a **draw**.

Let us now code this game

Board class

```
public class Board {
      private char board[][];
      private int boardSize = 3;
      private char p1Symbol, p2Symbol;
      private int count;
      public final static int PLAYER_1_WINS = 1;
      public final static int PLAYER_2_WINS = 2;
      public final static int DRAW = 3;
      public final static int INCOMPLETE = 4;
      public final static int INVALID = 5;



      public Board(char p1Symbol, char p2Symbol){
            board = new char[boardSize][boardSize];
            for(int i =0; i < boardSize; i++){
                  for(int j =0; j < boardSize; j++){
                        board[i][j] = ' ';
                  }
            }
            this.p1Symbol = p1Symbol;
            this.p2Symbol = p2Symbol;
      }
      public int move(char symbol, int x, int y) {

            if(x < 0 || x >= boardSize || y < 0 || y >= boardSize ||
                                                      board[x][y] != ' '){
                  return INVALID;
            }

            board[x][y] = symbol;
            count++;
            // Check Row
            if(board[x][0] == board[x][1] && board[x][0] == board[x][2]){
```

```
				return symbol == p1Symbol ? PLAYER_1_WINS :PLAYER_2_WINS;
			}
			// Check Col
			if(board[0][y] == board[1][y] && board[0][y] == board[2][y]){
				return symbol == p1Symbol ? PLAYER_1_WINS :PLAYER_2_WINS;
			}
			// First Diagonal
			if(board[0][0] != ' ' && board[0][0] == board[1][1] &&
								board[0][0] == board[2][2]){
				return symbol == p1Symbol ? PLAYER_1_WINS :PLAYER_2_WINS;
			}
			// Second Diagonal
			if(board[0][2] != ' ' && board[0][2] == board[1][1] &&
								board[0][2] == board[2][0]){
				return symbol == p1Symbol ? PLAYER_1_WINS :PLAYER_2_WINS;
			}
			if(count == boardSize * boardSize){
				return DRAW;
			}
			return  INCOMPLETE;

	}
	public void print() {
		System.out.println("---------------");
		for(int i =0; i < boardSize; i++){
			for(int j =0; j < boardSize; j++){
				System.out.print("| " + board[i][j] + " |");
			}
			System.out.println();
		}
		System.out.println();
		System.out.println("---------------");
	}
}
```

Player class

```
public class Player {

      private String name;
      private char symbol;

      public Player(String name, char symbol){
            setName(name);
            setSymbol(symbol);
      }

      public void setName(String name) {

            if(!name.isEmpty()) {
                  this.name = name;
            }
      }

      public String getName() {
            return this.name;
      }

      public void setSymbol(char symbol) {
            if(symbol != '\0') {
                  this.symbol = symbol;
            }
      }

      public char getSymbol() {
            return this.symbol;
      }
}
```

TicTacToe class (Main class)

```
import java.util.Scanner;

public class TicTacToe {
      private Player player1, player2;
      private Board board;

      public static void main(String args[]){
            TicTacToe t = new TicTacToe();
            t.startGame();
      }

      public void startGame(){
            Scanner s = new Scanner(System.in);
            // Players input
            player1 = takePlayerInput(1);
            player2 = takePlayerInput(2);
            while(player1.getSymbol() == player2.getSymbol()){
                  System.out.println("Symbol Already taken !! Pick another
                                                                  symbol !!");
                  char symbol = s.next().charAt(0);
                  player2.setSymbol(symbol);
            }
            // Create Board
            board = new Board(player1.getSymbol(), player2.getSymbol());
            // Conduct the Game
            boolean player1Turn = true;
            int status = Board.INCOMPLETE;
            while(status == Board.INCOMPLETE || status == Board.INVALID){
                  if(player1Turn){
                        System.out.println("Player 1 - " +
                                                    player1.getName() + "'s turn");
                        System.out.println("Enter x: ");
                        int x = s.nextInt();
                        System.out.println("Enter y: ");
                        int y = s.nextInt();
                         status =  board.move(player1.getSymbol(), x, y);
                        if(status != Board.INVALID){
                              player1Turn = false;
                              board.print();
                        }else{
                              System.out.println("Invalid Move!Try Again");
                        }
                  }else{
                        System.out.println("Player 2 - " +
                                          player2.getName() + "'s turn");
                        System.out.println("Enter x: ");
                        int x = s.nextInt();
                        System.out.println("Enter y: ");
                        int y = s.nextInt();
                        status = board.move(player2.getSymbol(), x, y);
                        if(status != Board.INVALID){
                              player1Turn = true;
                              board.print();
                        }else{
                             System.out.println("Invalid Move! Try Again");
                        }
                  }
            }

            if(status == Board.PLAYER_1_WINS){
                  System.out.println("Player 1 - " + player1.getName() +"
                                                               wins !!");
            }else if(status == Board.PLAYER_2_WINS){
                  System.out.println("Player 2 - " + player2.getName() +"
                                                               wins !!");
            }else{
                  System.out.println("Draw !!");
            }
      }

      private Player takePlayerInput(int num){
            Scanner s = new Scanner(System.in);
            System.out.println("Enter Player " + num + " name: ");
            String name = s.nextLine();
            System.out.println("Enter Player " + num + " symbol: ");
            char symbol = s.next().charAt(0);
            Player p = new Player(name, symbol);
            return p;
      }
}
```

