Java - What is OOP?

OOP stands for **Object-Oriented Programming**.

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

Java - What are Classes and Objects?

Classes and objects are the two main aspects of object-oriented programming.

So, a class is a template for objects, and an object is an instance of a class.

When the individual objects are created, they inherit all the variables and methods from the class.

Java Classes/Objects

Java is an object-oriented programming language.

Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.

A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword `class`:

Main.java

Create a class named "Main" with a variable x:

```
public class Main {
  int x = 5;
}
```

Create an Object

In Java, an object is created from a class. We have already created the class named `Main`, so now we can use this to create objects.

To create an object of `Main`, specify the class name, followed by the object name, and use the keyword `new`:

Example

Create an object called "`myObj`" and print the value of x:

Multiple Objects

You can create multiple objects of one class:

Example

Create two objects of `Sample`:

```
public class Sample {
  int x = 5;

  public static void main(String[] args) {
    Sample myObj1 = new Sample();  // Object 1
    Sample myObj2 = new Sample();  // Object 2
    System.out.println(myObj1.x);
    System.out.println(myObj2.x);
```

```
  }
}
```

Using Multiple Classes

You can also create an object of a class and access it in another class. This is often used for better organization of classes (one class has all the attributes and methods, while the other class holds the `main()` method (code to be executed)).

Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory/folder:

- First.java
- Second.java

First.java

```
public class First {

  int x = 5;

}
```

Second.java

```
class Second {

  public static void main(String[] args) {

    First myObj = new First();

    System.out.println(myObj.x);

  }

}
```

Accessing Attributes

You can access attributes by creating an object of the class, and by using the dot syntax (`.`):

The following example will create an object of the `Main` class, with the name `myObj`. We use the `x` attribute on the object to print its value:

Example

Create an object called "`myObj`" and print the value of `x`:

```
public class Sample {
  int x = 5;

  public static void main(String[] args) {
    Sample myObj = new Sample();
    System.out.println(myObj.x);
  }
}
```

Java Class Methods

Methods are declared within a class, and that they are used to perform certain actions:

Example

Create a method named `myMethod()` in Main:

```
public class Sample {
  static void myMethod() {
    System.out.println("Hello World!");
  }
}
```

`myMethod()` prints a text (the action), when it is **called**. To call a method, write the method's name followed by two parentheses **()** and a semicolon**;**

Example

Inside `main`, call `myMethod()`:

```
public class Sample {
  static void myMethod() {
    System.out.println("Hello World!");
  }

  public static void main(String[] args) {
    myMethod();
  }
}
// Outputs "Hello World!"
```

Static vs. Non-Static

You will often see Java programs that have either `static` or `public` attributes and methods.

In the example above, we created a `static` method, which means that it can be accessed without creating an object of the class, unlike `public`, which can only be accessed by objects:

Example

An example to demonstrate the differences between `static` and `public` **methods**:

```
public class Sample {
  // Static method
  static void myStaticMethod() {
    System.out.println("Static methods can be called without creating objects");
  }

  // Public method
  public void myPublicMethod() {
```

```
    System.out.println("Public methods must be called by creating
objects");
  }


  // Main method
  public static void main(String[] args) {
    myStaticMethod(); // Call the static method
    // myPublicMethod(); This would compile an error


    Sample myObj = new Sample(); // Create an object of Main
    myObj.myPublicMethod(); // Call the public method on the object
  }
}
```

Access Methods With an Object

Example

Create a Car object named `myCar`. Call the `move()` and `speed()` methods on the `myCar` object, and run the program:

```
// Create a Sample class
public class Sample {


  // Create a move() method
  public void move() {
    System.out.println("The car is going as fast as it can!");
  }


  // Create a speed() method and add a parameter
  public void speed(int maxSpeed) {
    System.out.println("Max speed is: " + maxSpeed);
```

```
  }


  // Inside main, call the methods on the myCar object
  public static void main(String[] args) {
    Sample myCar = new Sample();   // Create a myCar object
    myCar.move();      // Call the move() method
    myCar.speed(200);          // Call the speed() method
  }
}
// The car is going as fast as it can!
// Max speed is: 200
```

Using Multiple Classes

It is a good practice to create an object of a class and access it in another class.

Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory:

- First.java
- Second.java

First.java

```
public class First {
  public void move() {
    System.out.println("The car is going as fast as it can!");
  }


  public void speed(int maxSpeed) {
    System.out.println("Max speed is: " + maxSpeed);
```

```
  }
}
```

Second.java

```
class Second {
  public static void main(String[] args) {
    First myCar = new First();     // Create a myCar object
    myCar.move();      // Call the move() method
    myCar.speed(200);          // Call the speed() method
  }
}
```

Java Constructors

A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes:

Example

Create a constructor:

```
// Create a Main class
public class Sample {
  int x;  // Create a class attribute

  // Create a class constructor for the Main class
  public Sample () {
    x = 5;  // Set the initial value for the class attribute x
  }

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

```
    Sample myObj = new Sample(); // Create an object of class Main
(This will call the constructor)

    System.out.println(myObj.x); // Print the value of x

  }

}

// Outputs 5
```

Note that the constructor name must **match the class name**, and it cannot have a **return type** (like `void`).

Also note that the constructor is called when the object is created.

All classes have constructors by default: if you do not create a class constructor yourself, Java creates one for you. However, then you are not able to set initial values for object attributes.

Constructor Parameters

Constructors can also take parameters, which is used to initialize attributes.

The following example adds an `int y` parameter to the constructor. Inside the constructor we set x to y (x=y). When we call the constructor, we pass a parameter to the constructor (5), which will set the value of x to 5:

Example

```
public class Sample {

  int x;


  public Sample (int y) {

    x = y;

  }


  public static void main(String[] args) {

    Sample myObj = new Sample (5);

    System.out.println(myObj.x);

  }
```

```
}


// Outputs 5
```

You can have as many parameters as you want:

Example

```
public class Sample {
  int modelYear;
  String modelName;


  public Sample (int year, String name) {
    modelYear = year;
    modelName = name;
  }


  public static void main(String[] args) {
    Sample myCar = new Sample (1969, "Mustang");
    System.out.println(myCar.modelYear + " " + myCar.modelName);
  }
}


// Outputs 1969 Mustang
```

Java Modifiers

By now, you are quite familiar with the `public` keyword that appears in almost all of our examples:

```
public class Sample
```

The `public` keyword is an **access modifier**, meaning that it is used to set the access level for classes, attributes, methods and constructors.

We divide modifiers into two groups:

- **Access Modifiers** - controls the access level
- **Non-Access Modifiers** - do not control access level, but provides other functionality

Access Modifiers

For **classes**, you can use either `public` or *default*:

Modifier	Description
`public`	The class is accessible by any other class
default	The class is only accessible by classes in the same package. This is used when you don't specify a modifier.

For **attributes, methods and constructors**, you can use the one of the following:

Modifier	Description
`public`	The code is accessible for all classes
`private`	The code is only accessible within the declared class
default	The code is only accessible in the same package. This is used when you

	don't specify a modifier.
`protected`	The code is accessible in the same package and **subclasses**.

Non-Access Modifiers

For **classes**, you can use either `final` or `abstract`:

Modifier	Description
`final`	The class cannot be inherited by other classes
`abstract`	The class cannot be used to create objects (To access an abstract class, it must be inherited from another class

Final

If you don't want the ability to override existing attribute values, declare attributes as `final`:

Example

```
public class Sample {
  final int x = 10;
  final double PI = 3.14;

  public static void main(String[] args) {
    Sample myObj = new Sample ();
```

```
    myObj.x = 50; // will generate an error: cannot assign a value to a
final variable

    myObj.PI = 25; // will generate an error: cannot assign a value to
a final variable

    System.out.println(myObj.x);

  }

}
```

Static

A `static` method means that it can be accessed without creating an object of the class, unlike `public`:

Example

An example to demonstrate the differences
between `static` and `public` methods:

```
public class Sample{

  // Static method

  static void myStaticMethod() {

    System.out.println("Static methods can be called without creating
objects");

  }



  // Public method

  public void myPublicMethod() {

    System.out.println("Public methods must be called by creating
objects");

  }

  // Main method

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

```
    myStaticMethod(); // Call the static method
    // myPublicMethod(); This would output an error

    Sample myObj = new Sample(); // Create an object of Main
    myObj.myPublicMethod(); // Call the public method
  }
}
```

