CS6308- Java Programming

V P Jayachitra
Assistant Professor
Department of Computer Technology
MIT Campus
Anna University

MODULE II JAVA OBJECTS -1	L	T	P	EL
	3	0	4	3
Classes and Objects, Constructor, Destructor, Static instances, this, constants, Thinking in Objects, String class, Text I/O				
SUGGESTED ACTIVITIES : • Flipped classroom • Practical - Implementation of Java programs – using String class, Creating Classes and objects • EL – Thinking in Objects				
SUGGESTED EVALUATION METHODS: • Assignment problems • Quizzes				

Introduction

- **procedural programming:** Programs that perform their behavior as a series of steps to be carried out.
- **object-oriented programming (OOP)**: Programs that perform their behavior as interactions between objects

About ?

- The basic elements of a class
- How a class can be used to create objects?
- Learn about methods, constructors, and the **this** keyword.

Class

- Any concept to implement in a Java program must be encapsulated within a class.
- What is a class?
  - A class is a structure that defines the data and the methods to work on that data.
  - A new data type
  - A class is an encapsulation of attributes and methods.
  - A class is a logical construct or template.

Class Fundamentals

- Class
  - Defines a new data type
  - A class is to create an object or instance of the defined type
  - A class is a template for an object
  - A class is the blueprint of an object
  - A class describe object's properties and behaviors
- Example: Blueprint of a house (class) and the house (object)

Blueprint

```
public class Point
{
    int x;
    int y;
}
```

Instance

```
Point p=new Point();
```

Class

Properties : State/variables

Behaviors: Methods

Class Clock

Properties : State/variables

int Hour, minute, second;

Behaviors: Methods

int getMinutes();
int getSeconds();
int getHours();

Container class vs Definition class

- Container class:
  - Collection of static methods that are not bound to a specific objects.
  - Example: Math.sqrt(), Math.pow()
- Definition class:
  - A class that create new objects.
  - Example: Clock c1;

Object

- Object
  - An object is an instance of a class.
  - An entity that combines state and behavior
  - An object is a real world entity.

Characteristics of an Object:

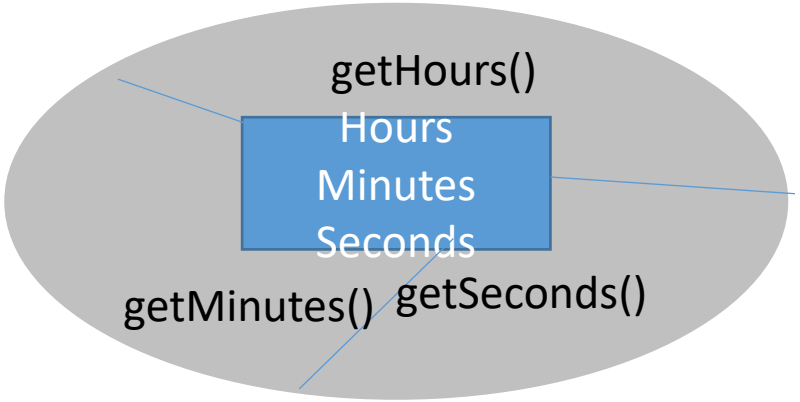
  - State : represents the data (value) of an object.
  - Behavior : represents the behavior (functionality) of an object.
  - Identity : An object identity is typically implemented via unique ID. The value of the ID is not visible to the external user. However, JVM can identify each object uniquely.

Class Abstraction

- Abstract data type (ADT)
  - Abstraction refers to the act of representing essential features without including the background details or explanations.
  - Class is an ADT as it uses the concept of abstraction.
- Abstraction:
  - An abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user.
  - Allow user to understand its external behaviour(interface) but not its internal details.
  - Example: Buttons
  - Why? To manage complexity
  - abstraction is achieved by using Interfaces and Abstract classes.

Encapsulation

- State is *encapsulated* or *hidden*
  - The internal state of an object is not directly accessible to other parts of the program
  - Other parts of the program can only access the object using its interface.
  - Data-hiding: Data of a class is hidden from any other class and can be accessed only through any member function of it's own class in which they are declared.



General form of the class

```
access modifier class ClassName {
   access modifier static type classVariable1; // Class variables)
   // ...
    access modifier type instanceVariable1; // Instance variables
   // ...
    access modifier ClassName() {// Default constructor
      // code   }
   // Parameterized constructor
   access modifier ClassName(type param1, type param2, ...) {
      this.instanceVariable1 = param1;
      this.instanceVariable2 = param2;
      // ...                                    }
   // Method with a return type
   access modifier ReturnType methodName1(ParameterType
param1, ParameterType param2, ...) {
      // Method body
      return returnValue;     }
   // Method without a return type (void)
      access modifier void methodName2(ParameterType
param1, ParameterType param2, ...) {
         // Method body
      }

      // Static method
      access modifier static ReturnType
staticMethodName(ParameterType param1, ParameterType
param2, ...) {
         // Method body
         return returnValue;
      }

      // Main method (entry point)
      access modifier static void main(String[] args) {
         // Code to test the class
      }
}
```

- The data, or variables, defined within a class are called instance variables.
  - Each instance of the class (that is, each object of the class) contains its own copy of these variables.
  - Thus, the data for one object is separate and unique from the data for another.
- The code is contained within .defined within a class are called members of the class.
- Classes have a main method(), if that class is the starting point of the program

A simple class

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

Create a Car object, use a statement like the following:

Car myCar = new Car();

// create a Car object called myCar

Instance variables

A class called Car that defines three instance variables:

make, model and year.

Class name

A class defines a new type of data.

In this case, the new data type is called Car.

class name is used to declare objects of type Car.

A class declaration only creates a template;

It does not create an actual object.

Object

myCar will refer to an instance of Car. Thus, it will have “physical” reality.

Each time on creating an instance of a class means creating an object that contains its own copy of each instance variable defined by the class

Every Car object will contain its own copies of the instance variables make, model, and year.

A simple class

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

To actually create a Car object, use a statement like the following:

```
Car myCar = new Car();
// create a Car object called mycar
```

Dot operator

- Use the dot operator to access both the instance variables and the methods within an object.
- The dot operator links the name of the object with the name of an instance variable or methods.

```
myCar.year = 2024;
//assign year variable of myCar the value 2024
```

```
class Car {
    String make;
    String model;
    int year;
    double fuelEfficiency;
}

// This class declares an object of type Car.
class CarDemo {
    public static void main(String args[]) {
        Car myCar = new Car();
        double range;  //local variable
        // myCar instance variables
        myCar.make = "Tesla ";
        myCar.model = " Roadster ";
        myCar.year = 2024;
        myCar.fuelEfficiency = 0.5;
        // compute range of car (assuming 15-gallon tank)
        range = myCar.fuelEfficiency * 15;
        System.out.println("A " + myCar.year + " " + myCar.make + " " + myCar.model +  " can travel approximately " + range +
" miles on a full tank.");
    }
}
```

A 2024 Tesla Roadster can travel approximately 7.5 miles on a full tank.

Save the file that contains this program CarDemo.java, because the main( ) method is in the class called CarDemo, not the class called Car.
Compilation will create two .class files, one for Car and one for CarDemo.
Each class can also be defined in its own file, called Car.java and CaeDemo.java,

```
//TWO OBJECTS
class Car {
    String make;
    String model;
    int year;
    double fuelEfficiency;  // miles per gallon
}
class CarDemo2 {
    public static void main(String args[]) {
        Car myCar1 = new Car();
        Car myCar2 = new Car();
        double range;

        // assign values to myCar1's instance variables
        myCar1.make = "Tesla";
        myCar1.model = "Roadster";
        myCar1.year = 2024;
        myCar1.fuelEfficiency = 0.5;

        // assign different values to myCar2's instance variables
        myCar2.make = "Tesla";
        myCar2.model = "Cybertruck";
        myCar2.year = 2024;
        myCar2.fuelEfficiency = 34.0;  // MPGe for electric cars
```

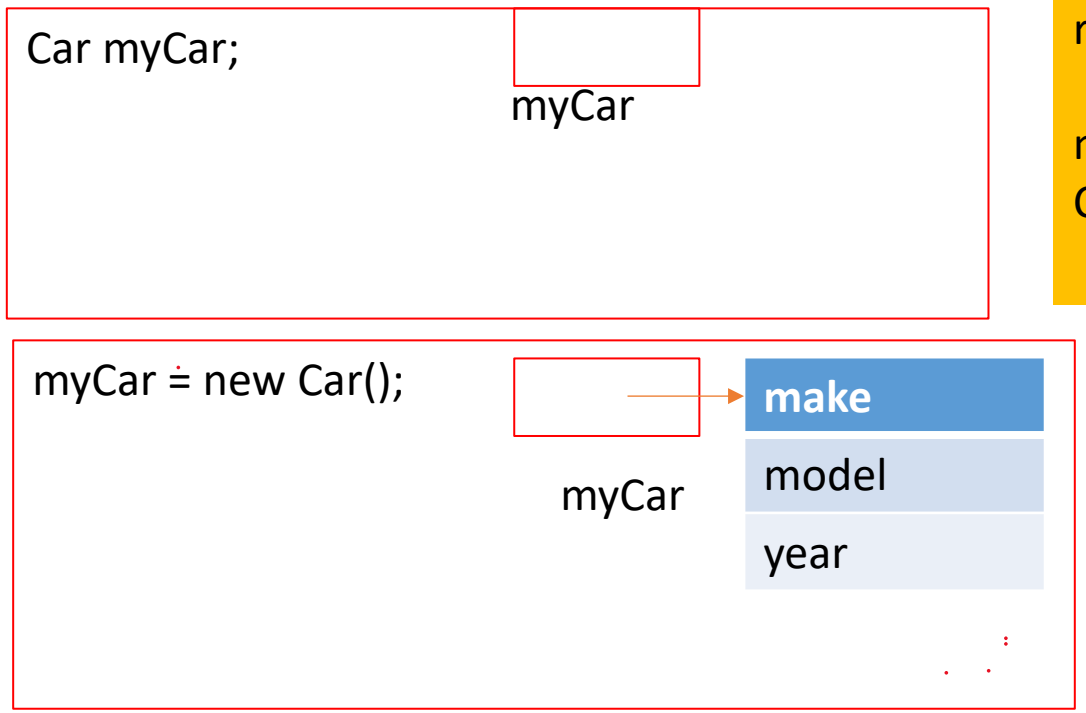
```
//compute range of first car (assuming 15-gallon tank)
        range = myCar1.fuelEfficiency * 15;
System.out.println("A " + myCar1.year + " " + myCar1.make + " " + myCar1.model
+ " can travel approximately " + range + " miles on a full tank.")

// compute range of second car (assuming 75 kWh battery)
        range = myCar2.fuelEfficiency * (75 / 33.7);
        System.out.println("A " + myCar2.year + " " + myCar2.make + " " +
myCar2.model + " can travel approximately " + range + " miles on a full charge.");
```

```
A 2024 Tesla Roadster can travel approximately 7.5 miles on a full tank.
A 2024 Tesla Cybertruck can travel approximately 75.66765578635014 miles on a full charge.
```

