U20CSCJ05 JAVA PROGRAMMING

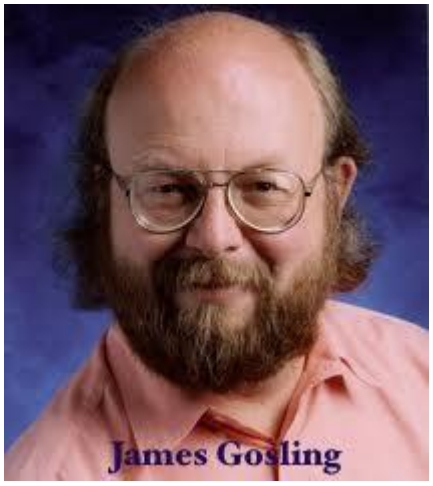
MODULE I

History of Java

The history of Java is very interesting. Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time. The history of Java starts with the Green Team. Java team members (also known as Green Team), initiated this project to develop a language for digital devices such as set-top boxes, televisions, etc. However, it was best suited for internet programming. Later, Java technology was incorporated by Netscape.

The principles for creating Java programming were "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted, and Dynamic".

Java was developed by James Gosling, who is known as the father of Java, in 1995. James Gosling and his team members started the project in the early '90s.



Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc. Following are given significant points that describe the history of Java.

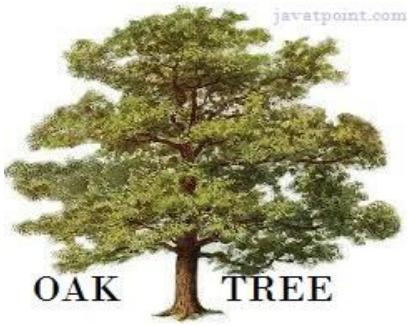
1) **James Gosling , Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.

2) Initially it was designed for small, embedded systems in electronic appliances like set-top boxes.

3) Firstly, it was called **"Greentalk"** by James Gosling, and the file extension was .gt.

4) After that, it was called **Oak** and was developed as a part of the Green project.

Why Java was named as "Oak"?



5) **Why Oak?** Oak is a symbol of strength and chosen as a national tree of many countries like the U.S.A., France, Germany, Romania, etc.

6) In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.

Why Java Programming named "Java"?

7) Why had they chose the name Java for Java language? The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA", etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell, and fun to say.

According to James Gosling, "Java was one of the top choices along with **Silk**". Since Java was so unique, most of the team members preferred Java than other names.

8) Java is an island in Indonesia where the first coffee was produced (called Java coffee). It is a kind of espresso bean. Java name was chosen by James Gosling while having a cup of coffee nearby his office.

9) Notice that Java is just a name, not an acronym.

10) Initially developed by James Gosling at Sun Microsystems

(which is now a subsidiary of Oracle Corporation) and released in 1995.

11) In 1995, Time magazine called **Java one of the Ten Best Products of 1995**.

12) JDK 1.0 was released on January 23, 1996. After the first release of Java, there have been many additional features added to the language. Now Java is being used in Windows applications, Web applications, enterprise applications, mobile applications, cards, etc. Each new version adds new features in Java.

Java Version History

Many java versions have been released till now. The current stable release of Java is Java SE 10.

Java Virtual Machine

JVM (Java Virtual Machine) Architecture

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

What is JVM

It is:

1. **A specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Oracle and other companies.
2. **An implementation** Its implementation is known as JRE (Java Runtime Environment).
3. **Runtime Instance** Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

What it does

The JVM performs following operation: Loads code

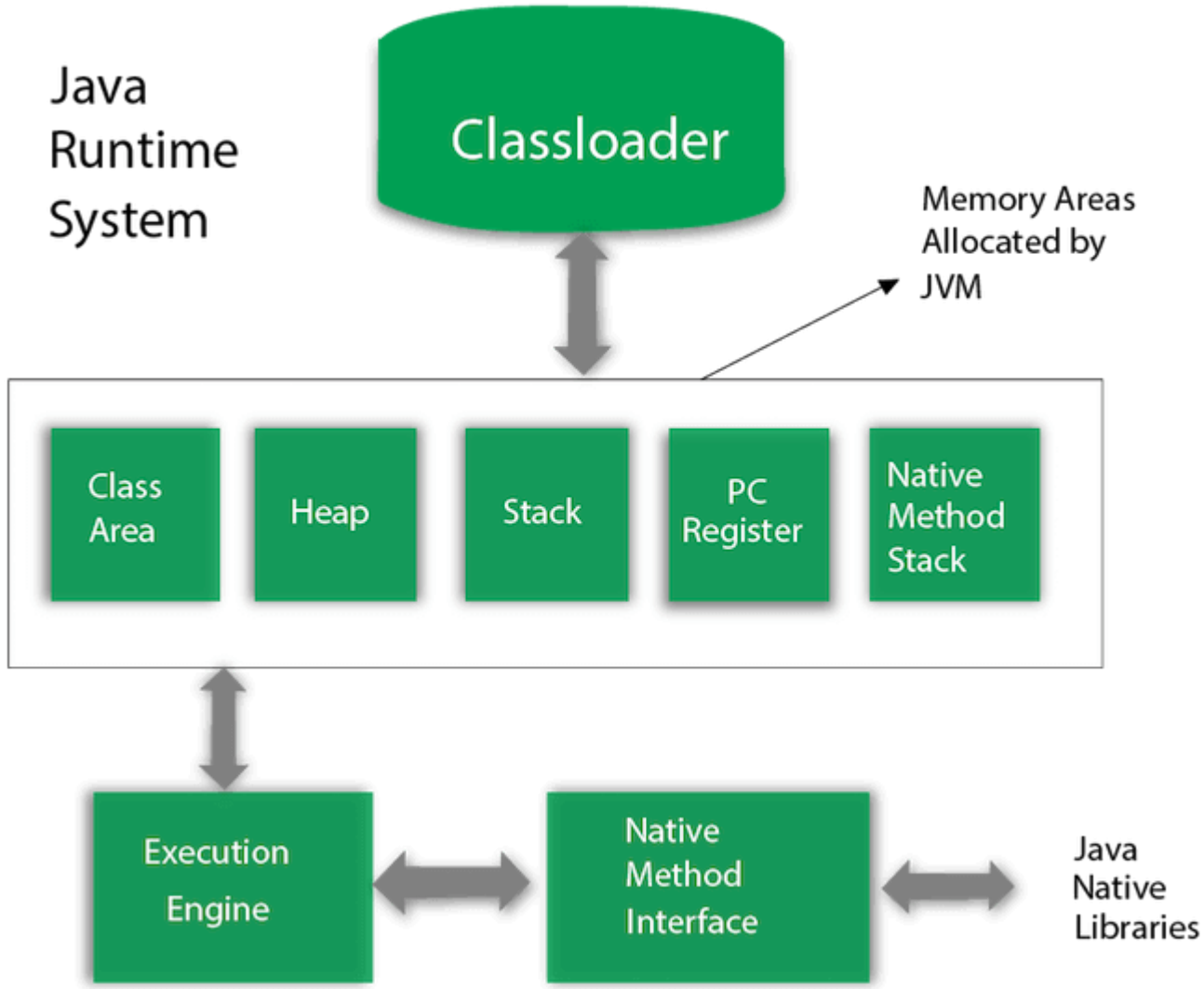
- Verifies code
- Executes code
- Provides runtime environment

JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.

JVM Architecture

Let's understand the internal architecture of JVM. It contains classloader, memory area, execution engine etc.



1) Classloader

Classloader is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the classloader. There are three built-in classloaders in Java.

1. **Bootstrap ClassLoader**: This is the first classloader which is the super class of Extension classloader. It loads the *rt.jar* file which contains all class files of Java Standard Edition like java.lang package classes, java.net package classes, java.util package classes, java.io package classes, java.sql package classes etc.
2. **Extension ClassLoader**: This is the child classloader of Bootstrap and parent classloader of System classloader. It loades the jar files located inside *$JAVA_HOME/jre/lib/ext* directory.
3. **System/Application ClassLoader**: This is the child classloader of Extension classloader. It loads the classfiles from classpath. By default, classpath is set to current directory. You can change the classpath using "-cp" or "-classpath" switch. It is also known as Application classloader.

```
//Let's see an example to print the classloader name
public class ClassLoaderExample
{
  public static void main(String[] args)
 {
      // Let's print the classloader name of current class.
     //Application/System classloader will load this class
      Class c=ClassLoaderExample.class;
      System.out.println(c.getClassLoader());
      //If we print the classloader name of String, it will print null because it is an
      //in-built class which is found in rt.jar, so it is loaded by Bootstrap classloader
      System.out.println(String.class.getClassLoader());
   }
}
```

Output:

```
sun.misc.Launcher$AppClassLoader@4e0e2f2a

null
```

These are the internal classloaders provided by Java. If you want to create your own classloader, you need to extend the ClassLoader class.

2) Class(Method) Area

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

3) Heap

It is the runtime data area in which objects are allocated.

4) Stack

Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return.

Each thread has a private JVM stack, created at the same time as thread.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

5) Program Counter Register

PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.

6) Native Method Stack

It contains all the native methods used in the application.

7) Execution Engine

It contains:

1. **A virtual processor**
2. **Interpreter:** Read bytecode stream then execute the instructions.
3. **Just-In-Time(JIT) compiler:** It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here, the term "compiler" refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

8) Java Native Interface

Java Native Interface (JNI) is a framework which provides an interface to communicate with another application written in another language like C, C++, Assembly etc. Java uses JNI framework to send output to the Console or interact with OS libraries.

Objects and Classes in Java

An **object** in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

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

State: represents the data (value) of an object.

Behavior: represents the behavior (functionality) of an object such as deposit, withdraw, etc.

Identity: An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

Characteristics of Object in Java

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

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

Object Definitions:

An object is a real-world entity.

An object is a runtime entity.

The object is an entity which has state and behavior.

The object is an instance of a class.

What is a class in Java

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

A class in Java can contain:

- Fields
- Methods

- Constructors
- Blocks
- Nested class and interface

Syntax to declare a class:

```
class <class_name>{

   field;

   method;

}
```

Instance variable in Java

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

Method in Java

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

Advantage of Method

- Code Reusability
- Code Optimization

new keyword in Java

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

Object and Class Example: main within the class

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

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

```
//Java Program to illustrate how to define a class and fields

//Defining a Student class.

class Student{
```

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

Output:

0

null

Object and Class Example: main outside the class

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.

We can have multiple classes in different Java files or single Java file. If you define multiple classes in a single Java source file, it is a good idea to save the file name with the class name which has main() method.

```
//Java Program to demonstrate having the main method in
//another class
//Creating Student class.
class Student{
```

```
 int id;
 String name;
}
//Creating another class TestStudent1 which contains the main method
class TestStudent1{
 public static void main(String args[]){
  Student s1=new Student();
  System.out.println(s1.id);
  System.out.println(s1.name);
 }
}
```

Output:

0

Null

3 Ways to initialize object

There are 3 ways to initialize object in Java.