Abstract

An `abstract` method belongs to an `abstract` class, and it does not have a body. The body is provided by the subclass:

Example

```
// Code from filename: Main.java

// abstract class
abstract class Sample {
  public String fname = "John";
  public int age = 24;
  public abstract void study(); // abstract method
}


// Subclass (inherit from Main)
class Student extends Sample {
  public int graduationYear = 2018;
  public void study() { // the body of the abstract method is provided here
    System.out.println("Studying all day long");
  }
```

```
}

// End code from filename: Main.java



// Code from filename: Second.java

class Second {

  public static void main(String[] args) {

    // create an object of the Student class (which inherits attributes
and methods from Main)

    Student myObj = new Student();


    System.out.println("Name: " + myObj.fname);

    System.out.println("Age: " + myObj.age);

    System.out.println("Graduation Year: " + myObj.graduationYear);

    myObj.study(); // call abstract method
  }

}
```

Encapsulation

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

- declare class variables/attributes as `private`
- provide public **get** and **set** methods to access and update the value of a `private` variable

Get and Set

You learned from the previous chapter that `private` variables can only be accessed within the same class (an outside class has no access to it). However, it is possible to access them if we provide
public **get** and **set** methods.

The `get` method returns the variable value, and the `set` method sets the value.

Syntax for both is that they start with either `get` or `set`, followed by the name of the variable, with the first letter in upper case:

Example

```
public class Person {
  private String name; // private = restricted access


  // Getter
  public String getName() {
    return name;
  }
  // Setter
  public void setName(String newName) {
    this.name = newName;
  }
}
```

Example explained

The `get` method returns the value of the variable `name`.

The `set` method takes a parameter (`newName`) and assigns it to the `name` variable. The `this` keyword is used to refer to the current object.

However, as the `name` variable is declared as `private`, we **cannot** access it from outside this class:

we use the `getName()` and `setName()` methods to access and update the variable:

Example

```
public class Sample {
```

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

    Person myObj = new Person();

    myObj.setName("John"); // Set the value of the name variable to
"John"

    System.out.println(myObj.getName());

  }

}


// Outputs "John"
```

Why Encapsulation?

- Better control of class attributes and methods
- Class attributes can be made **read-only** (if you only use the `get` method), or **write-only** (if you only use the `set` method)
- Flexible: the programmer can change one part of the code without affecting other parts
- Increased security of data

Java Packages

A package in Java is used to group related classes. Think of it as **a folder in a file directory**. We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:

- Built-in Packages (packages from the Java API)
- User-defined Packages (create your own packages)

Built-in Packages

The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.

The library contains components for managing input, database programming, and much much more. The complete list can be found at Oracles website: https://docs.oracle.com/javase/8/docs/api/.

The library is divided into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

To use a class or a package from the library, you need to use the `import` keyword:

Syntax

```
import package.name.Class;   // Import a single class

import package.name.*;   // Import the whole package
```

Import a Class

If you find a class you want to use, for example, the `Scanner` class, **which is used to get user input**, write the following code:

Example

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

In the example above, `java.util` is a package, while `Scanner` is a class of the `java.util` package.

To use the `Scanner` class, create an object of the class and use any of the available methods found in the `Scanner` class documentation. In our example, we will use the `nextLine()` method, which is used to read a complete line:

Example

Using the `Scanner` class to get user input:

```
import java.util.Scanner;


class MyClass {

  public static void main(String[] args) {

    Scanner myObj = new Scanner(System.in);

    System.out.println("Enter username");
```

```
    String userName = myObj.nextLine();

    System.out.println("Username is: " + userName);
  }
}
```

Import a Package

There are many packages to choose from. In the previous example, we used the `Scanner` class from the `java.util` package. This package also contains date and time facilities, random-number generator and other utility classes.

To import a whole package, end the sentence with an asterisk sign (`*`). The following example will import ALL the classes in the `java.util` package:

Example

```
import java.util.*;
```

User-defined Packages

To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders on your computer:

Example

```
└── root
  └── mypack
    └── MyPackageClass.java
```

To create a package, use the `package` keyword:

MyPackageClass.java

```
package mypack;

class MyPackageClass {

  public static void main(String[] args) {

    System.out.println("This is my package!");