Declaring Objects

```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```



Declaring an object of type Car

Declare object

The first line declares myCar as a reference to an object of type Car.

At this point, myCar does not yet refer to an actual object.

The next line allocates an object and assigns a reference to it to myCar.

myCar simply holds the memory address of the actual Car object.

new operator

- The new operator dynamically allocates memory for an object.

  class-var = new classname ( );

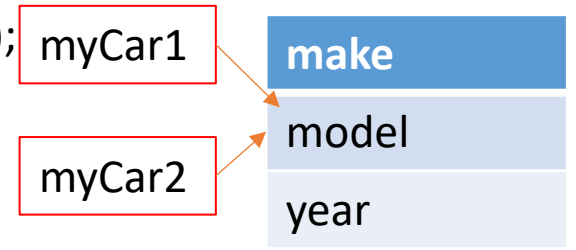
- The classname is the name of the class that is being instantiated.
- The class name followed by parentheses specifies the constructor for the class.
- A **constructor** defines what occurs when an object of a class is created.
- Most real-world classes explicitly define their own constructors within their class definition.
- However, if no explicit constructor is specified, then Java will automatically supply a default constructor.

Assigning Object Reference Variables

```
Car c1 = new Car();
Car c2 = c1;
```

- c1 and c2 will both refer to the same object.
- The assignment of c1 to c2 did not allocate any memory or copy any part of the original object.
- It simply makes c2 refer to the same object as does c1.
- Thus, any changes made to the object through b2 will affect the object to which c1 is referring, since they are the same object.
- Assign one object reference variable to another object reference variable, is not creating a copy of the object, only making a copy of the reference.

```
Car myCar1 = new Car();
Car myCar2=mycar1
```




```
Car c1 = new Car();
 Car c2 = c1;
// ...
c1 = null;
```

Here, c1 has been set to null, but c2 still points to the original object.

Introducing Methods

- Classes usually consist of two things:
  - instance variables
  - methods.
- This is the general form of a method:

```
type name(parameter-list) {
// body of method
return value;
 }
```

Type

Type specifies the type of data returned by the method.

This can be any valid type, including class types

If the method does not return a value, its return type must be void.

Name

The name of the method is specified by name.

This can be any legal identifier .

Parameter-list

The parameter-list is a sequence of type and identifier pairs separated by commas.

Parameters are essentially variables that receive the value of the arguments passed to the method when it is called.

If the method has no parameters, then the parameter list will be empty.

Return

Methods that have a return type other than void return a value to the calling routine using the return statement:

Adding a Method to the Car Class

Use methods to access the instance variables defined by the class.

methods define the interface to most classes.

This allows the class implementor to hide the specific layout of internal data structures behind cleaner method abstractions.

In addition to defining methods that provide access to data, we can also define methods that are used internally by the class itself.

Instance variable that is not part of the class, it must be accessed through an object, by use of the dot operator.
Instance variable that is part of the same class, that variable can be accessed directly.
The same thing applies to methods.

```
class Car {
    String make;
    String model;
    int year;
    double fuelEfficiency;  // miles per gallon
    // Method to calculate the range of the car
    public double calculateRange(double tankSize
        return fuelEfficiency * tankSize;
    }
} class CarDemo {
    public static void main(String args[]) {
        Car myCar = new Car();
        double tankSize = 15.0; // Size of the fuel tank in gallons
        myCar.make = "Tesla";
        myCar.model = "Roadster";
        myCar.year = 2024;
        myCar.fuelEfficiency = 0.5; // miles per gallon

        // Compute the range of the car using the new method
        double range = myCar.calculateRange(tankSize);
System.out.println("A " + myCar.year + " " + myCar.make + " " +
myCar.model + " can travel approximately " + range + " miles on a full
tank.");
```

No dot operator to access the class instance variable within class

dot operator to access the class instance variable outside the class

A 2024 Tesla Roadster can travel approximately 7.5 miles on a full tank.

{

Adding a Method That Takes Parameters

```
int square(){
   return 5*5;
}
```

```
int square(n){
   return n*n;
}
```

```
int x, y;
x = square(5); // x equals 25
x = square(9); // x equals 81
 y = 2;
x = square(y); // x equals 4
```

argument

parameter

- A parameterized method can operate on a variety of data.
- A non parameterized method use is very limited

parameter and argument:

- A parameter is a variable defined by a method that receives a value when the method is called.
- For example, in square( ), i is a parameter.
- An argument is a value that is passed to a method when it is invoked.
- For example, square(100) passes 100 as an argument.

Constructors

- A constructor initializes an object immediately upon creation.
- Automatic initialization of object is performed through the use of a constructor.
- Constructor has the same name as the class in which it resides and is syntactically similar to a method.
- Once defined, the constructor is automatically called when the object is created, before the new operator completes.
- Constructor's have no return type, not even void.
  - The implicit return type of a class' constructor is the class type itself.

```
Car mybox1Car = new Car();
```

The parentheses are needed after the class name is to invoke the constructor for the class is being called.

Default vs defined constructor

- Java creates a default constructor for the class if a constructor is not explicitly defined for a class.
- The default constructor will initialize all non initialized instance variables to their default values.
  - zero, null, and false, for numeric types, reference types, and boolean.
- Once define your own constructor, the default constructor is no longer used.

```
String make;
String model;
int year;
double fuelEfficiency;
// No-argument constructor
public Car() {
    // Default values
    this.make = "Unknown";
    this.model = "Unknown";
    this.year = 0;
    this.fuelEfficiency = 0.0;
}
// Parameterized constructor
public Car(String make, String model, int year, double fuelEfficiency)
{
    this.make = make;
    this.model = model;
    this.year = year;
    this.fuelEfficiency = fuelEfficiency;
}
// Method to calculate the range of the car
public double calculateRange(double tankSize) {
    return fuelEfficiency * tankSize;
}}
```

```
Default Car:
Make: Unknown
Model: Unknown
Year: 0
Fuel Efficiency: 0.0 miles per gallon

A 2024 Tesla Roadster can travel approximately 7.5 miles on a full tank.
```

```
class CarDemo {
    public static void main(String[] args) {
        // Using the no-argument constructor
        Car defaultCar = new Car();
        System.out.println("Default Car:");
        System.out.println("Make: " + defaultCar.make);
        System.out.println("Model: " + defaultCar.model);
        System.out.println("Year: " + defaultCar.year);
        System.out.println("Fuel Efficiency: " +
defaultCar.fuelEfficiency + " miles per gallon");

        // Using the parameterized constructor
        Car myCar = new Car("Tesla", "Roadster", 2024,
0.5);
        double tankSize = 15.0; // Size of the fuel tank in
gallons
         double range = myCar.calculateRange(tankSize);
        System.out.println("\nA " + myCar.year + " " +
myCar.make + " " + myCar.model +  " can travel
approximately " + range + " miles on a full tank.");
    }
}
```

The this Keyword

- **this keyword** allows a method to refer to the object that invoked it.
- **this** can be used inside any method to refer to the current object.
- **this** is always a reference to the object on which the method was invoked.
- use this anywhere as a reference to an object of the current class' type.

```
// no name space collision but this is used as redundant
 public Car(String a, String b, int c, double d) {
 this.make = a;
 this.model = b;
 this.year = c;
 this.fuelEfficiency = d; }
```

```
// no name space collision so no need of this operator
 public Car(String a, String b, int c, double d) {
 make = a;
 model = b;
 year = c;
 fuelEfficiency = d; }
```

this keyword:Instance Variable Hiding

- It is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes.
- There can be Overlap of local variables, including formal parameters to methods with the names of the class' instance variables.
- However, when a local variable has the same name as an instance variable, **the local variable hides the instance variable**.
- this directly refers to the object,
- use this to resolve any namespace collisions that might occur between instance variables and local variables.