- By reference variable
- By method
- By constructor

1) Object and Class Example: Initialization through reference

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

```
class Student{
 int id;
 String name;
}
```

```
class TestStudent2{
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name="Sonoo";
  System.out.println(s1.id+" "+s1.name);//printing members with a white space
 }
}
```

Output:

101 Sonoo

We can also create multiple objects and store information in it through reference variable.

```
class Student{
 int id;
 String name;
}
class TestStudent3{
 public static void main(String args[]){
  //Creating objects
  Student s1=new Student();
  Student s2=new Student();
  //Initializing objects
  s1.id=101;
  s1.name="Sonoo";
  s2.id=102;
```

```
  s2.name="Amit";

  //Printing data

  System.out.println(s1.id+" "+s1.name);

  System.out.println(s2.id+" "+s2.name);

 }

}
```

Output:

101 Sonoo

102 Amit

2) Object and Class Example: Initialization through method

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

```
class Student{

 int rollno;

 String name;

 void insertRecord(int r, String n){

  rollno=r;

  name=n;

 }

 void displayInformation(){System.out.println(rollno+" "+name);}

}

class TestStudent4{

 public static void main(String args[]){

  Student s1=new Student();
```

```
  Student s2=new Student();
  s1.insertRecord(111,"Karan");
  s2.insertRecord(222,"Aryan");
  s1.displayInformation();
  s2.displayInformation();
 }
}
```

Output:

111 Karan

222 Aryan

Object in Java with values

As you can see in the above figure, object gets the memory in heap memory area. The reference variable refers to the object allocated in the heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

3) Object and Class Example: Initialization through a constructor

Object and Class Example: Employee

```
class Employee{
    int id;
    String name;
    float salary;
    void insert(int i, String n, float s) {
        id=i;
        name=n;
        salary=s;
    }
```

```
    void display(){System.out.println(id+" "+name+" "+salary);}
}
public class TestEmployee {
public static void main(String[] args) {
    Employee e1=new Employee();
    Employee e2=new Employee();
    Employee e3=new Employee();
    e1.insert(101,"ajeet",45000);
    e2.insert(102,"irfan",25000);
    e3.insert(103,"nakul",55000);
    e1.display();
    e2.display();
    e3.display();
}
}
```

Output:

```
101 ajeet 45000.0
102 irfan 25000.0
103 nakul 55000.0
```

Object and Class Example: Rectangle

There is given another example that maintains the records of Rectangle class.

```
class Rectangle{
 int length;
 int width;
```

```
 void insert(int l, int w){
  length=l;
  width=w;
 }
 void calculateArea(){System.out.println(length*width);}
}
class TestRectangle1{
 public static void main(String args[]){
  Rectangle r1=new Rectangle();
  Rectangle r2=new Rectangle();
  r1.insert(11,5);
  r2.insert(3,15);
  r1.calculateArea();
  r2.calculateArea();
}
}
```

Output:

55

45

What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

- By new keyword
- By newInstance() method
- By clone() method
- By deserialization
- By factory method etc.

Anonymous object

Anonymous simply means nameless. An object which has no reference is known as an anonymous object. It can be used at the time of object creation only.

If you have to use an object only once, an anonymous object is a good approach. For example:

new Calculation();//anonymous object

Calling method through a reference:

Calculation c=new Calculation();

c.fact(5);

Calling method through an anonymous object

new Calculation().fact(5);

Let's see the full example of an anonymous object in Java.

```
class Calculation{
 void fact(int  n){
  int fact=1;
  for(int i=1;i<=n;i++){
   fact=fact*i;
  }
 System.out.println("factorial is "+fact);
}
public static void main(String args[]){
 new Calculation().fact(5);//calling method with anonymous object
}
}
```

Output:

Factorial is 120

Creating multiple objects by one type only

We can create multiple objects by one type only as we do in case of primitives.

Initialization of primitive variables:

int a=10, b=20;

Initialization of refernce variables:

Rectangle r1=new Rectangle(), r2=new Rectangle();//creating two objects

Let's see the example:

```
//Java Program to illustrate the use of Rectangle class which
//has length and width data members
class Rectangle{
 int length;
 int width;
 void insert(int l,int w){
  length=l;
  width=w;
 }
 void calculateArea(){System.out.println(length*width);}
}
class TestRectangle2{
 public static void main(String args[]){
  Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects
```

```
  r1.insert(11,5);
  r2.insert(3,15);
  r1.calculateArea();
  r2.calculateArea();
}
}
```

Output:

```
55
45
```

Real World Example: Account

```
//Java Program to demonstrate the working of a banking-system
//where we deposit and withdraw amount from our account.
//Creating an Account class which has deposit() and withdraw() methods
class Account{
int acc_no;
String name;
float amount;
//Method to initialize object
void insert(int a,String n,float amt){
acc_no=a;
name=n;
amount=amt;
}
//deposit method
```

```
void deposit(float amt){
amount=amount+amt;
System.out.println(amt+" deposited");
}
//withdraw method
void withdraw(float amt){
if(amount<amt){
System.out.println("Insufficient Balance");
}else{
amount=amount-amt;
System.out.println(amt+" withdrawn");
}
}
//method to check the balance of the account
void checkBalance(){System.out.println("Balance is: "+amount);}
//method to display the values of an object
void display(){System.out.println(acc_no+" "+name+" "+amount);}
}
//Creating a test class to deposit and withdraw amount
class TestAccount{
public static void main(String[] args){
Account a1=new Account();
a1.insert(832345,"Ankit",1000);
a1.display();
```

```
a1.checkBalance();

a1.deposit(40000);

a1.checkBalance();

a1.withdraw(15000);

a1.checkBalance();

}}
```

Output:

```
832345 Ankit 1000.0

Balance is: 1000.0

40000.0 deposited

Balance is: 41000.0

15000.0 withdrawn

Balance is: 26000.0
```

Constructors in Java

A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes.
In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling the constructor, memory for the object is allocated in the memory. It is a special type of method which is used to initialize the object. Every time an object is created using the new() keyword, at least one constructor is called.