Othello

Refer to course videos for better understanding of the game.
Othello is a board game and you are expected to implement the move function for this game.

Approach: We have eight different directions to explore before we make a move (before we make changes to the board) to ensure which all boxes will toggle. We will move in every direction one by one and check for the player's value who just made his turn. We will then toggle the desired boxes, which the current player secured by playing in that position. To explore all eight directions we can create arrays for different directions and looping in the board we can increment or decrement in respective value to move in the particular direction, like we explore ways in backtracking lecture for rat in a maze game.

Now this code can be written on your own. Refer to the solution tab for the solution.

Linked-List 1

Data Structures

Data structures are just a way to store and organize our data so that it can be used and retrieved as per our requirements. Using these can affect the efficiency of our algorithms to a greater extent. There are many data structures that we will be going through throughout the course, linked-lists are a part of them.

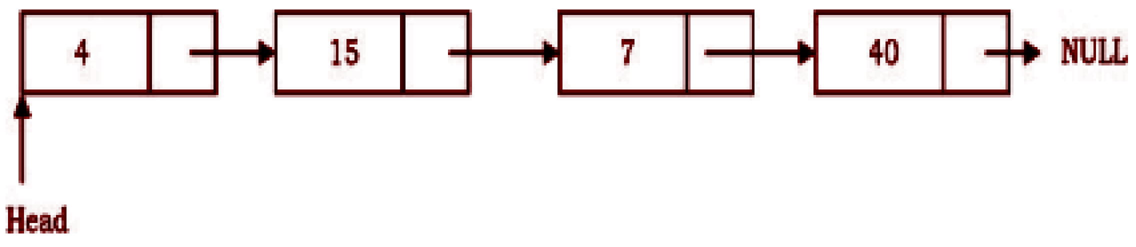
Introduction

A **Linked List** is a data structure used for storing collections of data. A linked list has the following properties:

- Successive elements are connected by pointers.
- Can grow or shrink in size during the execution of a program.
- Can be made just as long as required (until systems memory exhausts).
- Does not waste memory space (but takes some extra memory for pointers). It allocates memory as the list grows.

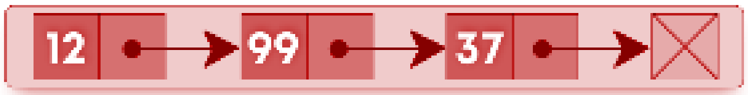
Basic Properties:

- Each element or node of a list is comprising of two items:
    - Data
    - Pointer(reference) to the next node.
- In a Linked List, the elements are not stored at contiguous memory locations.
- The first node of a linked list is known as **Head**.
- The last node of a linked list is known as **Tail.**
- The last node has a reference to null.

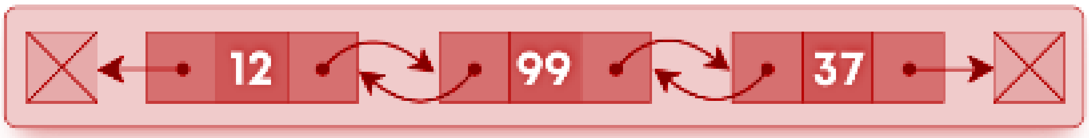


Types of A Linked List

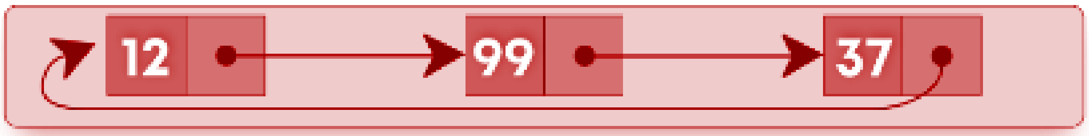
- **Singly-Linked List:** Generally "linked list" means a singly linked list. Each node contains only one link which points to the subsequent node in the list.