```
// use this operator to avoid name space collision
 public Car(String make, String model, int year, double fuelEfficiency) {
this.make = make;
 this.model = model;
.year = year;
this.fuelEfficiency = fuelEfficiency; }
```

```
public class Stack {
    private char[] vowel;
    private int top;
    private static final int INITIAL_CAPACITY = 10; // Initial
public Stack() {
        vowel = new char[INITIAL_CAPACITY];
        top = -1;
    }
    public void push(char c) {
        if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u' ||
            c == 'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U') {
            if (top == vowel.length - 1) {
                System.out.println("Stack is full");
            }
            vowel[++top] = c;
        } else {
            System.out.println(c + " is not a vowel.");
        }       }
    public Character pop() {
        if (top == -1) {
            System.out.println("Stack is empty.");
            return null;
        }
        return vowel[top--];       }
```

```
// Method to peek the top vowel of the stack
    public Character peek() {
        if (top == -1) {
            System.out.println("Stack is empty.");
            return null;
        }
        return vowel[top];
    }

public static void main(String[] args) {
        Stack vowelStack = new Stack();
        vowelStack.push('a');
        vowelStack.push('b');   // Not a vowel
        vowelStack.push('e');
        vowelStack.push('i');
        System.out.println("Top of the stack: " + vowelStack.peek());
        System.out.println("Popped: " + vowelStack.pop());
        System.out.println("Popped: " + vowelStack.pop());
        System.out.println("Popped: " + vowelStack.pop());
        System.out.println("Popped: " + vowelStack.pop());
    }
  }
```

```
b is not a vowel.
Top of the stack: i
Popped: i
Popped: e
Popped: a
Stack is empty.
Popped: null
```

Garbage collections

- In java, since objects are dynamically allocated by using the new operator, such objects are destroyed and their memory released for later reallocation automatically.
- In some languages, such as traditional C++, dynamically allocated objects must be manually released by use of a delete operator.
- It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.
- There is no need to explicitly destroy objects.
- Garbage collection only occurs sporadically (if at all) during the execution of your program.

CS6308- Java Programming

V P Jayachitra
Assistant Professor
Department of Computer Technology
MIT Campus
Anna University

MODULE II JAVA OBJECTS -1	L	T	P	EL
	3	0	4	3
Classes and Objects, Constructor, Destructor, Static instances, this, constants, Thinking in Objects, String class, Text I/O				
SUGGESTED ACTIVITIES : • Flipped classroom • Practical - Implementation of Java programs – using String class, Creating Classes and objects • EL – Thinking in Objects				
SUGGESTED EVALUATION METHODS: • Assignment problems • Quizzes				

Constructor

```
public class  Sample{
        public Sample(){   //constructor


        }
}
```

```
//Syntax constructor
accessModifier ClassName() {
   // Initialization code
}
```

No Return Type: no return type including void

Implicit Return: implicitly return reference of newly created object

Naming: Constructor name is same as classname

Access Modifiers: can be public , private, protoected

Constructor :Default Constructor

- **A default constructor is a no-argument constructor**
- **Default constructor is provided by the Java compiler if no other constructors are defined in the class**
- **Initializes member variables to default values**

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

   // Default constructor
     public MyClass() {
     }
}
```

Parameterized public constructor

```
public class MyClass {
   int value;


   //Parameterized  Public constructor
   public MyClass(int value) {
      this.value = value;
   }
}

public class Main {
   public static void main(String[] args) {
      MyClass obj = new MyClass(10);
      System.out.println(obj.value);
   }
}
```

10

- Constructors are intended to initialize instance variables when an object is created.
- Takes arguments that are used to set initial values for object instance variables

Parameterized public constructor

```
public class MyClass {
   int value;

 private static int staticVar; // Static variable

// Static initialization block
 static {
            staticVar = 30; }

   public MyClass(int value) { //Parameterized  Public constructor
      this.value = value;
   }
}
public class Main {
   public static void main(String[] args) {
      MyClass obj = new MyClass(10);
      System.out.println(obj.value);
      System.out.println(MyClass.staticVar);
   }
}
```

```
10
30
```

- Constructors are intended to initialize instance variables when an object is created.
- Constructors cannot directly initialize static variables.
- Takes arguments that are used to set initial values for object instance variables

Protected constructor

```
public class Base {
    int value;
    // Protected constructor
    protected Base() {
        value=10;
    }
}
public class Derived extends Base {
    public Derived() {
        super();            // Calls the protected constructor of Base
    }
}
public class Main {
    public static void main(String[] args) {
        Derived obj = new Derived();
        System.out.println(obj.value);
    }
}
```

10

- The constructor can be accessed within the same package and by subclasses

Private constructor

```
public class Singleton {
   private static Singleton instance;   //to hold single instance of the class
   // Private constructor
   private Singleton() {

   }
   // Static method to get the singleton instance
   public static Singleton getInstance() {
      if (instance == null) {
         instance = new Singleton();
      }
      return instance;
   }
}
public class Main {
   public static void main(String[] args) {
      Singleton obj = Singleton.getInstance(); // Access via static method
      System.out.println(obj);
   }
}
```

Singleton@15db9742

- private constructor ensures that no other class can directly create an instance of the class.

Default parametrized constructor

```
public class MyClass {
   int value;

   //Default parametrized constructor: i.e  no access modifier stated
    MyClass(int value) {
      this.value = value;
   }
}

public class Main {
   public static void main(String[] args) {
      MyClass obj = new MyClass(10);    // Valid within the same package
      System.out.println(obj.value);
}
}
```

10

- Constructors are intended to initialize instance variables when an object is created.
- Takes arguments that are used to set initial values for object instance variables

.

Default constructor

```
public class MyClass {
   int value;

   //Default parametrized constructor
    MyClass(int value) {
      this.value = value;
   }
}

public class Main {
   public static void main(String[] args) {
      //MyClass obj = new MyClass();  compilation error
      MyClass obj = new MyClass(10);    // Valid within the same package
      System.out.println(obj.value);
}
}
```

10

- Constructors are intended to initialize instance variables when an object is created.
- Takes arguments that are used to set initial values for object instance variables
- Once you define any parameterized constructor in a class, the default no-argument constructor is not automatically provided. If you need a no-argument constructor, you must explicitly define it alongside any parameterized constructors.

Default constructor

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

   //Default constructor
    MyClass(int value) {
      this.value = value;
   }
}

public class Main {
   public static void main(String[] args) {
      MyClass obj = new MyClass();
      MyClass obj = new MyClass(10);    // Valid within the same package
      System.out.println(obj.value);
}
}
```

10

- Constructors are intended to initialize instance variables when an object is created.
- Takes arguments that are used to set initial values for object instance variables
- Once you define any parameterized constructor in a class, the default no-argument constructor is not automatically provided. If you need a no-argument constructor, you must explicitly define it alongside any parameterized constructors.

Copy constructor

```
public class MyClass {
   int num;
   public MyClass() {
    }
   // Copy constructor
   public MyClass(MyClass other) {
      this.num = other.num;
   }
   public static void main(String[] args) {
      MyClass original = new MyClass();
      original.num=10;

      MyClass copy = new MyClass(original);

      System.out.println("Original object:");
     System.out.println (original.num)

      System.out.println("Copied object:");
    System.out.println (copy.num)    } }
```

```
Original object:
10
Copied object:
10
```

- A copy constructor is used to create a new object as a copy of an existing object.

this

- ‘this’ is a keyword used as a special reference variable that refers to the current object of the class
- Used to refer current object instance variable within the class.
  - Distinguishes the instance variable from parameters with the same name
  - Avoids ambuigty

```
public class Student {
    private String name,regno; // Instance variables
    // Default constructor
    public Student() { // Initialize with default values
        this.name = "Unknown";
        this.regno = "0000";      }
    public Student(String name, String regno) { // Parameterized constructor
        this.name = name;
        this.regno = regno;      }
    public String getDetails() {
        return "Name: " + this.name + ", Registration Number: " + this.regno;      }
    public void setDetails(String name, String regno) {
        this.name = name;
        this.regno = regno;      }
    public static void main(String[] args) {
        // Creating two Student objects
        Student student1 = new Student();
        Student student2 = new Student("Mark Reinhold", "MR002");
        // Setting details for both students
        student1.setDetails("James Gosling", "JG001");
        // Printing details of both students
        System.out.println("Student 1 Details: " + student1.getDetails());
        System.out.println("Student 2 Details: " + student2.getDetails());
    }   }
```

```
Student 1 Details: Name: James Gosling, Registration Number: JG001
Student 2 Details: Name: Mark Reinhold, Registration Number: MR002
```

```
public class Student {
    private String name,regno; // Instance variables
    // Default constructor
    public Student() { // Initialize with default values
            name = "Unknown";
            regno = "0000";      }
    public Student(String name, String regno) { // Parameterized constructor
            name = name; //initialize without using this
            regno = regno;      }
    public String getDetails() {
       return "Name: " + this.name + ", Registration Number: " + this.regno;      }
    public void setDetails(String name, String regno) {
       this.name = name;
       this.regno = regno;      }
    public static void main(String[] args) {
       // Creating two Student objects
       Student student1 = new Student();
       Student student2 = new Student("Mark Reinhold", "MR002");
       // Setting details for both students
       student1.setDetails("James Gosling", "JG001");
       // Printing details of both students
       System.out.println("Student 1 Details: " + student1.getDetails());
       System.out.println("Student 2 Details: " + student2.getDetails());
    }   }
```

```
Student 1 Details: Name: James Gosling, Registration Number: JG001
Student 2 Details: Name: null, Registration Number: null
```

this

- ‘this’ is a keyword used as a special reference variable that refers to the current object of the class
- Used to invoke current class method

```
public class Counter {
   private int count; // Instance variable to store the counter value
   public Counter() {// Constructor
      this.count = 0; }
  public void increment() {// Method to increment the counter
      this.count++;
      this.displayCount(); } // Call displayCount method using 'this'
  public void decrement() {// Method to decrement the counter
      this.count--; // Decrement the count
      this.displayCount(); } // Call displayCount method using 'this'
  public void displayCount() {// Method to display the current count
      System.out.println("Current count: " + this.count);
   }
   public static void main(String[] args) {
     Counter counter = new Counter();
      counter.increment(); // Increment count to 1
      counter.increment(); // Increment count to 2
      counter.decrement(); // Decrement count to 1
      counter.decrement(); // Decrement count to 0
   }
}
```

```
Current count: 1
Current count: 2
Current count: 1
Current count: 0
```

```
public class Counter{
   private int count; // Instance variable to store the counter value
   public Counter() {// Constructor
      this.count = 0; }
   public void increment() {// Method to increment the counter
      this.count++;
      displayCount(); } //  same as using this. displayCount()
   public void decrement() {// Method to decrement the counter
      this.count--; // Decrement the count
      displayCount(); } // same as using this. displayCount()
   public void displayCount() {// Method to display the current count
      System.out.println("Current count: " + this.count);
   }
   public static void main(String[] args) {
      Counter counter = new Counter();
      counter.increment(); // Increment count to 1
      counter.increment(); // Increment count to 2
      counter.decrement(); // Decrement count to 1
      counter.decrement(); // Decrement count to 0
   }
}
```

```
Current count: 1
Current count: 2
Current count: 1
Current count: 0
```

```
public class Car{
   private String model;

   public Car setModel(String model) {
      this.model = model;
      return this; // Returns the current object for method chaining
   }

   public void showModel() {
      System.out.println("Model: " + this.model); // Refers to the instance variable
   }

  public static void main(String[] args) {
         new Car().setModel("Tesla").showModel(); // Method chaining
      }
   }
```