How methods are Different From Methods in Java?

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

Now let us come up with the syntax for the constructor being invoked at the time of object or instance creation.

```
class Geek
{
  .......

  // A Constructor
  new Geek() {
  }

  .......
}

// We can create an object of the above class
// using the below statement. This statement
// calls above constructor.
Geek obj = new Geek();
```

The first line of a constructor is a call to super() or this(), (a call to a constructor of a super-class or an overloaded constructor), if you don't type in the call to super in your constructor the compiler will provide you with a non-argument call to super at the first line of your code, the super constructor must be called to create an object:

```
import java.io.*;

 class Geeks {

    Geeks() { super(); }

    public static void main(String[] args)

    {

        Geeks geek = new Geeks();
```

```
    }

}
```

If you think your class is not a subclass it actually is, every class in java is the subclass of a class **object** even if you don't say extends object in your class definition.

Need of Constructor

Think of a Box. If we talk about a box class then it will have some class variables (say length, breadth, and height). But when it comes to creating its object(i.e Box will now exist in the computer's memory), then can a box be there with no value defined for its dimensions. The answer is no.

So constructors are used to assign values to the class variables at the time of object creation, either explicitly done by the programmer or by Java itself (default constructor).

When is a Constructor called?

Each time an object is created using a **new()** keyword, at least one constructor (it could be the default constructor) is invoked to assign initial values to the **data members** of the same class.

The rules for writing constructors are as follows:

- Constructor(s) of a class must have the same name as the class name in which it resides.
- A constructor in Java can not be abstract, final, static, or Synchronized.
- Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.

So by far, we have learned constructors are used to initialize the object's state. Like methods, a constructor also contains a collection of statements(i.e. instructions) that are executed at the time of Object creation.

Types of Constructors in Java

Now is the correct time to discuss the types of the constructor, so primarily there are two types of constructors in java:

- No-argument constructor
- Parameterized Constructor
- Copy Constructor

1. No-argument constructor: A constructor that has no parameter is known as the default constructor. If we don't define a constructor in a class, then the compiler creates a **default constructor(with no arguments)** for the class. And if we write a constructor with arguments or no arguments then the compiler does not create a default constructor.

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

Example:

```
// Java Program to illustrate calling a
// no-argument constructor

import java.io.*;

class Geek {
    int num;
    String name;

    // this would be invoked while an object
    // of that class is created.
    Geek() { System.out.println("Constructor called"); }
}

class GFG {
    public static void main(String[] args)
    {
        // this would invoke default constructor.
        Geek geek1 = new Geek();
```

```
        // Default constructor provides the default

        // values to the object like 0, null

        System.out.println(geek1.name);

        System.out.println(geek1.num);

    }

}
```

Output

Constructor called

null

0

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

Example:

```
// Java Program to Illustrate Working of

// Parameterized Constructor

 // Importing required inputoutput class

import java.io.*;

 // Class 1

class Geek {

    // data members of the class.
```

```
    String name;
    int id;
     // Constructor would initialize data members
    // With the values of passed arguments while
    // Object of that class created
    Geek(String name, int id)
    {
        this.name = name;
        this.id = id;
    }
}
 // Class 2
class GFG {
    // main driver method
    public static void main(String[] args)
    {
        // This would invoke the parameterized constructor.
        Geek geek1 = new Geek("adam", 1);
        System.out.println("GeekName :" + geek1.name
                           + " and GeekId :" + geek1.id);
```

```
    }

}
```

Output
GeekName :adam and GeekId :1

3.Copy Constructor

Like C++, Java also supports a copy constructor. But, unlike C++, Java doesn't create a default copy constructor if you don't write your own. A prerequisite prior to learning copy constructors is to learn about constructors in java to deeper roots. Below is an example Java program that shows a simple use of a copy constructor.

Example 1A

```
// Java Program to Illustrate Copy Constructor

 // Class 1

class Complex {

    // Class data members

    private double re, im;

     // Constructor 1

    // Parameterized constructor

    public Complex(double re, double im)

    {

        // this keyword refers to current instance itself

        this.re = re;
```

```
        this.im = im;
    }
    // Constructor 2
    // Copy constructor
    Complex(Complex c)
    {
        System.out.println("Copy constructor called");
        re = c.re;
        im = c.im;
    }
    // Overriding the toString() of Object class
    @Override public String toString()
    {
        return "(" + re + " + " + im + "i)";
    }
}
// Class 2
// Main class
public class Main {
```

```
    // Main driver method
    public static void main(String[] args)
    {
        // Creating object of above class
        Complex c1 = new Complex(10, 15);
        // Following involves a copy constructor call
        Complex c2 = new Complex(c1);
        // Note: Following doesn't involve a copy
        // constructor call
        // as non-primitive variables are just references.
        Complex c3 = c2;
        // toString() of c2 is called here
        System.out.println(c2);
    }
}
```

Output

```
Copy constructor called
(10.0 + 15.0i)
```

Constructor Overloading

Now the most important topic that comes into play is the strong incorporation of OOPS with constructors known as constructor overloading. JustLike methods, we can overload constructors for creating objects in different ways. Compiler differentiates constructors on the basis of numbers of parameters, types of the parameters, and order of the parameters.