  }
```

```
}
```

Java Inheritance

In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

- **subclass** (child) - the class that inherits from another class
- **superclass** (parent) - the class being inherited from

To inherit from a class, use the `extends` keyword.

In the example below, the `Car` class (subclass) inherits the attributes and methods from the `Vehicle` class (superclass):

Example

```
class Vehicle {
  protected String brand = "Ford";         // Vehicle attribute
  public void honk() {                     // Vehicle method
    System.out.println("Tuut, tuut!");
  }
}
class Car extends Vehicle {
  private String modelName = "Mustang";    // Car attribute
  public static void main(String[] args) {

    // Create a myCar object
    Car myCar = new Car();

    // Call the honk() method (from the Vehicle class) on the myCar
object
    myCar.honk();
```

```
    // Display the value of the brand attribute (from the Vehicle
class) and the value of the modelName from the Car class

    System.out.println(myCar.brand + " " + myCar.modelName);

  }

}
```

Java Polymorphism

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous chapter; **Inheritance** lets us inherit attributes and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a superclass called `Animal` that has a method called `animalSound()`. Subclasses of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

Example

```
class Animal {

  public void animalSound() {

    System.out.println("The animal makes a sound");

  }

}


class Pig extends Animal {

  public void animalSound() {

    System.out.println("The pig says: wee wee");

  }

}
```

```
class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}
```

Now we can create `Pig` and `Dog` objects and call the `animalSound()` method on both of them:

Example

```
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}
class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}
class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}
class Sample {
  public static void main(String[] args) {
    Animal myAnimal = new Animal();  // Create a Animal object
    Animal myPig = new Pig();  // Create a Pig object
```

```
    Animal myDog = new Dog();  // Create a Dog object
    myAnimal.animalSound();
    myPig.animalSound();
    myDog.animalSound();
  }
}
```

Abstract Classes and Methods

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.
Abstraction can be achieved with either **abstract classes** or **interfaces** (which you will learn more about in the next chapter).

The `abstract` keyword is a non-access modifier, used for classes and methods:

- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

An abstract class can have both abstract and regular methods:

```
abstract class Animal {
  public abstract void animalSound();
  public void sleep() {
    System.out.println("Zzz");
  }
}
```

From the example above, it is not possible to create an object of the Animal class:

```
Animal myObj = new Animal(); // will generate an error
```

To access the abstract class, it must be inherited from another class. Let's convert the Animal class we used in the Polymorphism chapter to an abstract class:

Remember from the Inheritance chapter that we use the `extends` keyword to inherit from a class.

Example

```
// Abstract class
abstract class Animal {
  // Abstract method (does not have a body)
  public abstract void animalSound();
  // Regular method
  public void sleep() {
    System.out.println("Zzz");
  }
}
// Subclass (inherit from Animal)
class Pig extends Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
}
class Sample {
  public static void main(String[] args) {
    Pig myPig = new Pig(); // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
```

```
}
```

Interfaces

Another way to achieve abstraction in Java, is with interfaces.

An `interface` is a completely "**abstract class**" that is used to group related methods with empty bodies:

Example

```
// interface
interface Animal {
  public void animalSound(); // interface method (does not have a body)
  public void run(); // interface method (does not have a body)
}
```

To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class with the `implements` keyword (instead of `extends`). The body of the interface method is provided by the "implement" class:

Example

```
// Interface
interface Animal {
  public void animalSound(); // interface method (does not have a body)
  public void sleep(); // interface method (does not have a body)
}

// Pig "implements" the Animal interface
class Pig implements Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
```

```
    System.out.println("The pig says: wee wee");
  }
  public void sleep() {
    // The body of sleep() is provided here
    System.out.println("Zzz");
  }
}


class Sample {
  public static void main(String[] args) {
    Pig myPig = new Pig();  // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Multiple Interfaces

To implement multiple interfaces, separate them with a comma:

Example

```
interface FirstInterface {
  public void myMethod(); // interface method
}


interface SecondInterface {
  public void myOtherMethod(); // interface method
}
```

```
class DemoClass implements FirstInterface, SecondInterface {
  public void myMethod() {
    System.out.println("Some text..");
  }
  public void myOtherMethod() {
    System.out.println("Some other text...");
  }
}
```

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

Java Exceptions

When executing Java code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, Java will normally stop and generate an error message. The technical term for this is: Java will throw an **exception** (throw an error).