```
Model: Tesla
```

this

- ‘this’ is a keyword used as a special reference variable that refers to the current object of the class
- Used to invoke current class constructor
  - Used to reuse the constructor
  - Constructor chaining

```
public class Car{
    private String model;
    public Car() {  // Default constructor
        this("Unknown"); // Calls the parameterized constructor with a default model
    }
    public Car(String model) {  // Parameterized constructor
        this.model = model;     // Sets the model
    }
    public Car setModel(String model) {
        this.model = model;
        return this;        // Returns the current object for method chaining
    }
    public void showModel() {
        System.out.println("Model: " + this.model); // Refers to the instance variable
    }
    public static void main(String[] args) {
        // Using the parameterized constructor
        Car car1 = new Car("Tesla");
        car1.showModel();       // Output: Model: Tesla
        // Using the default constructor and then setting the model
        Car car2 = new Car();     // Calls the default constructor
        //car2.setModel("BMW");    // Sets the model
        car2.showModel();        // Output: Model: BMW ?
    }   }
```

```
Model: Tesla
Model: Unknown
```

```
public class Student {
    private String name,regno; // Instance variables
    public Student() { // Default constructor
       name = "Unknown";
       regno = "0000";       }
    public Student(String name) { // Parameterized constructor
       this.name = name;
    }
    public Student(String name, String regno) { // Parameterized constructor
       this(name); //reusing constructor from the constructor
       this.regno = regno;
    }
    public String getDetails() {
       return "Name: " + this.name + ", Registration Number: " + this.regno;
    }
    public static void main(String[] args) {
       Student student1 = new Student();
       Student student2 = new Student("James Gosling");
       Student student3 = new Student("Mark Reinhold", "MR002");
       // Printing details of both students
       System.out.println("Student 1 Details: " + student1.getDetails());
       System.out.println("Student 2 Details: " + student2.getDetails());
       System.out.println("Student 2 Details: " + student3.getDetails());      }
  }
```

```
Student 1 Details: Name: Unknown, Registration Number: 0000
Student 2 Details: Name: James Gosling, Registration Number: null
Student 2 Details: Name: Mark Reinhold, Registration Number: MR002
```

```
public class Student {
    private String name,regno; // Instance variables
    public Student() { // Default constructor
       name = "Unknown";
       regno = "0000";       }
    public Student(String name) { // Parameterized constructor
       this.name = name;
    }
    public Student(String name, String regno) { // Parameterized constructor
      this.regno = regno;
      this(name); //reusing constructor from the constructor
    }
    public String getDetails() {
       return "Name: " + this.name + ", Registration Number: " + this.regno;
    }
    public static void main(String[] args) {
       Student student1 = new Student();
       Student student2 = new Student("James Gosling");
       Student student3 = new Student("Mark Reinhold", "MR002");
       // Printing details of both students
       System.out.println("Student 1 Details: " + student1.getDetails());
       System.out.println("Student 2 Details: " + student2.getDetails());
       System.out.println("Student 2 Details: " + student3.getDetails());      }
  }
```

```
Student 1 Details: Name: Unknown, Registration Number: 0000
Student 2 Details: Name: James Gosling, Registration Number: null
Student 2 Details: Name: Mark Reinhold, Registration Number: MR002
```

```
public class Student {
    private String name,regno; // Instance variables
    public Student() { // Default constructor
       name = "Unknown";
       regno = "0000";       }
    public Student(String name) { // Parameterized constructor
       this.name = name;
    }
    public Student(String name, String regno) { // error
      this.regno = regno;
      this(name); // this()should be the first statement
    }
    public String getDetails() {
       return "Name: " + this.name + ", Registration Number: " + this.regno;
    }
    public static void main(String[] args) {
       Student student1 = new Student();
       Student student2 = new Student("James Gosling");
       Student student3 = new Student("Mark Reinhold", "MR002");
       // Printing details of both students
       System.out.println("Student 1 Details: " + student1.getDetails());
       System.out.println("Student 2 Details: " + student2.getDetails());
       System.out.println("Student 2 Details: " + student3.getDetails());      }
  }
```

```
//compile time error
java: call to this must be first statement in constructor
```

```
public class Student {
    private String name,regno; // Instance variables
    public Student() { // Default constructor
       name = "Unknown";
       regno = "0000";       }
    public Student(String name) { // Parameterized constructor
       this.name = name;
    }
    public Student(String name, String regno) { // Parameterized constructor
       this();   //calling default constructor
    }
    public String getDetails() {
       return "Name: " + this.name + ", Registration Number: " + this.regno;
    }
    public static void main(String[] args) {
       Student student1 = new Student();
       Student student2 = new Student("James Gosling");
       Student student3 = new Student("Mark Reinhold", "MR002");
       // Printing details of both students
       System.out.println("Student 1 Details: " + student1.getDetails());
       System.out.println("Student 2 Details: " + student2.getDetails());
       System.out.println("Student 2 Details: " + student3.getDetails());      }
  }
```

```
Student 1 Details: Name: Unknown, Registration Number: 0000
Student 2 Details: Name: James Gosling, Registration Number: null
Student 2 Details: Name: Unknown, Registration Number: 0000
```

```
class Person {
   Student obj;

   Person(Student obj) {
      this.obj = obj;
   }

   public void printStudentDetails() {
      System.out.println(obj.name);
   }
}

   class Student {

   public String name="James"; // Instance variables
   public Student(){
      Person objP=new Person(this);
      objP.printStudentDetails();
   }

   public static void main(String[] args) {
     Student student = new Student();
       }
}
```

James

this

- ‘this’ is a keyword used as a special reference variable that refers to the current object of the class
- Used to return current class instance

```
class Student {

  public String name; // Instance variables
  public Student(String name){
     this.name=name;
  }
  public Student sendObject(){
     return this;
  }

  public String toString(){
        return "Name::" + name;
  }
  public static void main(String[] args) {
    Student obj=new Student("Jayachitra");
     System.out.println(obj.sendObject());
      }
}
```

```
Name::Jayachitra
```

```
class Student {

  public String name; // Instance variables
  public Student(String name){
     this.name=name;
  }
  public void printObject(){
     System.out.println(this);
  }


  public static void main(String[] args) {
    Student obj=new Student("Jayachitra");
     System.out.println(obj);
     obj.printObject();
      }
}
```

```
Student@e9e54c2
Student@e9e54c2
```

```
class Sample {
   int x;
   Sample(int x) {
      this.x = x;            }
   void modify(Sample obj) {
      obj.x = 100;    }
   void reassign(Sample obj) {
      obj = new Sample(200);     }
}
public class Main {
   public static void main(String[] args) {
      Sample original = new Sample(10);
      System.out.println(" x before modify: " + original.x);
      original.modify(original);
      System.out.println("x after modify: " + original.x);
      original.reassign(original);
      System.out.println("x after reassign: " + original.x);
   }
}
```

```
class Sample {
   int x;
   Sample(int x) {
      this.x = x;
   }
   void modify(Sample obj) {
      obj.x = 100;    }
   void reassign(Sample obj) {
      obj = new Sample(200);      }
}
public class Main {
   public static void main(String[] args) {
      Sample original = new Sample(10);
      System.out.println(" x before modify: " + original.x);
      original.modify(original);
      System.out.println(" x after modify: " + original.x);
      original.reassign(original);
      System.out.println("x after reassign: " + original.x);
   }
}
```

```
class Sample {
   int x;
   Sample(int x) {
      this.x = x;            }
   void modify(Sample obj) {
      obj.x = 100;     }
   void reassign(Sample obj) {
      obj = new Sample(200);      }
}
public class Main {
   public static void main(String[] args) {
      Sample original = new Sample(10);
      System.out.println(" x before modify: " + original.x);
      original.modify(original);
      System.out.println("x after modify: " + original.x);
      original.reassign(original);
      System.out.println("x after reassign: " + original.x);
   }
}
```

```
x before modify: 10
x after modify: 100
x after reassign: 100
```

```
#include <iostream>
class Sample {
public:
   int x;
   Sample(int x) : x(x) {}
   void modify(Sample& obj) {
      obj.x = 100;     }
   void reassign(Sample*& obj) {
      obj = new Sample(200);    }};
int main() {
   Sample original(10);
   std::cout << " x before modify: " << original.x << std::endl;
   original.modify(original);
   std::cout << "x after modify: " << original.x << std::endl;
   Sample* originalPtr = &original;
   original.reassign(originalPtr);
   std::cout << "x after reassign: " << original.x << std::endl;
   delete originalPtr;
   return 0;
}
```

```
x before modify: 10
x after modify: 100
x after reassign: 200
```

CS6308- Java Programming

V P Jayachitra
Assistant Professor
Department of Computer Technology
MIT Campus
Anna University

MODULE II JAVA OBJECTS -1	L	T	P	EL
	3	0	4	3
Classes and Objects, Constructor, Destructor, Static instances, this, constants, Thinking in Objects, String class, Text I/O				
SUGGESTED ACTIVITIES : • Flipped classroom • Practical - Implementation of Java programs – using String class, Creating Classes and objects • EL – Thinking in Objects				
SUGGESTED EVALUATION METHODS: • Assignment problems • Quizzes				

Static

- **Why static member?**
  - A class member must be accessed only in conjunction with an object of its class.
  - Is it possible to create a member that can be used by itself, without reference to a specific instance?
  - Yes, static class member will be used independently of any object of that class.

What is static member?

- A member that is declared static can be accessed before any objects of its class are created, and without reference to any object.
- To create such a member, precede its declaration with the keyword static.
- Both methods and variables can be declared to be static.
- Example : main()method
  - main( ) is declared as static
  - main() should be called before any objects exist.

Static variables

- Instance variables declared as static are, essentially, **global variables**.
- When objects of its class are declared, all instances of the class share the same static variable.
- Static variables are used independently of any object using class name followed by the dot operator.
  - classname.staticVariable

Static methods

- Methods declared as static are, essentially, **global methods**.
- Methods declared as static have several restrictions:
  - They can **only directly call other static methods** of their class.
  - They can **only directly access static variables** of their class.
  - **They cannot refer to this or super** in any way. (super relates to inheritance).
  - They are used **independently of any object** using class name followed by the dot operator.
    - classname.method( )

Static block

- Static block is used to initialize your static variables.

- Static block gets executed exactly once, when the class is first loaded.