Example:

```
// Java Program to illustrate constructor overloading

// using same task (addition operation ) for different

// types of arguments.

import java.io.*;

class Geek

{

    // constructor with one argument

    Geek(String name)

    {

        System.out.println("Constructor with one " +

                    "argument - String : " + name);

    }

     // constructor with two arguments

    Geek(String name, int age)

    {
```

```
        System.out.println("Constructor with two arguments : " +
                " String and Integer : " + name + " "+ age);


    }
     // Constructor with one argument but with different
    // type than previous..
    Geek(long id)
    {
        System.out.println("Constructor with one argument : " +
                                    "Long : " + id);
    }
}


class GFG
{
    public static void main(String[] args)
    {
        // Creating the objects of the class named 'Geek'
        // by passing different arguments
         // Invoke the constructor with one argument of
```

```
        // type 'String'.
        Geek geek2 = new Geek("Shikhar");

        // Invoke the constructor with two arguments
        Geek geek3 = new Geek("Dharmesh", 26);
         // Invoke the constructor with one argument of
        // type 'Long'.
        Geek geek4 = new Geek(325614567);
    }
}
```

Output

Constructor with one argument - String : Shikhar

Constructor with two arguments : String and Integer : Dharmesh 26

Constructor with one argument : Long : 325614567

Methods in Java

In general, a method is a way to perform some task. Similarly, the method in Java is a collection of instructions that performs a specific task. It provides the reusability of code. We can also easily modify code using methods. In this section, we will learn what is a method in Java, types of methods, method declaration, and how to call a method in Java.

What is a method in Java?

A method is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the reusability of code. We write a method once and use it many times. We do not require to write code again and

again. It also provides the easy modification and readability of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it.

The most important method in Java is the main() method. If you want to read more about the main() method, go through the link https://www.javatpoint.com/java-main-method

.**Method Declaration**

The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as method header, as we have shown in the following figure.

Method Signature: Every method has a method signature. It is a part of the method declaration. It includes the method name and parameter list.

Access Specifier: Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides four types of access specifier:

Public: The method is accessible by all classes when we use public specifier in our application.

Private: When we use a private access specifier, the method is accessible only in the classes in which it is defined.

Protected: When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.

Default: When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

Return Type: Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.

Method Name: It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be subtraction(). A method is invoked by its name.

Parameter List: It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.

Method Body: It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

Naming a Method

While defining a method, remember that the method name must be a verb and start with a lowercase letter. If the method name has more than two words, the first name must be a verb followed by adjective or noun. In the multi-word method name, the first letter of each word must be in uppercase except the first word. For example:

Single-word method name: sum(), area()

Multi-word method name: areaOfCircle(), stringComparision()

It is also possible that a method has the same name as another method name in the same class, it is known as method overloading.

Types of Method

There are two types of methods in Java:

- Predefined Method
- User-defined Method

Predefined Method

In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the standard library method or built-in method. We can directly use these methods just by calling them in the program at any point. Some pre-defined methods are length(), equals(), compareTo(), sqrt(), etc. When we call any of the predefined methods in our program, a series of codes related to the corresponding method runs in the background that is already stored in the library.

Each and every predefined method is defined inside a class. Such as print() method is defined in the java.io.PrintStream class. It prints the statement that we write inside the method. For example, print("Java"), it prints Java on the console.

Let's see an example of the predefined method.

```
public class Demo

{

public static void main(String[] args)

{

// using the max() method of Math class
```

```
System.out.print("The maximum number is: " + Math.max(9,7));

}

}
```

Output:

```
The maximum number is: 9
```

In the above example, we have used three predefined methods main(), print(), and max(). We have used these methods directly without declaration because they are predefined. The print() method is a method of PrintStream class that prints the result on the console. The max() method is a method of the Math class that returns the greater of two numbers.

We can also see the method signature of any predefined method by using the link https://docs.oracle.com/

When we go through the link and see the max() method signature, we find the following:

In the above method signature, we see that the method signature has access specifier public, non-access modifier static, return type int, method name max(), parameter list (int a, int b). In the above example, instead of defining the method, we have just invoked the method. This is the advantage of a predefined method. It makes programming less complicated.

Similarly, we can also see the method signature of the print() method.

User-defined Method

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

How to Create a User-defined Method

Let's create a user defined method that checks the number is even or odd. First, we will define the method.

```
//user defined method

public static void findEvenOdd(int num)

{

//method body

if(num%2==0)
```

```
System.out.println(num+" is even");

else

System.out.println(num+" is odd");

}
```

We have defined the above method named findevenodd(). It has a parameter num of type int. The method does not return any value that's why we have used void. The method body contains the steps to check the number is even or odd. If the number is even, it prints the number is even, else prints the number is odd.

How to Call or Invoke a User-defined Method

Once we have defined a method, it should be called. The calling of a method in a program is simple. When we call or invoke a user-defined method, the program control transfer to the called method.

```
import java.util.Scanner;

public class EvenOdd

{

public static void main (String args[])

{

//creating Scanner class object

Scanner scan=new Scanner(System.in);

System.out.print("Enter the number: ");

//reading value from the user

int num=scan.nextInt();

//method calling

findEvenOdd(num);

}
```

In the above code snippet, as soon as the compiler reaches at line findEvenOdd(num), the control transfer to the method and gives the output accordingly.