Java try and catch

The `try` statement allows you to define a block of code to be tested for errors while it is being executed.

The `catch` statement allows you to define a block of code to be executed, if an error occurs in the try block.

The `try` and `catch` keywords come in pairs:

Syntax

```
try {
  //  Block of code to try
}
catch(Exception e) {
  //  Block of code to handle errors
}
```

Consider the following example:

This will generate an error, because **myNumbers[10]** does not exist.

```
public class Sample {
  public static void main(String[ ] args) {
    int[] myNumbers = {1, 2, 3};
    System.out.println(myNumbers[10]); // error!
  }
}
```

The output will be something like this:

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 10
        at Main.main(Main.java:4)
```

If an error occurs, we can use `try...catch` to catch the error and execute some code to handle it:

Example

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

```
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      System.out.println("Something went wrong.");
    }
  }
}
```

The output will be:

```
Something went wrong.
```

Finally

The `finally` statement lets you execute code, after `try...catch`, regardless of the result:

Example

```
public class Sample {
  public static void main(String[] args) {
    try {
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      System.out.println("Something went wrong.");
    } finally {
      System.out.println("The 'try catch' is finished.");
    }
  }
```

```
}
```

The output will be:

```
Something went wrong.
The 'try catch' is finished.
```

The throw keyword

The `throw` statement allows you to create a custom error.

The `throw` statement is used together with an **exception type**. There are many exception types available in Java: `ArithmeticException`, `FileNotFoundException`, `ArrayIndexOutOfBoundsException`, `SecurityException`, etc:

Example

Throw an exception if **age** is below 18 (print "Access denied"). If age is 18 or older, print "Access granted":

```
public class Main {

  static void checkAge(int age) {

    if (age < 18) {

      throw new ArithmeticException("Access denied - You must be at least 18 years old.");

    }

    else {

      System.out.println("Access granted - You are old enough!");

    }

  }

  public static void main(String[] args) {

    checkAge(15); // Set age to 15 (which is below 18...)

  }

}
```

The output will be:

```
Exception in thread "main" java.lang.ArithmeticException: Access
denied - You must be at least 18 years old.
        at Main.checkAge(Main.java:4)
        at Main.main(Main.java:12)
```

If **age** was 20, you would **not** get an exception:

Example

```
checkAge(20);
```

The output will be:

```
Access granted - You are old enough!
```

Java String

In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string. For example:

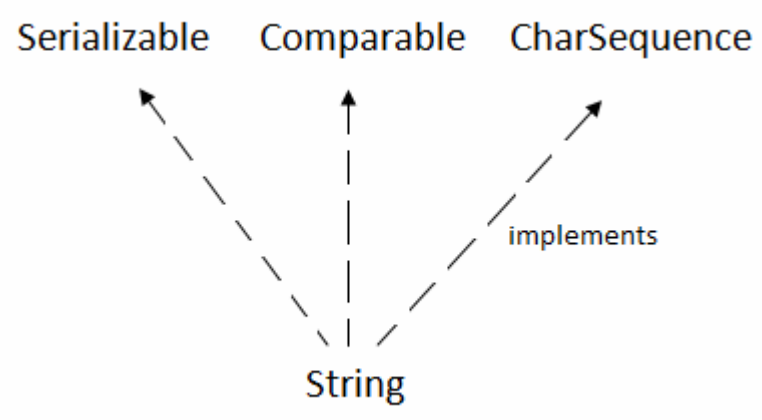
1. **char**[] ch={'j','a','v','a','t','p','o','i','n','t'};
2. String s=**new** String(ch);

is same as:

1. String s="rogersoft";

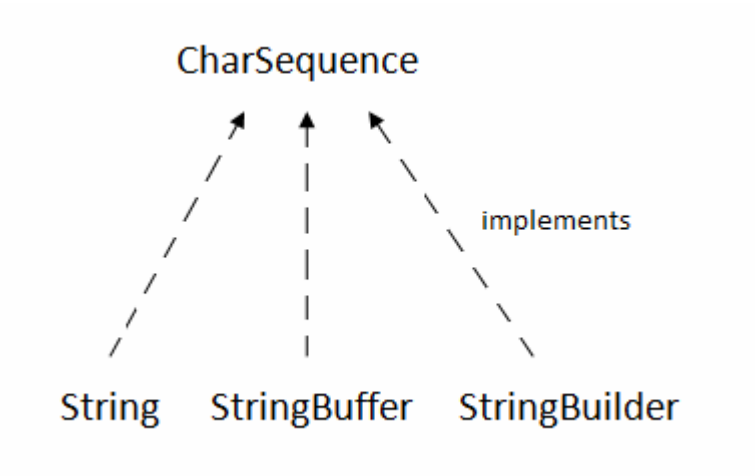
Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces.



CharSequence Interface

The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in Java by using these three classes.



The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

We will discuss immutable string later. Let's first understand what String in Java is and how to create the String object.

What is String in Java?

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

How to create a string object?

There are two ways to create String object:

1. By string literal
2. By new keyword

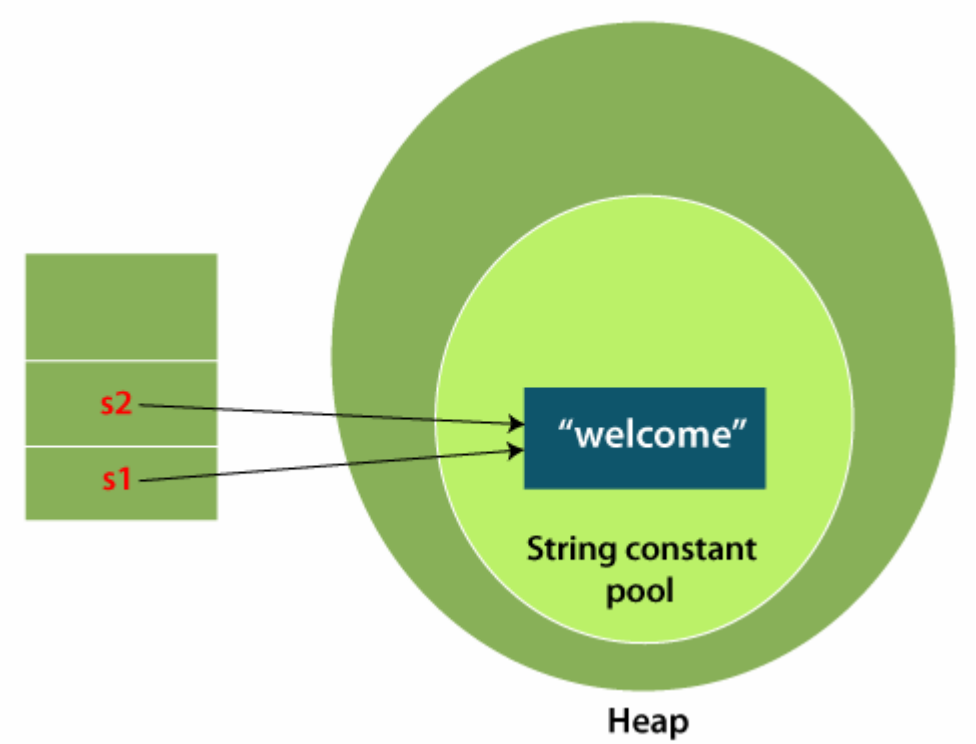
1) String Literal

Java String literal is created by using double quotes. For Example:

```
1. String s="welcome";
```

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

1. String s1="Welcome";
2. String s2="Welcome";//It doesn't create a new instance



In the above example, only one object will be created. Firstly, JVM will not find any string object with the value "Welcome" in string constant pool that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

Note: String objects are stored in a special memory area known as the "string constant pool".

Why Java uses the concept of String literal?

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

2) By new keyword

1. String s=**new** String("Welcome");//creates two objects and one reference variable

In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

Java String Example

StringExample.java

```
public class StringExample{
public static void main(String args[]){
String s1="java";//creating string by Java string literal
char ch[]={'s','t','r','i','n','g','s'};
String s2=new String(ch);//converting char array to string
String s3=new String("example");//creating Java string by new keyword
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}}
```

Java String charAt()

The **Java String class charAt()** method returns *a char value at the given index number*.

The index number starts from 0 and goes to n-1, where n is the length of the string. It returns **StringIndexOutOfBoundsException,** if the given index number is greater than or equal to this string length or a negative number.