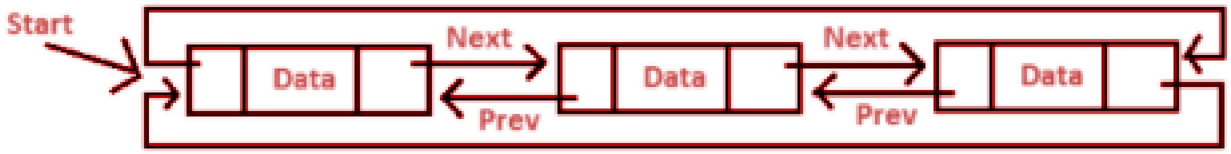
- **Doubly-Linked List:** It's a two-way linked list as each node points not only to the next pointer but also to the previous pointer.



- **Circular-Linked List:** There is no tail node i.e., the next field is never **null** and the next field for the last node points to the head node.



- **Circular Doubly-Linked List:** Combination of both Doubly linked list and circular linked list.



Node Class (Singly Linked List)

```
// Node class
class Node{
    int data;
    Node next;
    // Function to initialize the node object
    Node(int data){
        this.data = data;  // Data that the node contains
        this.next = null; // Next node that this node points to
    }
}
```

<u>Note</u>: The first node in the linked list is known as **Head** pointer and the last node is referenced as **Tail** pointer. We must never lose the address of the head pointer as it references the starting address of the linked list and if lost, would lead to loss of the list.

Traversing the Linked List

Let us assume that the head points to the first node of the list. To traverse the list we do the following:

- Follow the pointers.
- Display the contents of the nodes (or count) as they are traversed.
- Stop when the next pointer points to **null**.

Printing the Linked List

To print the linked list, we will start traversing the list from the beginning of the list(head) until we reach the **null** pointer which will always be the tail pointer. Let us add a new function `printList()` to our **LinkedList** class.

```
// This function prints contents of linked list starting from head
public static void printList(Node headNode){
    Node temp = headNode; //Start from the head of the list
    while (temp != null){ //Till we reach the last node
        System.out.print(temp.data + “ ”);
        temp = temp.next; //Update temp to point to the next Node
    }
}
```

Insertion of A Node in a Singly Linked List

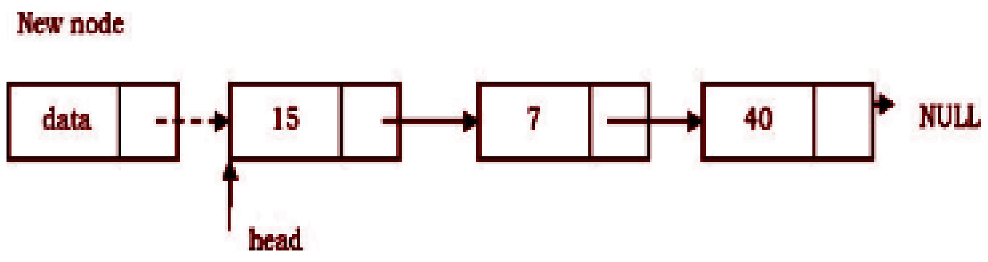
There are 3 possible cases:

- Inserting a new node before the head (at the beginning).
- Inserting a new node after the tail (at the end of the list).
- Inserting a new node in the middle of the list (random location).