Let's combine both snippets of codes in a single program and execute it.

EvenOdd.java

```
import java.util.Scanner;
public class EvenOdd
{
public static void main (String args[])
{
//creating Scanner class object
Scanner scan=new Scanner(System.in);
System.out.print("Enter the number: ");
//reading value from user
int num=scan.nextInt();
//method calling
findEvenOdd(num);
}
//user defined method
public static void findEvenOdd(int num)
{
//method body
if(num%2==0)
System.out.println(num+" is even");
else
System.out.println(num+" is odd");
}
```

}

Output 1:

Enter the number: 12

12 is even

Output 2:

Enter the number: 99

99 is odd

Let's see another program that return a value to the calling method.

In the following program, we have defined a method named add() that sum up the two numbers. It has two parameters n1 and n2 of integer type. The values of n1 and n2 correspond to the value of a and b, respectively. Therefore, the method adds the value of a and b and store it in the variable s and returns the sum.

Addition.java

```
public class Addition

{

public static void main(String[] args)

{

int a = 19;

int b = 5;

//method calling

int c = add(a, b);   //a and b are actual parameters

System.out.println("The sum of a and b is= " + c);

}

//user defined method

public static int add(int n1, int n2)   //n1 and n2 are formal parameters
```

```
{

int s;

s=n1+n2;

return s; //returning the sum

}

}
```

Output:

The sum of a and b is= 24

Static Method

A method that has static keyword is known as static method. In other words, a method that belongs to a class rather than an instance of a class is known as a static method. We can also create a static method by using the keyword static before the method name.

The main advantage of a static method is that we can call it without creating an object. It can access static data members and also change the value of it. It is used to create an instance method. It is invoked by using the class name. The best example of a static method is the main() method.

Example of static method

```
public class Display

{

public static void main(String[] args)

{

show();

}

static void show()

{

System.out.println("It is an example of static method.");

}
```

```
}
```

Output:

It is an example of a static method.

Instance Method

The method of the class is known as an instance method. It is a non-static method defined in the class. Before calling or invoking the instance method, it is necessary to create an object of its class. Let's see an example of an instance method.

```
public class InstanceMethodExample

{

public static void main(String [] args)

{

//Creating an object of the class

InstanceMethodExample obj = new InstanceMethodExample();

//invoking instance method

System.out.println("The sum is: "+obj.add(12, 13));

}

int s;

//user-defined method because we have not used static keyword

public int add(int a, int b)

{

s = a+b;

//returning the sum

return s;

}

}
```

Output:

The sum is: 25

There are two types of instance method:

- Accessor Method
- Mutator Method

Accessor Method: The method(s) that reads the instance variable(s) is known as the accessor method. We can easily identify it because the method is prefixed with the word get. It is also known as getters. It returns the value of the private field. It is used to get the value of the private field.

Example

```
public int getId()

{

return Id;

}
```

Mutator Method: The method(s) read the instance variable(s) and also modify the values. We can easily identify it because the method is prefixed with the word set. It is also known as setters or modifiers. It does not return anything. It accepts a parameter of the same data type that depends on the field. It is used to set the value of the private field.

Example

```
public void setRoll(int roll)

{

this.roll = roll;

}
```

Example of accessor and mutator method

Student.java

```
public class Student
```

```
{
private int roll;
private String name;
public int getRoll()    //accessor method
{
return roll;
}
public void setRoll(int roll) //mutator method
{
this.roll = roll;
}
public String getName()
{
return name;
}
public void setName(String name)
{
this.name = name;
}
public void display()
{
System.out.println("Roll no.: "+roll);
System.out.println("Student name: "+name);
}
```

}

Abstract Method

The method that does not has method body is known as abstract method. In other words, without an implementation is known as abstract method. It always declares in the abstract class. It means the class itself must be abstract if it has abstract method. To create an abstract method, we use the keyword abstract.

Syntax

abstract void method_name();

Example of abstract method

Demo.java

```
abstract class Demo //abstract class

{

//abstract method declaration

abstract void display();

}

public class MyClass extends Demo

{

//method impelmentation

void display()

{

System.out.println("Abstract method?");

}

public static void main(String args[])

{

//creating object of abstract class

Demo obj = new MyClass();
```

```
//invoking abstract method

obj.display();

}

}
```

Output:

```
Abstract method...

Factory method
```

It is a method that returns an object to the class to which it belongs. All static methods are factory methods. For example, NumberFormat obj = NumberFormat.getNumberInstance();

Method Overloading in Java

Having multiple methods having same name but different in parameters, it is known as Method Overloading.

Different ways to overload the method

- By changing the no. of arguments
- By changing the datatype

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

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

Advantage of method overloading

- Method overloading increases the readability of the program.