```
public class CharAtExample{
public static void main(String args[]){
String name="rogersoft";
char ch=name.charAt(10);//returns the char value at the 10th index
System.out.println(ch);
}}
```

Java String concat

The **Java String class concat()** method *combines specified string at the end of this string*. It returns a combined string. It is like appending another string.

```
public class ConcatExample{
public static void main(String args[]){
String s1="java string";
// The string s1 does not get changed, even though it is invoking the method
// concat(), as it is immutable. Therefore, the explicit assignment is required here.
```

```
s1.concat("is immutable");
System.out.println(s1);
s1=s1.concat(" is immutable so assign it explicitly");
System.out.println(s1);
}}
```

Java String contains()

The **Java String class contains()** method searches the sequence of characters in this string. It returns *true* if the sequence of char values is found in this string otherwise returns *false*.

```
class ContainsExample{
public static void main(String args[]){
String name="what do you know about me";
System.out.println(name.contains("do you know"));
System.out.println(name.contains("about"));
System.out.println(name.contains("hello"));
}}
```

Java String equals()

The **Java String class equals()** method compares the two given strings based on the content of the string. If any character is not matched, it returns false. If all characters are matched, it returns true.

The String equals() method overrides the equals() method of the Object class.

```
public class EqualsExample{
public static void main(String args[]){
String s1="rogersoft";
String s2="rogersoft";
String s3="ROGERSOFT";
String s4="python";
System.out.println(s1.equals(s2));//true because content and case is same
System.out.println(s1.equals(s3));//false because case is not same
System.out.println(s1.equals(s4));//false because content is not same
}}
```

Java String join()

The **Java String class join()** method returns a string joined with a given delimiter. In the String join() method, the delimiter is copied for each element. The join() method is included in the Java string since JDK 1.8.

```
public class StringJoinExample{
public static void main(String args[]){
String joinString1=String.join("-","welcome","to","rogersoft");
System.out.println(joinString1);
}}
```

Java String length()

The **Java String class length()** method finds the length of a string. The length of the Java string is the same as the Unicode code units of the string.

```
public class LengthExample{
public static void main(String args[]){
String s1="rogersoft";
String s2="python";
System.out.println("string length is: "+s1.length());//10 is the length of rogersoft string

System.out.println("string length is: "+s2.length());//6 is the length of python string
}}
```

Java String substring()

The **Java String class substring()** method returns a part of the string.

We pass beginIndex and endIndex number position in the Java substring method where beginIndex is inclusive, and endIndex is exclusive. In other words, the beginIndex starts from 0, whereas the endIndex starts from 1.

```
public class SubstringExample{
public static void main(String args[]){
String s1="rogersoft";
System.out.println(s1.substring(2,4));//returns va
System.out.println(s1.substring(2));//returns vatpoint
}}
```

Java String toLowerCase()

The **java string toLowerCase()** method returns the string in lowercase letter. In other words, it converts all characters of the string into lower case letter.

```
public class StringLowerExample{
public static void main(String args[]){
String s1="ROGERSOFT HELLO stRIng";
String s1lower=s1.toLowerCase();
System.out.println(s1lower);
}}
```

Java String toUpperCase()

The **java string toUpperCase()** method returns the string in uppercase letter. In other words, it converts all characters of the string into upper case letter.

```
public class StringUpperExample{
public static void main(String args[]){
String s1="hello string";
String s1upper=s1.toUpperCase();
System.out.println(s1upper);
}}
```

Java String trim()

The **Java String class trim()** method eliminates leading and trailing spaces. The Unicode value of space character is '\u0020'. The trim() method in Java string checks this Unicode value before and after the string, if it exists then the method removes the spaces and returns the omitted string.

```
public class StringTrimExample{
public static void main(String args[]){
String s1="  hello string   ";
System.out.println(s1+"rogersoft");//without trim()
System.out.println(s1.trim()+"rogersoft");//with trim()
}}
```

Java String split()

The **java string split()** method splits this string against given regular expression and returns a char array.

```
public class SplitExample{
```

```
public static void main(String args[]){
String s1="java string split method by rogersoft";
String[] words=s1.split("\\s");//splits the string based on whitespace
//using java foreach loop to print elements of string array
for(String w:words){
System.out.println(w);
}
}}
```