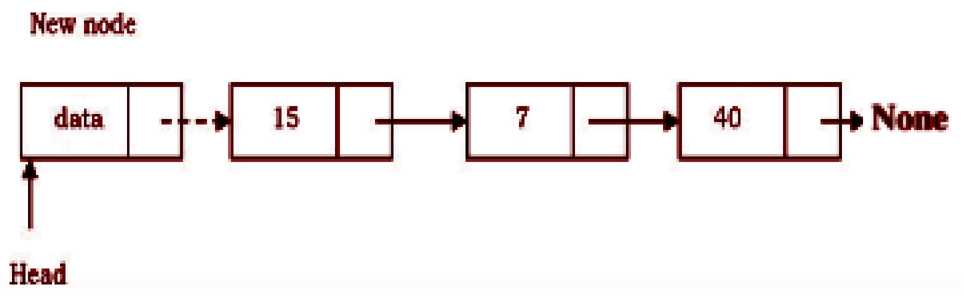
Case 1: Insert node at the beginning:

In this case, a new node is inserted before the current head node. Only one next pointer needs to be modified (new node's next pointer) and it can be done in two steps:

• Create a new node. Update the **next** pointer of the new node, to point to the current head.



• Update **head** pointer to point to the new node.



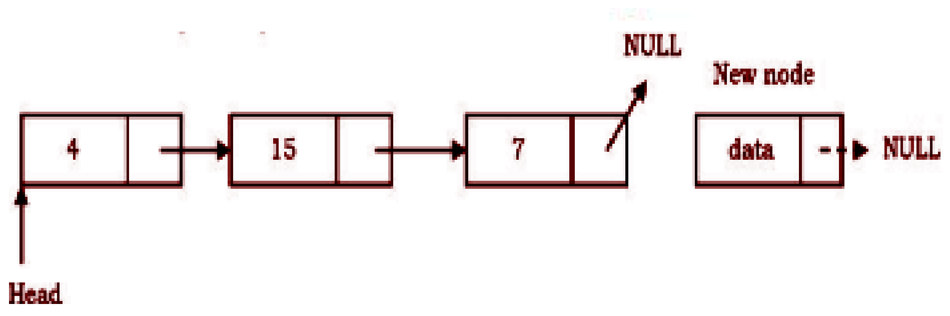
Java Code:

```java
public static Node insertAtStart(Node head, int data){
    Node newNode = new Node(data); //Create a new node
    newNode.next = head; //Set next node of new node to current head
    head = newNode; //Update the head pointer to the new node
    return head;
}
```

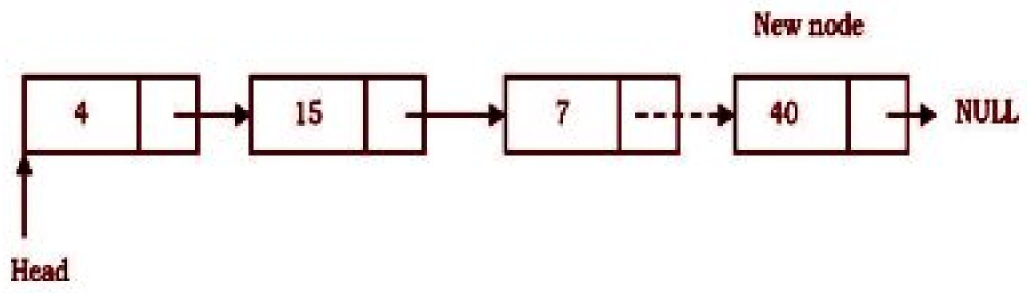
Case 2: Insert node at the ending:

In this case, we need to modify two next pointers (last nodes next pointer and new nodes next pointer).

• New node's **next** pointer points to **null**.



• Last node's **next** pointer points to the new node.



Java Code:

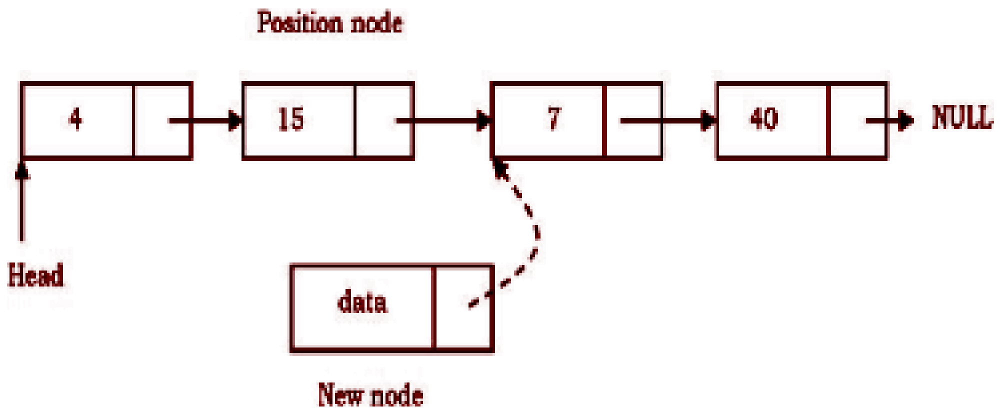
```
public static Node insertAtEnd(Node head, int data){
        Node newNode = new Node(data); //Create a new node
        if (head == null){ //Incase of empty LL
            head = newNode;
            return;
        }
        Node n = head;
        while (n.next != null) //If not empty traverse till last node
            n = n.next;
        n.next = newNode; //Set next = new node
        return head;
}
```