```
class StaticDemo {
    //static variables
    static int a = 42;
    static int b = 99;
    //static method
    static void callme() {
        System.out.println("a = " + a);
    }
}

class StaticByName {
    public static void main(String args[]) {
        StaticDemo.callme();
        System.out.println("b = " + StaticDemo.b);
    }
}
```

```
a = 42
b = 99
```

```
public class StaticBlockExample {
    // Static block
    static {
        System.out.println("Static block is executed.");
    }

    // Constructor
    public StaticBlockExample() {
        System.out.println("Constructor is executed.");
    }

    // Static variable
    static int staticVariable = initializeStaticVariable();

    // Static method
    static int initializeStaticVariable() {
        System.out.println("Static variable initializer is executed.");
        return 10;
    }
public static void main(String[] args) {
        System.out.println("Main method is executed.");
        StaticBlockExample obj = new StaticBlockExample();
    }
}
```

```
Static block is executed.
Static variable initializer is executed.
Main method is executed.
Constructor is executed.
```

- The **static block** is used to initialize static variables and runs only once when the class is first loaded.
- Static variables are initialized after the static block.
- The main method is executed after the static initialization phase, and the **constructor** is executed

```java
public class Final {
    public void final_varaible_method(){
        final int a=1;
        System.out.println(a++);
    }

    public static void main(String []argh){
        Final f=new Final();
        f.final_varaible_method();
    }
}
```

java: cannot assign a value to final variable a

```java
// Demonstrate static variables, methods, and blocks.
class UseStatic {
  //static variables
    static int a = 3;
    static int b;
 //static method
    static void meth(int x) {
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }
//static block
    static {
        System.out.println("Static block initialized.");
        b = a * 4;
    }
    public static void main(String args[]) {
        meth(42);
    }
}
```

```
a = 3
b = 12
```

Introducing Nested and Inner Classes

- **Define a class within another class and such classes are known as nested classes.**
- The scope of a nested class is bounded by the scope of its enclosing class.
  - class B defined within class A does not exist independently of A.
- A nested class has access to the members, including private members, of the class in which it is nested.
- The **enclosing class does not have access to the members of the nested class.**
- A nested class that is declared directly within its enclosing class scope is a member of its enclosing class.
- A nested class also be declared that is local to a block.

```
public class EnclosingClass {
  . . .
  public class NestedClass {
  . . .  }
}
```

Introducing Nested and Inner Classes

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
    static class StaticNestedClass {
        ...
    }
}
```

- A **nested class** is a member of its enclosing class.
- Non-static nested classes (**inner classes**) have access to other members of the enclosing class, even if they are declared private.
- **Static nested classes** do not have access to other members of the enclosing class.
- As a member of the OuterClass, a nested class can be declared private, public, protected

Why nested class?

- A nested class is a class that’s tightly coupled with the class in which it’s defined.

- A **nested class has access to the private data** within its enclosing class

Why Use Nested Classes?

- Compelling reasons for using nested classes include the following:
  - It is a way of logically grouping classes that are only used in one place:
    - If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together
  - It increases encapsulation:
    - Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private.
    - By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.
  - It can lead to more readable and maintainable code:
    - Nesting small classes within top-level classes places the code closer to where it is used.

Nested classes-static nested class

- There are two types of nested classes:
  - **Static**
  - **non-static.**

- A static nested class is one that has the static modifier applied.
- Static class must access the non-static members of its enclosing class through an object.
- Static class cannot refer to non-static members of its enclosing class directly.
  - Because of this restriction, static nested classes are seldom used
- **Note:** Top level class can not be declared as static. Only nested classes can be declared as static.

Nested classes-Inner class

- A non-static nested class is an inner class.
- Inner class has access to all of the variables and methods of its outer class.
- Inner class refer variables and methods directly
- in the same way that other non-static members of the outer class do.

```
class OuterClass {
...
 class InnerClass {
 ...
 }
}
```

To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:

OuterClass outerObject = new OuterClass();
OuterClass.InnerClass innerObject = outerObject.new InnerClass();

```
public class Manager extends Employee {
  private DirectReports directReports;
  public Manager() {
    this.directReports = new DirectReports();
  }
  . . .
  private class DirectReports {
  . . .
  }
}
```

Creating DirectReports instances: First attempt

Creating DirectReports instances: Second attempt

```
public class Manager extends Employee {
  public Manager() {
  }
  . . .
  public class DirectReports {
  . . .
  }
  }
public static void main(String[] args) {
  Manager manager = new Manager();
  Manager.DirectReports dr = manager.new DirectReports();
}
```

```
public class Manager extends Employee {
  public Manager() {
  }
  . . .
  public class DirectReports {
  . . .
  }
}
public static void main(String[] args) {
  Manager.DirectReports dr = new Manager.DirectReports();// This won't
work!
```

```
public class OuterClass {

    String outerField = "Outer field";
    static String staticOuterField = "Static outer field";

    class InnerClass {
        void accessMembers() {
            System.out.println(outerField);
            System.out.println(staticOuterField);
        }
    }

    static class StaticNestedClass {
        void accessMembers(OuterClass outer) {
 // Compiler error: Cannot make a static reference to the non-
static
            //     field outerField
            // System.out.println(outerField);
            System.out.println(outer.outerField);
            System.out.println(staticOuterField);
        }
    }
```

```
 public static void main(String[] args) {
        System.out.println("Inner class:");
        System.out.println("------------");
        OuterClass outerObject = new OuterClass();
        OuterClass.InnerClass innerObject = outerObject.new
InnerClass();
        innerObject.accessMembers();

        System.out.println("\nStatic nested class:");
        System.out.println("--------------------");
        StaticNestedClass staticNestedObject = new
StaticNestedClass();
        staticNestedObject.accessMembers(outerObject);

        System.out.println("\nTop-level class:");
        System.out.println("--------------------");
        TopLevelClass topLevelObject = new TopLevelClass();
        topLevelObject.accessMembers(outerObject);
    }
```

```
Inner class:
------------
Outer field
Static outer field

Static nested class:
--------------------
Outer field
Static outer field
```

Shadowing

```
public class ShadowTest {

    public int x = 0;

    class FirstLevel {

        public int x = 1;

        void methodInFirstLevel(int x) {
            System.out.println("x = " + x);
            System.out.println("this.x = " + this.x);
            System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);
        }
    }

    public static void main(String... args) {
        ShadowTest st = new ShadowTest();
        ShadowTest.FirstLevel fl = st.new FirstLevel();
        fl.methodInFirstLevel(23);
    }
}
```

```
x = 23
this.x = 1
ShadowTest.this.x = 0
```

1.What is a nested class?

1. A class that provides some utility to other classes in the application
2. A class that is defined within another class
3. A class where one or more interface references are passed into its constructor
4. None of the above

1.What is a nested class?

1. A class that provides some utility to other classes in the application
2. **A class that is defined within another class**
3. A class where one or more interface references are passed into its constructor
4. None of the above

1.Why might you use a nested class?

1. When you need to define a class that is tightly coupled (functionally) with another class
2. When you want to write a class that makes use of a large number of interfaces from the JDK
3. When one class needs to access to another class private data, but you have exceeded the maximum number of allowable classes for your application
4. For a class has more than 20 methods defined on it and should be refactored

1.Why might you use a nested class?

1. **When you need to define a class that is tightly coupled (functionally) with another class**
2. When you want to write a class that makes use of a large number of interfaces from the JDK
3. When one class needs to access to another class private data, but you have exceeded the maximum number of allowable classes for your application
4. For a class has more than 20 methods defined on it and should be refactored

```
public class Outer {

 private static final Logger log = Logger.getLogger(Outer.class.getName());

 public void setInner(Inner inner) {
 this.inner = inner;
 }
 private Inner inner;

 public Inner getInner() {
 return inner;
 }

 private class Inner {
 }
  public static void main(String[] args) {
 Outer outer = new Outer();
 Inner inner = new Outer.Inner();
 outer.setInner(inner);
 log.info("Outer/Inner: " + outer.hashCode() + "/" + outer.getInner().hashCode());
 }

}
```

1.The class name `Outer` is not a legal Java class name.
2.The class name `Inner` is confusing.
3.The main method defined in class `Outer` has the wrong method signature.
4.The `log.info()` call in main() has too many parameters.
5.The class body for class `Inner` is empty.
6.None of the above. The line Inner inner = new Outer.Inner(); is in error. This is not the correct syntax for instantiating a nested class using an enclosing class reference. Rather, it should be: Inner inner = outer.new Inner();

```
public class Outer {

 private static final Logger log = Logger.getLogger(Outer.class.getName());

 public void setInner(Inner inner) {
 this.inner = inner;
 }
 private Inner inner;

 public Inner getInner() {
 return inner;
 }

 private class Inner {
 }
  public static void main(String[] args) {
 Outer outer = new Outer();
 Inner inner = new Outer.Inner();
 outer.setInner(inner);
 log.info("Outer/Inner: " + outer.hashCode() + "/" + outer.getInner().hashCode());
 }

}
```

1. The class name `Outer` is not a legal Java class name.
2. The class name `Inner` is confusing.
3. The main method defined in class `Outer` has the wrong method signature.
4. The `log.info()` call in main() has too many parameters.
5. The class body for class `Inner` is empty.
6. None of the above. The line Inner inner = new Outer.Inner(); is in error. This is not the correct syntax for instantiating a nested class using an enclosing class reference. Rather, it should be: Inner inner = outer.new Inner();

1.Is it possible to write an inner class that can be instantiated by any class in your application (regardless of what package in which it resides) without an enclosing instance of the outer class? Explain your answer.

1.Is it possible to write an inner class that can be instantiated by any class in your application (regardless of what package in which it resides) without an enclosing instance of the outer class? Explain your answer. **Yes, it is. The inner class should be declared with the static modifier, along with public visibility. Then any class in your application can instantiate the inner class without an enclosing outer instance of the class.**

1.What’s the difference between a nested class an an inner class?