Different ways to overload the method

There are two ways to overload the method in java

- By changing number of arguments
- By changing the data type

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

1) Method Overloading: changing no. of arguments

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

In this example, we are creating static methods

so that we don't need to create instance for calling methods.

```
class Adder{

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

static int add(int a,int b,int c){return a+b+c;}

}

class TestOverloading1{

public static void main(String[] args){

System.out.println(Adder.add(11,11));

System.out.println(Adder.add(11,11,11));

}}
```

Output:

22

33

2) Method Overloading: changing data type of arguments

In this example, we have created two methods that differs in data type

The first add method receives two integer arguments and second add method receives two double arguments.

```
class Adder{
```

```
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

Output:

22

24.9

Example :

```
public class Sum {
     // Overloaded sum(). This sum takes two int parameters
    public int sum(int x, int y) { return (x + y); }

    // Overloaded sum(). This sum takes three int parameters
    public int sum(int x, int y, int z)
    {
        return (x + y + z);
    }

    // Overloaded sum(). This sum takes two double
    // parameters
```

```
    public double sum(double x, double y)
    {
        return (x + y);
    }
   // Driver code
    public static void main(String args[])
    {
        Sum s = new Sum();
        System.out.println(s.sum(10, 20));
        System.out.println(s.sum(10, 20, 30));
        System.out.println(s.sum(10.5, 20.5));
    }
}
```

Output :

30

60

31.0

Parameter Passing

There are different ways in which parameter data can be passed into and out of methods and functions. Let us assume that a function B() is called from another function A(). In this case A is called the “caller function” and B is called the “called function or callee function”. Also, the arguments which A sends to B are called actual arguments and the parameters of B are called formal arguments.

Types of parameters:

Formal Parameter: A variable and its type as they appear in the prototype of the function or method.

Syntax:

function_name(datatype variable_name)

Actual Parameter: The variable or expression corresponding to a formal parameter that appears in the function or method call in the calling environment.

Syntax:

func_name(variable name(s));

Important methods of Parameter Passing

1. Pass By Value: Changes made to formal parameter do not get transmitted back to the caller. Any modifications to the formal parameter variable inside the called function or method affect only the separate storage location and will not be reflected in the actual parameter in the calling environment. This method is also called as call by value.

Java in fact is strictly call by value.

Example:

```
// Java program to illustrate

// Call by Value

// Callee

class CallByValue {

    // Function to change the value

   // of the parameters

   public static void example(int x, int y)

   {

      x++;

      y++;

   }

}
```

```
// Caller
public class Main {
    public static void main(String[] args)
    {
        int a = 10;
       int b = 20;
        // Instance of class is created
       CallByValue object = new CallByValue();
        System.out.println("Value of a: " + a
                           + " & b: " + b);
        // Passing variables in the class function
       object.example(a, b);
        // Displaying values after
       // calling the function
       System.out.println("Value of a: "
                           + a + " & b: " + b);
    }
}
```

Output:

Value of a: 10 & b: 20

Value of a: 10 & b: 20

Shortcomings:

- Inefficiency in storage allocation
- For objects and arrays, the copy semantics are costly

2. Call by reference(aliasing): Changes made to formal parameter do get transmitted back to the caller through parameter passing. Any changes to the formal parameter are reflected in the actual parameter in the calling environment as formal parameter receives a reference (or pointer) to the actual data. This method is also called as call by reference. This method is efficient in both time and space.

Example

```
// Java program to illustrate

// Call by Reference

 // Callee

class CallByReference {

     int a, b;

     // Function to assign the value

    // to the class variables

    CallByReference(int x, int y)

    {

        a = x;

        b = y;

    }

     // Changing the values of class variables

    void ChangeValue(CallByReference obj)

    {

        obj.a += 10;

        obj.b += 20;

    }

}
```

```
// Caller
public class Main {
    public static void main(String[] args)
    {
        // Instance of class is created
        // and value is assigned using constructor
        CallByReference object
            = new CallByReference(10, 20);

        System.out.println("Value of a: " + object.a
                           + " & b: " + object.b);

        // Changing values in class function
        object.ChangeValue(object);
        // Displaying values
        // after calling the function
        System.out.println("Value of a: " + object.a
                           + " & b: " + object.b);
    }
}
```

Output:

```
Value of a: 10 & b: 20
Value of a: 20 & b: 40
```

String Handling

String is an object that represents sequence of characters. In Java, String is represented by String class which is located into java.lang package

It is probably the most commonly used class in java library. In java, every string that we create is actually an object of type String. One important thing to notice about string object is that string objects are immutable that means once a string object is created it cannot be changed.

The Java String class implements Serializable, Comparable and CharSequence interface that we have represented using the below image.

In Java, CharSequence Interface is used for representing a sequence of characters. CharSequence interface is implemented by String, StringBuffer and StringBuilder classes. This three classes can be used for creating strings in java.

What is an Immutable object?

An object whose state cannot be changed after it is created is known as an Immutable object. String, Integer, Byte, Short, Float, Double and all other wrapper classes objects are immutable.

Creating a String object

String can be created in number of ways, here are a few ways of creating string object.

1) Using a String literal

String literal is a simple string enclosed in double quotes " ". A string literal is treated as a String object.

```
public class Demo{

   public static void main(String[] args) {

        String s1 = "Hello Java";

        System.out.println(s1);

   }

}
```

Output :

Hello Java

2) Using new Keyword

We can create a new string object by using new operator that allocates memory for the object.

```
public class Demo{

   public static void main(String[] args) {

        String s1 = new String("Hello Java");

        System.out.println(s1);

   }

}
```

Output :

Hello Java

Each time we create a String literal, the JVM checks the string pool first. If the string literal already exists in the pool, a reference to the pool instance is returned. If string does not exist in the pool, a new string object is created, and is placed in the pool. String objects are stored in a special memory area known as string constant pool inside the heap memory.

String object and How they are stored

When we create a new string object using string literal, that string literal is added to the string pool, if it is not present there already.

```
String str= "Hello";
```

Creating String in heap

And, when we create another object with same string, then a reference of the string literal already present in string pool is returned.

```
String str2 = str;
```

Creating String in heap

But if we change the new string, its reference gets modified.

str2=str2.concat("world");

Creating String in heap

Concatenating String

There are 2 methods to concatenate two or more string.

- Using concat() method
- Using + operator

1) Using concat() method

Concat() method is used to add two or more string into a single string object. It is string class method and returns a string object.