Case 3: Insert node anywhere in the middle: (At any specified Index)

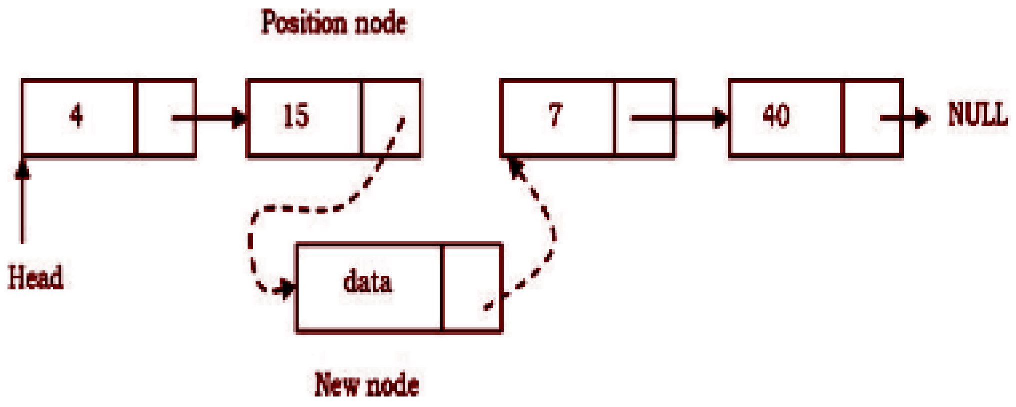
Let us assume that we are given a position where we want to insert the new node. In this case, also, we need to modify two next pointers.

• If we want to add an element at position 3 then we stop at position 2. That means we traverse 2 nodes and insert the new node.

• For simplicity let us assume that the second node is called the **position node**. The new node points to the next node of the position where we want to add this node.



• Position node's **next** pointer now points to the new node.



Java Code:

```
public static Node insertAtIndex (int head, int index, int data){
        if (index == 1)  // Insert at beginning
            insertAtStart(head, data);
        int i = 1;
        Node n = head;
        while (i < index-1 && n != null){
            n = n.next;
            i = i+1;
        }
        if (n == null)
            print("Index out of bound");
        else{
            Node newNode = new Node(data);
            newNode.next = n.next;
            n.next = newNode;
        }
}
```

Deletion of A Node in a Singly Linked List

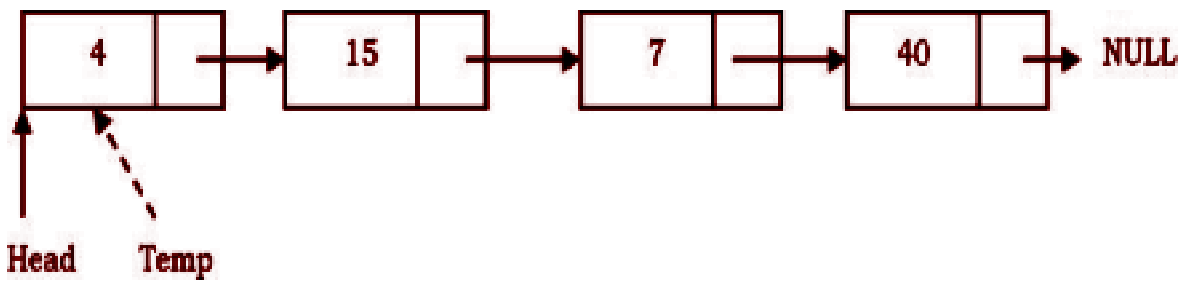
Similar to insertion, here we also have three cases.

- Deleting the first node
- Deleting the last node
- Deleting an intermediate node.