1. An inner class is one that is defined using private access only, whereas a nested class is declared static.
2. The two terms are synonymous and can be used interchangeably.
3. A nested class must reside inside the enclosing class and be declared before any of the enclosing class’s variables.
4. A nested class is defined in main(), whereas an inner class can be defined anywhere.
5. There is no such thing as a nested class.
6. None of the above.

1.What’s the difference between a nested class an an inner class?

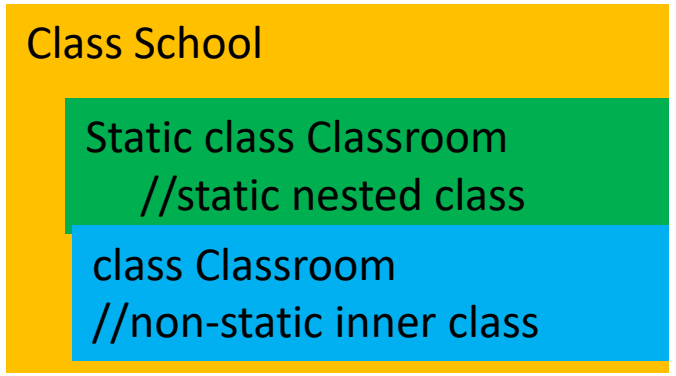
1. An inner class is one that is defined using private access only, whereas a nested class is declared static.
2. **The two terms are synonymous and can be used interchangeably.**
3. A nested class must reside inside the enclosing class and be declared before any of the enclosing class’s variables.
4. A nested class is defined in main(), whereas an inner class can be defined anywhere.
5. There is no such thing as a nested class.
6. None of the above.

1. Which of these statements is true regarding inner classes?
    1. An inner class can access any private data variables of its enclosing class unless it is declared `static`.
    2. An inner class must be declared `public` to be instantiated by any other class than its enclosing class.
    3. A static inner class is not allowed except under special circumstances.
    4. An inner class is completely invisible to its enclosing class.
    5. None of the above.

1. Which of these statements is true regarding inner classes?
    1. **An inner class can access any private data variables of its enclosing class unless it is declared `static`.**
    2. An inner class must be declared `public` to be instantiated by any other class than its enclosing class.
    3. A static inner class is not allowed except under special circumstances.
    4. An inner class is completely invisible to its enclosing class.
    5. None of the above.

```
public class School {
    private String schoolName;
    private Classroom[] classrooms;
    private Student[] students;
    private int classroomCount = 0;
    private int studentCount = 0;
    public School(String schoolName, int maxClassrooms, int maxStudents) {
        this.schoolName = schoolName;
        this.classrooms = new Classroom[maxClassrooms];
        this.students = new Student[maxStudents];
    }

// Static nested class                    // print student details
    public static class Classroom {
        private String roomNumber;
        private String subject;
         public Classroom(String roomNumber, String subject) {
            this.roomNumber = roomNumber;
            this.subject = subject;                                    }
        public String getRoomNumber() {
            return roomNumber;        }
        public String getSubject() {
            return subject;                }
    }
```



```
// Non-static inner class
    public class Student {
        private String name;
        private String grade;

     public Student(String name, String grade) {
            this.name = name;
            this.grade = grade;    }

    public String getName() {
            return name;       }

     public String getGrade() {
            return grade;        }

     public void printStudentDetails() {// print student details
            System.out.println("Student Name: " + name + ", Grade: " + grade + ", School Name: " + schoolName);          }       }

     public void addClassroom(Classroom classroom) {// add a classroom
       if (classroomCount < classrooms.length) {
            classrooms[classroomCount++] = classroom;
       } else {
            System.out.println("No more space to add classrooms.");
       }     }
```

```
// add a student
   public void addStudent(Student student) {
      if (studentCount < students.length) {
         students[studentCount++] = student;
      } else {
         System.out.println("No more space to add students.");
      }
   }
   // print all classrooms
   public void printAllClassrooms() {
      System.out.println("Classrooms:");
      for (int i = 0; i < classroomCount; i++) {
         Classroom classroom = classrooms[i];
         System.out.println("Room Number: " +
classroom.getRoomNumber() + ", Subject: " +
classroom.getSubject());
      }
   }
// print all students
   public void printAllStudents() {
      System.out.println("Students:");
      for (int i = 0; i < studentCount; i++) {
         students[i].printStudentDetails();
      }    }
```

```
   public static void main(String[] args) {
       School school = new School(“MIT", 5, 10);
      // Add classrooms
  school.addClassroom(new School.Classroom("101","Maths"));
school.addClassroom(new School.Classroom("102", "Science"));

      // Add students
      School.Student st1 = school.new Student("ARUN", "A");
      School.Student st2 = school.new Student(“BANU", "B");
      school.addStudent(st1);
      school.addStudent(st2);

      // Print all classrooms
      school.printAllClassrooms();

      // Print all students
      school.printAllStudents();
   } }
```

What is a static nested class and how is it different from an inner class?

```
class OuterClass {

static class StaticNestedClass {
  void display() {
        System.out.println("This is a static nested class");
    }
 }

 class InnerClass {
   void display() {
      System.out.println("This is an inner class");
    }
 }

 }
```

```java
class Calculator {
    static int baseValue = 10;

    static class Adder {
        int add(int x) {
            return x + baseValue;  // Directly accessing static member of outer class
        }
    }
}
```

```java
class Container {
    static class StaticNested {
        static int staticVar = 5;
        static void staticMethod() {
            System.out.println("Static method in static nested class");
        }
    }
}
```

```java
class Outer {
    class Inner {
        void innerMethod() {
            System.out.println("Inner class method");
        }
    }
}
```

```java
Outer outer = new Outer();
Outer.Inner inner = outer.new Inner();
inner.innerMethod();
```

```java
class ShadowExample {
    private int x = 10;

    class InnerShadow {
        private int x = 20;

        void printBoth() {
            System.out.println("Inner x: " + x);
            System.out.println("Outer x: " + ShadowExample.this.x);
        }
    }
}
```

CS6308- Java Programming

V P Jayachitra
Assistant Professor
Department of Computer Technology
MIT Campus
Anna University

MODULE II JAVA OBJECTS -1	L	T	P	EL
	3	0	4	3
Classes and Objects, Constructor, Destructor, Static instances, this, constants, Thinking in Objects, String class, Text I/O				
SUGGESTED ACTIVITIES : • Flipped classroom • Practical - Implementation of Java programs – using String class, Creating Classes and objects • EL – Thinking in Objects				
SUGGESTED EVALUATION METHODS: • Assignment problems • Quizzes				

String handling in java

Strings

- In Java a string is a sequence of characters.
- In Java a string is an Object
  - Other languages that implement strings as character arrays
- Strings are Immutable.
  - String object that is created cannot be changed.
- However, a variable declared as a String reference can be changed to point at some other String object at any time.

Strings

- String can be created using three string classes namely String, StringBuffer and StringBuilder
- Use the class called StringBuffer to perform changes in original strings.
- String, StringBuilder and StringBuffer classes are declared final and there cannot be subclasses of these classes.
- The String, StringBuffer, and StringBuilder classes are defined in java.lang.
- Java.lang is the default package in java
- Java.lang is automatically imported without needing to explicitly import classes from this package
    - Common classes in lang are String, Math, System, Object, Thread

Creating Strings

- The default constructor creates an empty string object.
  - String s = new String();
- Create string object that have initial values from a character array
  - String s= new String(char[] chars)
- Create string object using String literals
  - String s = "String Literal";
- **Examples:**

```
String str = "abc";
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

- Construct a string object by passing another string object.
  - String(String strObj)
  - Example: String str2 = new String(str);

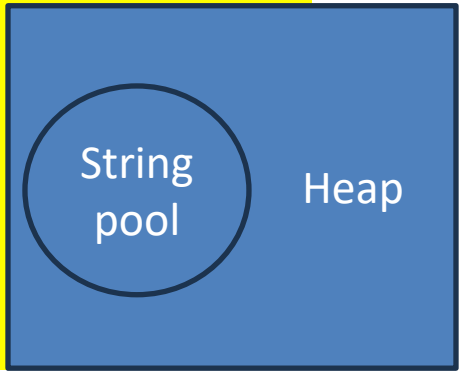
String memory

- The string pool or string constant pool, is a special area of the heap where Java stores unique string literals.
- String pool helps in saving memory and improving performance by avoiding duplicate strings.
- The heap is the runtime data area from which memory for all class instances and arrays is allocated.
- Strings created using the new keyword are allocated on the heap

```
public class StringExample {
public static void main(String[] args) {
    String str1 = new String("hello"); // String created in the heap
    String str2 = "hello"; // string literal created in string pool of heap
    String str3 = "hello"; // Reuses the interned string literal
System.out.println(str1 == str2);
System.out.println(str1 == str3);
System.out.println(str1.equals(str2));
System.out.println(str1.equals(str3));

}}
```

```
false
true
true
```



```java
public class StringExample {
    public static void main(String[] args) {
        String stringLiteral="Java";  //String literals
        String stringObject=new String( original: "Java"); //String Object
        String stringObject1=new String(stringLiteral);  //passing String object
        char[] chararray={'J','a','v','a'};
        String stringObject2=new String(chararray);  //passing String object
        //String stringLiteral2=chararray; error java: incompatible types: char[] cannot be convert
        String stringLiteral2=stringObject1;
        String stringLiteral3=stringObject;
        System.out.println("Address of stringObject1: " + System.identityHashCode(stringObject1));
        stringObject1=stringLiteral;
        System.out.println("Address of stringObject1: " + System.identityHashCode(stringObject1));
        System.out.println(stringLiteral.equals(stringObject2));
        System.out.println(stringLiteral.equals(chararray));
        System.out.println("Address of stringLiteral: " + System.identityHashCode(stringLiteral));
        System.out.println("Address of stringLitera2: " + System.identityHashCode(stringLiteral2));
        System.out.println("Address of stringLitera3: " + System.identityHashCode(stringLiteral3));
        System.out.println("Address of stringObject: " + System.identityHashCode(stringObject));
        System.out.println("Address of stringObject1: " + System.identityHashCode(stringObject1));
        System.out.println("Address of stringObject2: " + System.identityHashCode(stringObject2));
    }
}
```