```
public class Demo{

   public static void main(String[] args) {

        String s = "Hello";

        String str = "Java";

        String str1 = s.concat(str);

        System.out.println(str1);

   }

}
```

Output :

HelloJava

2) Using + operator

Java uses "+" operator to concatenate two string objects into single one. It can also concatenate numeric value with string object. See the below example.

```
public class Demo{

   public static void main(String[] args) {

        String s = "Hello";
```

```
        String str = "Java";
        String str1 = s+str;
        String str2 = "Java"+11;
        System.out.println(str1);
        System.out.println(str2);
    }
}
```

Output :

HelloJava

Java11

String Comparison

To compare string objects, Java provides methods and operators both. So we can compare string in following three ways.

- Using equals() method
- Using == operator
- By CompareTo() method

Using equals() method

equals() method compares two strings for equality. Its general syntax is,

boolean equals (Object str)

Example

It compares the content of the strings. It will return true if string matches, else returns false.

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

```
        String s = "Hell";
        String s1 = "Hello";
        String s2 = "Hello";
        boolean b = s1.equals(s2);    //true
        System.out.println(b);
        b =      s.equals(s1) ;   //false
        System.out.println(b);
    }
}
```

Output :

true

false

Using == operator

The double equal (==) operator compares two object references to check whether they refer to same instance. This also, will return true on successful match else returns false.

```
public class Demo{
    public static void main(String[] args) {
        String s1 = "Java";
        String s2 = "Java";
        String s3 = new String ("Java");
        boolean b = (s1 == s2);     //true
        System.out.println(b);
```

```
        b =     (s1 == s3);     //false

        System.out.println(b);

    }

}
```

Output :

true

false

Explanation

We are creating a new object using new operator, and thus it gets created in a non-pool memory area of the heap. s1 is pointing to the String in string pool while s3 is pointing to the String in heap and hence, when we compare s1 and s3, the answer is false.

By compareTo() method

String compareTo() method compares values and returns an integer value which tells if the string compared is less than, equal to or greater than the other string. It compares the String based on natural ordering i.e alphabetically. Its general syntax is.

Syntax:

int compareTo(String str)

Example:

```
public class HelloWorld{

    public static void main(String[] args) {

        String s1 = "Abhi";

        String s2 = "Viraaj";

        String s3 = "Abhi";

        int a = s1.compareTo(s2);     //return -21 because s1 < s2

        System.out.println(a);

        a = s1.compareTo(s3);     //return 0 because s1 == s3
```

```
        System.out.println(a);
        a = s2.compareTo(s1);      //return 21 because s2 > s1
        System.out.println(a);
    }
}
```

Output:

```
-21
0
21
```

Java String class methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

No.	Method	Description
1	char charAt(int index)	It returns char value for the particular index
2	int length()	It returns string length
3	static String format(String format, Object... args)	It returns a formatted string.
4	static String format(Locale l, String format, Object... args)	It returns formatted string with given locale.
5	String substring(int beginIndex)	It returns substring for given begin index.
6	String substring(int beginIndex, int endIndex)	It returns substring for given begin index and end index.
7	boolean contains(CharSequence s)	It returns true or false after matching the sequence of char value.
8	static String join(CharSequence delimiter, CharSequence... elements)	It returns a joined string.
9	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	It returns a joined string.
10	boolean equals(Object another)	It checks the equality of string with the given object.
11	boolean isEmpty()	It checks if string is empty.
12	String concat(String str)	It concatenates the specified string.
13	String replace(char old, char new)	It replaces all occurrences of the specified char value.
14	String replace(CharSequence old, CharSequence new)	It replaces all occurrences of the specified CharSequence.
15	static String equalsIgnoreCase(String another)	It compares another string. It doesn't check case.
16	String[] split(String regex)	It returns a split string matching regex.
17	String[] split(String regex, int limit)	It returns a split string matching regex and limit.
18	String intern()	It returns an interned string.
19	int indexOf(int ch)	It returns the specified char value index.
20	int indexOf(int ch, int fromIndex)	It returns the specified char value index starting with given index.

21	int indexOf(String substring)	It returns the specified substring index.
22	int indexOf(String substring, int fromIndex)	It returns the specified substring index starting with given index.
23	String toLowerCase()	It returns a string in lowercase.
24	String toLowerCase(Locale l)	It returns a string in lowercase using specified locale.
25	String toUpperCase()	It returns a string in uppercase.
26	String toUpperCase(Locale l)	It returns a string in uppercase using specified locale.
27	String trim()	It removes beginning and ending spaces of this string.
28	static String valueOf(int value)	It converts given type into string. It is an overloaded method.

Example:

```
public class JavaStringExample {
    public static void main(String[] args) {
        String title = "Java Tutorials";
        String siteName = "www.btechsmartclass.com";
        System.out.println("Length of title: " + title.length());
        System.out.println("Char at index 3: " + title.charAt(3));
        System.out.println("Index of 'T': " + title.indexOf('T'));
        System.out.println("Last index of 'a': " + title.lastIndexOf('a'));
        System.out.println("Empty: " + title.isEmpty());
        System.out.println("Ends with '.com': " + siteName.endsWith(".com"));
        System.out.println("Equals: " + siteName.equals(title));
        System.out.println("Sub-string: " + siteName.substring(9, 14));
```

```
        System.out.println("Upper case: " + siteName.toUpperCase());
    }
}
```

Output