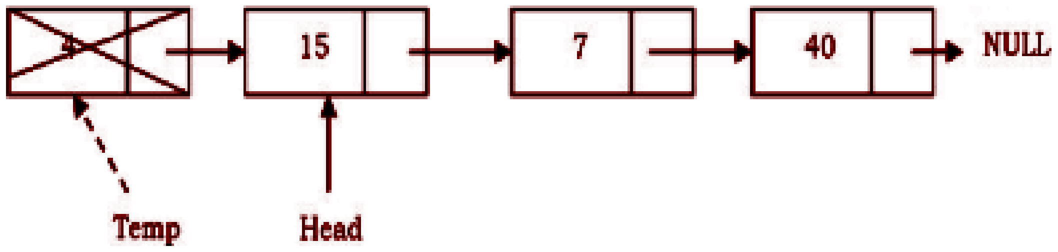
Deleting the First Node in Singly Linked List

It can be done in two steps:

• Create a temporary node which will point to the same node as that of the head.



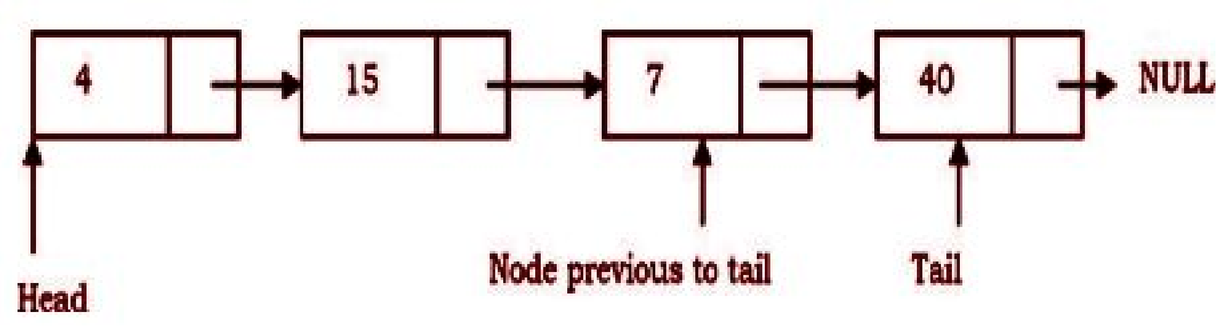
• Now, move the head nodes pointer to the next node and dispose of the temporary node.



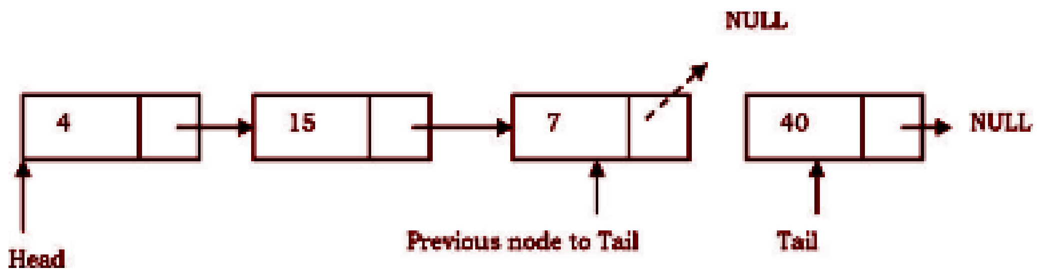
Deleting the Last Node in Singly Linked List

In this case, the last node is removed from the list. This operation is a bit trickier than removing the first node because the algorithm should find a node, which is previous to the tail. It can be done in three steps:

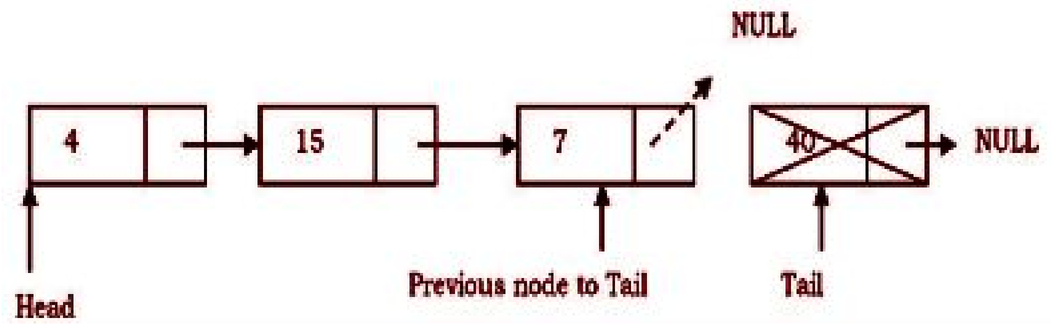
• Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of the list, we will have two pointers, one pointing to the tail node and the other pointing to the node before the tail node.