```
Address of stringObject1: 245257410
Address of stringObject1: 1023892928
true
false
Address of stringLiteral: 1023892928
Address of stringLitera2: 245257410
Address of stringLitera3: 558638686
Address of stringObject: 558638686
Address of stringObject1: 1023892928
Address of stringObject2: 1149319664
```

Since char is not String type and hence equals method return false

Creating Strings

- To specify a subrange of a character array as an initializer using the following constructor:
  - String(char chars[ ], int startIndex, int numChars)
- Here, startIndex specifies the index at which the subrange begins, and numChars specifies the number of characters to use.
- **Examples:**

```
char arr[] = {'J', 'A', 'V', 'A'};
String str = new String(arr,2,1);
String str11 = new String(arr,2,2);
System.out.println(arr);
System.out.println(str);
System.out.println(str11);
```

```
JAVA
V
VA
```

Creating Strings

- String class provides constructors that initialize a string when given a byte array.
  - String(byte chrs[ ])
  - String(byte chrs[ ], int startIndex, int numChars)
    - Here, chrs specifies the array of bytes. The second form allows you to specify a subrange.
  - **Examples:**

```
class PrintStringValues{
    public static void main(String args[]) {
        byte[] values = {65, 66,67, 68};
        String fullValues = new String(values);
        System.out.println("Full value " + fullValues);
        String partialValue = new String(values, 1, 3);
        System.out.println("Partial values: " + partialValue);
    }
}
```

Full value ABCD
Partial values: BCD

String METHODS

- int length( )
  - The length() method returns the length of the string.

    Eg: System.out.println(“Hello”.length()); // prints 5

```
char vowels[] = { 'a', 'e', 'i', 'o', 'u' };
String vowelString = new String(vowels);
System.out.println("Number of vowels: " +
vowelString.length());
```

- The + operator is used to concatenate two or more strings.

  Eg: String name = “Harry”

  String str = “Name : ” + name+ “.”;

- Java compiler converts an operand to a String whenever the other operand of the + is a String object.

String Concatenation with Other Data Types

```
double temperature = 25.5;
String weather = "The temperature is " + temperature + " degrees Celsius.";
System.out.println(weather);

String result = "Sum: " + (10 + 20);
System.out.println(result);
// output
// Sum: 30
// rather than
// Sum: 1020

String resultWithParentheses = "Sum: " + ((10 + 20));
// Now resultWithParentheses contains the string "Sum: 30".
```

String Conversion and toString( )

- To determine the string representation for objects of classes that is created.
- Classes that is created has to override toString( ) and provide your own string representations.
- The toString( ) method has this general form:
  - **String toString( )**
  - can be used in print( ) and println( ) statements and in concatenation expressions.

String toString()

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

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

    public String toString() {
        return year + " " + make + " " + model;
    }
}
class CarDemo {
    public static void main(String args[]) {
        Car myCar = new Car("Toyota", "Corolla", 2022);
        String carDescription = "My car: " + myCar;

        System.out.println(myCar); // implicitly calls toString()
        System.out.println(carDescription);
    }
}
```

```
2022 Toyota Corolla
My car: 2022 Toyota Corolla
```

Character Extraction

Character Extraction

- The String class provides a number of ways in which characters can be extracted from a String object.
- The characters that comprise a string within a String object cannot be indexed as if they were a character array.
- Many of the String methods employ an index (or offset) into the string for their operation.
- Like arrays, the string indexes begin at zero.

Character Extraction

- public char **charAt**(int INDEX)
  - Returns the character at the specified index.
  - INDEX is the index of the character that is to be to obtained.
  - An index ranges from 0 to length() - 1.

```
char ch;
ch = "XYZ".charAt(1); // ch = "Y"
```

- **Method `getChars`**

Get entire set of characters in `String`

void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)

s1.getChars(start, end, charArray, start);

compareTo( )

int compareTo(String str)

Here, str is the String being compared with the invoking String.

Character Extraction

```
public class StringExample2 {
    public static void main(String args[]) {
      //character extraction
      String email = "contact@example.com";
      int start=8;
      int end=15;
      char buffer[] = new char[end - start];
      email.getChars(start, end, buffer,0);
      System.out.println("Domain: " + new String(buffer));       }
    }
```

```
Domain: example
```

Searching Strings

int indexOf(int ch):

Finds the first occurrence of a character.

// Here ch is represented by its Unicode code point

// example '@' is represented by its Unicode code point 64

int indexOf(String str):

Finds the first occurrence of a substring.

int lastIndexOf(int ch):

Finds the last occurrence of a character.

int lastIndexOf(String str):

Finds the last occurrence of a substring.

int indexOf(int ch, int fromIndex):

Finds the first occurrence starting from a specified index.

int lastIndexOf(int ch, int fromIndex):

Finds the last occurrence searching backward from a specified index.

boolean contains(CharSequence sequence):

Checks if a substring is present.

boolean startsWith(String prefix):

Checks if the string starts with a specified prefix.

boolean endsWith(String suffix):

Checks if the string ends with a specified suffix.

```
public class StringExample3 {
    public static void main(String[] args) {
        String email = "contact@domain.com";
        System.out.println("email:"+ email);
        int atIndex = email.indexOf('@’); // 1. indexOf(int ch)
        System.out.println("Index of '@': " + atIndex);
        int domainIndex = email.indexOf("domain"); // 2. indexOf(String str)
        System.out.println("Index of 'domain': " + domainIndex);
        int lastDotIndex = email.lastIndexOf('.’); // 3. lastIndexOf(int ch)
        System.out.println("Last index of '.': " + lastDotIndex);
        int lastDomainIndex = email.lastIndexOf("domain"); // 4. lastIndexOf(String str)
        System.out.println("Last index of 'domain': " + lastDomainIndex);
        int atIndexAfter5 = email.indexOf('@', 5); // 5. indexOf(int ch, int fromIndex)
        System.out.println("Index of '@' after index 5: " + atIndexAfter5);
        int lastDotIndexBefore15 = email.lastIndexOf('.', 15); // 6. lastIndexOf(int ch, int fromIndex)
        System.out.println("Last index of '.' before index 15: " + lastDotIndexBefore15);
       boolean containsDomain = email.contains("domain"); // 7. contains(CharSequence seq)
        System.out.println("Contains 'domain': " + containsDomain);
        boolean startsWithContact = email.startsWith("contact"); // 8. startsWith(String prefix)
        System.out.println("Starts with 'contact': " + startsWithContact);
       boolean endsWithCom = email.endsWith(".com");    // 9. endsWith(String suffix)
        System.out.println("Ends with '.com': " + endsWithCom);
    }
}
```

```
email:contact@domain.com
Index of '@': 7
Index of 'domain': 8
Last index of '.': 14
Last index of 'domain': 8
Index of '@' after index 5: 7
Last index of '.' before index 15: 14
Contains 'domain': true
Starts with 'contact': true
Ends with '.com': true
```

```
class EmailExtractor {
    public static void main(String args[]) {
        String email = "contact@example.com";
        int atIndex = email.indexOf('@');
        int dotIndex = email.lastIndexOf('.');

        char domain[] = new char[dotIndex - atIndex - 1];

        email.getChars(atIndex + 1, dotIndex, domain, 0);
        System.out.println("Domain: " + new String(domain));
    }
}
```

```
Domain: example
```

```
public class StringSort {
    public static void main(String[] args) {
        String[] cskPlayers = {"Dhoni","Ruturaj","Stokes","Rachin","Ambati"};
        for(int j = 0; j < cskPlayers.length; j++) {
            for(int i = j + 1; i <cskPlayers.length; i++) {
               if(cskPlayers[i].compareToIgnoreCase(cskPlayers[j] )< 0) {
                  String t =cskPlayers[j];
                  cskPlayers[j] = cskPlayers[i];
                  cskPlayers[i] = t;
               }
            }
            System.out.println(cskPlayers[j]);
        }
    }
}
```

```
Ambati
Dhoni
Rachin
Ruturaj
Stokes
```

Character Extraction

- getBytes( )
- There is an alternative to getChars( ) that stores the characters in an array of bytes.
- This method is called getBytes( ), and it uses the default character-to-byte conversions provided by the platform.
- Here is its simplest form:
  - **byte[ ] getBytes( )**
- getBytes( ) is most useful when you are exporting a String value into an environment that does not support 16-bit Unicode characters.

Character Extraction

- toCharArray( )
- To convert all the characters in a String object into a character array, the easiest way is to call toCharArray( ).
- It returns an array of characters for the entire string.
- It has this general form:
  - **char[ ] toCharArray( )**
- This function is provided as a convenience, since it is possible to use getChars( ) to achieve the same result.

String Comparison

- **equals()** - Compares the invoking string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as the invoking object.

  public boolean **equals**(Object anObject)

- **equalsIgnoreCase()-** Compares this String to another String, ignoring case considerations.

  - When it compares two strings, it considers A-Z to be the same as a-z.
  - Two strings are considered equal ignoring case if they are of the same length, and corresponding characters in the two strings are equal ignoring case.

  public boolean **equalsIgnoreCase**(String anotherString)

String Comparison

- **regionMatches( )**
- The regionMatches( ) method compares a specific region inside a string with another specific region in another string.
- There is an overloaded form that allows you to ignore case in such comparisons.
- Here are the general forms for these two methods:

boolean regionMatches(int startIndex, String str, int strStartIndex, int numChars)	boolean regionMatches(boolean ignoreCase, int startIndex, String str, int strStartIndex, int numChars)

• For both versions, startIndex specifies the index at which the region begins within the invoking String object.
• The String being compared is specified by str.
•The index at which the comparison will start within str is specified by strStartIndex.
•The length of the substring being compared is passed in numChars.
•In the second version, if ignoreCase is true, the case of the characters is ignored. Otherwise, case is significant.

```
public class RegionMatchesExample {
    public static void main(String[] args) {
        String str1 = "Hello World";
        String str2 = "world";
        String str3 = "Hello";

        // Case-sensitive match
       // 'World' starting from index 5 of str1 compared to 'world'
        boolean result1 = str1.regionMatches(6, str2, 0, 5);
        System.out.println("Case-sensitive match: " + result1);

        // Case-insensitive match
        boolean result2 = str1.regionMatches(true, 6, str2, 0, 5);
     // 'World' starting from index 5 of str1 compared to 'world' ignoring case
        System.out.println("Case-insensitive match: " + result2);

        // Checking a specific region with exact length
        boolean result3 = str1.regionMatches(0, str3, 0, 5);
      // 'Hello' starting from index 0 of str1 compared to 'Hello'
        System.out.println("Exact length match: " + result3);     }
}
```

```
Case-sensitive match: false
Case-insensitive match: true
Exact length match: true
```

String Comparison

- **startsWith()** – Tests if this string starts with the specified prefix.

  public boolean **startsWith**(String prefix)

  “Figure”.startsWith(“Fig”); // true

- **endsWith()** - Tests if this string ends with the specified suffix.

  public boolean **endsWith**(String suffix)

  “Figure”.endsWith(“re”); // true

  - boolean startsWith(String str, int startIndex)
    - Example : "Foobar".startsWith("bar", 3) => returns true.