• Update the previous node's next pointer with **null**.



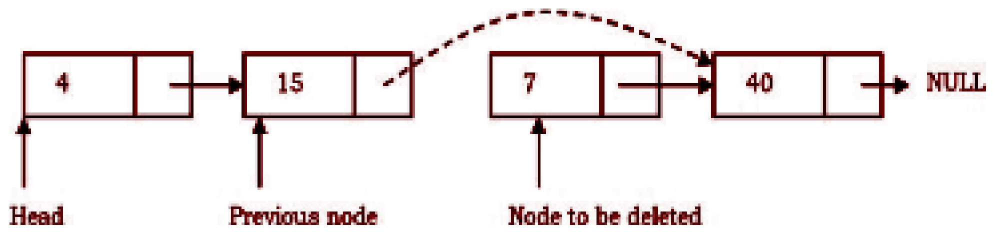
• Dispose of the tail node.



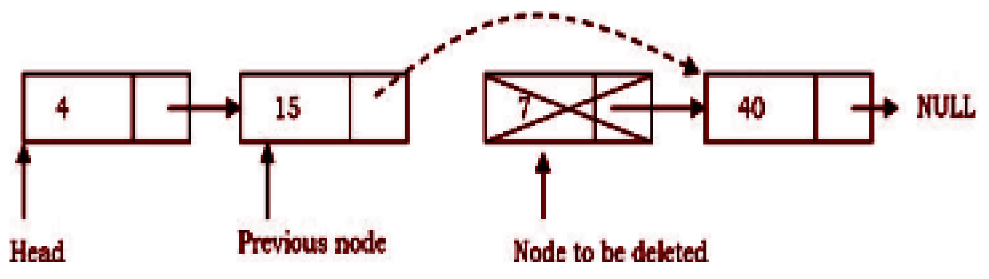
Deleting an Intermediate Node in Singly Linked List

In this case, the node to be removed is always located between two nodes. Head and tail links are not updated in this case. Such removal can be done in two steps:

• Similar to the previous case, maintain the previous node while traversing the list. Once we find the node to be deleted, change the previous node's next pointer to the next pointer of the node to be deleted.



• Dispose of the current node to be deleted.



Insert node recursively

Follow the steps below and try to implement it yourselves:

- If **Head** is **null** and **position** is not 0. Then exit it.
- If **Head** is **null** and **position** is 0. Then insert a new Node to the **Head** and exit it.
- If **Head** is not **null** and **position** is 0. Then the **Head** reference set to the new Node. Finally, a new Node set to the Head and exit it.
- If not, iterate until finding the Nth position or **end**.

For the code, refer to the Solution section of the problem.

Delete node recursively

Follow the steps below and try to implement it yourselves:

- If the node to be deleted is the **root**, simply delete it.
- To delete a middle node, we must have a pointer to the node previous to the node to be deleted. So if the position is not zero, we run a loop position-1 times and get a pointer to the previous node.
- Now, simply point the previous node's next to the current node's next and delete the current node.

For the code, refer to the Solution section of the problem.

Linked-List 2

Now moving further with the topic, let's try to solve some problems now...

The Midpoint of A Linked List

The Trivial Approach- Two Passes

- This approach requires us to traverse through the linked list twice i.e. 2 passes.
- In the first pass, we will calculate the **length** of the linked list. After every iteration, update the **length** variable.
- In the second pass, we will find the element of the linked list at the **(length-1)/2th** position. This element shall be the middle element in the linked list.
- However, we wish to traverse the linked list only once, therefore let us see another approach.

The Optimal Approach- One Pass

- The midpoint of a linked list can be found out very easily by taking two pointers, one named **slow** and the other named **fast**.
- As their names suggest, they will move in the same way respectively.
- The **fast** pointer will move ahead **two pointers at a time**, while the **slow** pointer one will move at a speed of **a pointer at a time**.
- In this way, when the fast pointer will reach the end, by that time the slow pointer will be at the middle position of the array.
- These pointers will be updated like this:
    - `slow = slow.next`
    - `fast = fast.next.next`

Java Code

```
public static Node returnMiddle(Node headNode){
    if (headNode == null || headNode.next == null)
        return head;
    Node slow = headNode; //Slow pointer
    Node fast = headNode.next; //Fast Pointer
    while (fast != null && fast.next != null){
        fast = fast.next.next;
        slow = slow.next;
    }
    return slow; // Slow pointer shall point to our middle element
}
```

Note:

- For odd length there will be only one middle element, but for the even length there will be two middle elements.
- In case of an even length LL, both these approaches will return the first middle element and the other one will be the direct **next** of the first middle element.

Merge Two sorted linked lists

- We will be merging the linked list, similar to the way we performed merge over two sorted arrays.
- We will be using the two **head** pointers, compare their data and the one found smaller will be directed to the new linked list, and increase the **head** pointer of the corresponding linked list.
- Just remember to maintain the **head** pointer separately for the new sorted list.
- And also if one of the linked list's length ends and the other one's not, then the remaining linked list will directly be appended to the final list.
- Try to implement this approach on your own.

Mergesort over a linked list

- Like the merge sort algorithm is applied over the arrays, the same way we will be applying it over the linked list.
- Just the difference is that in the case of arrays, the middle element could be easily figured out, but here you have to find the middle element, each time you send the linked list to split into two halves using the above approach.
- The merging part of the divided lists can also be done using the **merge sorted linked lists code** as discussed above.
- The functionalities of this code have already been implemented by you, just use them directly in your functions at the specified places.
- Try to implement this approach on your own.

Reverse the linked list

Recursive approach:

- In this approach, we will store the last element of the list in the small answer, and then update that by adding the next last node and so on.
- Finally, when we will be reaching the first element, we will assign the **next** to **null**.
- Follow the Java code below, for better understanding.

```java
public static Node reverseLinkedList(Node head){
    if (headNode == null || headNode.next == null)
        return head;

    Node smallHead = reverseLinkedList(head.next);
    Node tail = smallHead;
    while(tail.next != null){
        tail = tail.next;
    }
    tail.next = head;
    head.next = null;
    return smallHead;
}
```

After calculation, you can see that this code has a time complexity of **O(n²)**. Now let’s think about how to improve it.

Recursive approach (Optimal):

- There is another recursive approach to the order of **O(n)**.
- What we will be doing is that head but also the tail pointer, which can save our time in searching over the list to figure out the tail pointer for appending or removing.
- Check out the given code for your reference:

```
class Pair{
    Node head;
    Node tail;
    public Pair(Node head, Node tail){
        this.head = head;
        this.tail = tail;
    }
}

class main{
    private static Pair reverse2Helper(Node head){
         if (head == null || head.next == null)
              return new Pair(head, head);

         Pair p = reverse2Helper(head.next);
         p.tail.next= head;
         head.next = null;
         return new Pair(p.head,head);
    }

    private static Node reverse2(Node head){
         return reverse2Helper(head).head;
    }
    // Main driver function can be written by yourself
}
```

Now let us try to improve this code further.

A simple observation is that the **tail** is always **head.next**. By making the recursive call we can directly use this as our **tail** pointer and reverse the linked list by **tail.next = head**. Refer to the code below, for better understanding.

```
public static Node reverse3(Node head){
      if (head == null || head.next == null)
           return head;

      smallHead = reverse3(head.next);
      tail = head.next;
      tail.next = head;
      head.next = null;
      return smallHead;
}
```

Iterative approach:

- We will be using three-pointers in this approach: **previous, current,** and **next.**
- Initially, the **previous** pointer would be **null** as in the reversed linked list, we want the original head to be the last element pointing to **null** .
- The **current** pointer will point to the current node whose **next** will be pointing to the previous element but before pointing it to the previous element, we need to store the next element's address somewhere otherwise we will lose that element.
- Similarly, iteratively, we will keep updating the pointers as **current** to the **next**, **previous** to the **current,** and **next** to **current**'s **next**.

Refer to the given Java code for better understanding:

```
public static Node reverse(Node head){
     if (head == null || head.next == null)
         return head;

    Node prev = null;
    Node curr = head.next;
    Node next = curr.next;

    while (next != null){
        curr.next = prev;
        prev = curr;
        curr = next;
        next = next.next;
    }
    curr.next = prev;
    return curr;
}
```