String Comparison

- **compareTo()** - Compares two strings.
  - A string is less than another if it comes before the other in dictionary order.
  - A string is greater than another if it comes after the other in dictionary order
  - The result is a negative integer if this String object lexicographically precedes the argument string.
  - The result is a positive integer if this String object lexicographically follows the argument string.
  - The result is zero if the strings are equal.
  - compareTo returns 0 exactly when the equals(Object) method would retu true.

public int **compareTo**(String anotherString) public int

compareToIgnoreCase(String str)

Modifying a String

- **substring()** - Returns a new string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string.

public String **substring**(int beginIndex)

Eg: "unhappy".substring(2)

returns "happy"

public String **substring**(int beginIndex, int endIndex)

Eg: "smiles".substring(1, 5)

returns "mile“

```
class CharReplace {
    public static void main(String args[]) {
        String org = "Hello, World! Hello, Java!";
        char search = 'o';
        char sub = '0';
        StringBuilder result = new StringBuilder();
        int i;

        do { // replace all matching characters
            System.out.println(org);
            i = org.indexOf(search);

            if(i != -1) {
                result = new StringBuilder(org.substring(0, i));
                result.append(sub);
                result.append(org.substring(i + 1));
                org = result.toString();
            }
        } while(i != -1);
    }
}
```

```
Hello, World! Hello, Java!
Hell0, World! Hello, Java!
Hell0, W0rld! Hello, Java!
Hell0, W0rld! Hell0, Java!
Hell0, W0rld! Hell0, Java!
```

String METHODS

Method call	Meaning
S2=s1.toLowerCase()	Convert string s1 to lowercase
S2=s1.toUpperCase()	Convert string s1 to uppercase
S2=s1.repalce(„x", „y")	Replace occurrence x with y
S2=s1.trim()	Remove whitespaces at the beginning and end of the string s1
S1.equals(s2)	If s1 equals to s2 return true
S1.equalsIgnoreCase(s2)	If s1==s2 then return true with irrespective of case of charecters
S1.length()	Give length of s1
S1.CharAt(n)	Give nth character of s1 string
S1.compareTo(s2)	If s1<s2 -ve no If s1>s2 +ve no If s1==s2 then 0
S1.concat(s2)	Concatenate s1 and s2
S1.substring(n)	Give substring staring from nth character

String Operations

- **concat()** - Concatenates the specified string to the end of this string.

- If the length of the argument string is 0, then this String object is returned.

- Otherwise, a new String object is created, containing theinvoking string with the contents of the str appended to it.

public String concat(String str)

"to".concat("get").concat("her")

returns "together"

String Operations

- replace()- Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

- public String **replace**(char oldChar, char newChar)

- “iam aq iqdiaq ” .replace(„q', „n’) //returns “I am an indian"

String Operations

- **trim()** - Returns a copy of the string, with leading and trailing whitespace omitted.

  public String trim()

```
String s="      Hi mom";
System.out.println(s);
System.out.println(s.trim());
```

```
      Hi mom
Hi mom
```

- **valueOf()** – Returns the string representation of the char array argument.

  public static String **valueOf**(char[] data)

String Operations

- **toLowerCase():** Converts all of the characters in a String to lower case.
- **toUpperCase():** Converts all of the characters in this String to upper case.

public String **toLowerCase**()

public String **toUpperCase**()

Eg: “HELLO YOU”.toLowerCase();

“hello you”.toUpperCase();

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

      char[] characters = {'a', '1', ' ', 'A', 'b', 'D', 'z', '@', '9', '1', '_'};
         for (char ch : characters) {
         System.out.println("Character: " + ch);

         System.out.println("Is Letter: " + Character.isLetter(ch));    // isLetter(char ch)

         System.out.println("Is Digit: " + Character.isDigit(ch));      // isDigit(char ch)

       System.out.println("Is Whitespace: "+Character.isWhitespace(ch)); // isWhitespace(char ch)

       System.out.println("Is Upper Case: " + Character.isUpperCase(ch)); // isUpperCase(char ch)

        System.out.println("Is Lower Case: " + Character.isLowerCase(ch)); // isLowerCase(char ch)

      System.out.println("To Upper Case: " + Character.toUpperCase(ch)); // toUpperCase(char ch)

      System.out.println("To Lower Case: " + Character.toLowerCase(ch)); // toLowerCase(char ch)

  System.out.println("Is Letter or Digit: " + Character.isLetterOrDigit(ch));// isLetterOrDigit(char ch)
}
   }
}
```

```
public class ValidationExample {
    public static void main(String[] args) {
                String phoneNumber = "+1234567890";
                validatePhoneNumber(phoneNumber); }
}
public static void validatePhoneNumber(
String phoneNumber) {
   // Remove non-numeric characters manually
   String cleanedNumber = "";
  for (char c : phoneNumber.toCharArray()) {
         if (Character.isDigit(c)) {
        cleanedNumber += c;
     }
      }
   // Check if cleaned number has exactly 10 digits
    if (cleanedNumber.length() == 10) {
         System.out.println("Phone number is valid.");
       }
else
{
         System.out.println("Invalid phone number. It should be exactly 10 digits long.");
}
 }
```

```
public static void validatePassword(String password) {
    if (password.length() < 8) {
        System.out.println("Invalid password. It must be at least 8 characters long.");
        return;
    }
    boolean hasUpperCase = false;       boolean hasLowerCase = false;
    boolean hasDigit = false;       boolean hasSpecialChar = false;
    boolean hasSpace = false;
    for (char c : password.toCharArray()) {
        if (Character.isUpperCase(c)) {
            hasUpperCase = true;
        } else if (Character.isLowerCase(c)) {
            hasLowerCase = true;
        } else if (Character.isDigit(c)) {
            hasDigit = true;
        } else if (c == '@' || c == '#'
|| c == '$' || c == '%' ) {
            hasSpecialChar = true;
        } else if (Character.isWhitespace(c)) {
            hasSpace = true;
            break;
        }
    }
```

```
 if (hasSpace) {
            System.out.println("Invalid password.");
        }
else if (hasUpperCase && hasLowerCase &&
hasDigit && hasSpecialChar) {
            System.out.println("Password is valid.");
        } else {
            System.out.println("Invalid password.");
        }
    }
}
}
```

Method	Description	Example
int codePointCount(int beginIndex, int endIndex)	Returns the number of Unicode code points in the specified text range.	int count = exampleString.codePointCount(0, 5); // Returns 5
int codePointBefore(int index)	Returns the Unicode code point before the specified index in the string.	int codePoint = exampleString.codePointBefore(6); // output 32(Unicode of space)
int codePointAt(int index)	Returns the Unicode code point at the specified index.	int codePoint = exampleString.codePointAt(5); // output: 32(space character)
int codePointCount(int beginIndex, int endIndex)	Returns the number of Unicode code points in the specified text range	int count = exampleString.codePointCount(0, 5); // Output: 5
boolean contentEquals(CharSequence str)	Compares the content of the string with the specified str.	boolean result = exampleString.contentEquals("Hello World"); // Output: true

Method	Description	Example
String format(Locale loc, String frmstr, Object... args)	Returns a formatted string using the specified loc, format string, and arguments.	String formatted = String.format(Locale.US, "Formatted example: %s", exampleString); //output: "Formatted example: Hello World"
boolean contains(CharSequence str)	Checks if the string contains the specified str of characters.	boolean result = exampleString.contains("World "); // Returns true
String format(String format, Object... args)	Returns a formatted string using the specified format string and arguments.	String formatted = String.format("Message: %s", "Hello World"); // Returns "Message: Hello World"
boolean isEmpty()	Checks if the string is empty (""). Returns true if the string is empty, otherwise false.	boolean result = "Hello World".isEmpty(); // Returns false

Method	Description	Example

Stream<String> lines()	Returns a stream of lines extracted from the string, split by line separators.	"Hello\nWorld".lines().forEach(System.out::println);
		// Output: "Hello"
		// Output: "World"

String replaceFirst(String regex, String replacement)	Replaces the first substring that matches the given regular expression with the specified replacement string.	String replaced = exampleString.replaceFirst("Hello", "Hi"); // Returns "Hi World, Hello Universe"
String replaceAll(String regex, String replacement)	Replaces all substrings that match the given regular expression with the specified replacement string.	String replaced = exampleString.replaceAll("Hell o", "Hi"); // Returns "Hi World, Hi Universe"
String[] split(String regex)	Splits the string around matches of the given regular expression and returns an array of substrings.	String[] parts = exampleString.split(" "); // Returns ["Hello", "World,", "Hello", "Universe"]

Wrapper class

- To handle primitive data types java support it by using wrapper class.
- **java** provides the mechanism *to convert primitive into object and object into primitive*.
- **autoboxing** and **unboxing** feature converts primitive into object and object into primitive automatically.
- The automatic conversion of primitive into object is known and autoboxing and vice-versa unboxing.

Example of wrapper class

```
public class Wrapper{
public static void main(String args[]){
//Converting int into Integer
 int k=20;
Integer i=new Integer(k); //converting int into Integer
Integer j=k;//autoboxing, compiler will write Integer.valueOf(a) internally
System.out.println(k+" "+i+" "+j);
}}
```

Output:

20 20 20