# 

# An incubated co-working space for technology innovation

## Internal Document Template

**Working Document**

**Table of Contents**

[1 Document Overview 1](#_Toc414904876)

[1.1 Introduction and Purpose of this Document 1](#_Toc414904877)

[1.2 Scope 1](#_Toc414904878)

[2 Section Heading 2](#_Toc414904879)

[2.1 Sub Section 1 2](#_Toc414904880)

[2.2 Sub Section 2 2](#_Toc414904881)

# Document Change Log

This is a working document, which will be maintained with time. Team members, please ensure that any changes are recorded in the change log below – this is to ensure that each team member is always clear about which changes have been made and when.

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Author** | **Description** |
| 0.01 | 21 March 2015 | Malusi Gcakasi | Created initial template document. |
|  |  |  |  |

# Definition of Terms

|  |  |
| --- | --- |
| **Term** | **Definition** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# Team Register

|  |  |
| --- | --- |
| **Member Name** | **Role Description** |
| Akona Ntantiso | Leader |
| Unathi Waphi | Secretary |
| Nontokozo Ntuka | Time keeper |

# Description of Acronyms

|  |  |
| --- | --- |
| **Acronym** | **Description** |
| CH | The Cortex Hub |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# Document Overview

## Introduction and Purpose of this Document

This document gives a short overview of actions taken by team Ambicious (Akona Ntantiso, Unathi Waphi, Nontokozo Ntuka) the Remote Team (RT) and Central Team (CT) for the past week. The actions described herein will cover only those things that have been done and have verifiable artefacts produced by each team. The onus will be on each team member to describe the artefact produced by their work, and to describe the work carried out.

## Scope

Tell exactly what this document does and what it does not do. Can be reused.

# Java SE 7

## What is java?

Java is a programming language and computing platform first released by Sun Microsystems in 1995. There are lots of applications and websites that will not work unless you have Java installed, and more are created every day. Java is fast, secure, and reliable. From laptops to datacenters, game consoles to scientific supercomputers, cell phones to the Internet, Java is everywhere!

### Is Java free to download?

Yes, Java is free to download.

## Object-Oriented Programming Concepts

If you've never used an object-oriented programming language before, you'll need to learn a few basic concepts before you can begin writing any code. This lesson will introduce you to objects, classes, inheritance, interfaces, and packages. Each discussion focuses on how these concepts relate to the real world, while simultaneously providing an introduction to the syntax of the Java programming language.

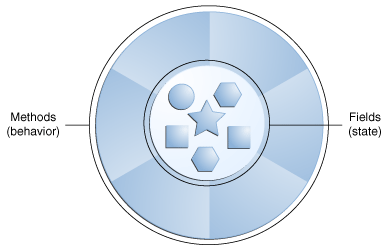
### What Is an Object?

An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life. This lesson explains how state and behavior are represented within an object, introduces the concept of data encapsulation, and explains the benefits of designing your software in this manner.

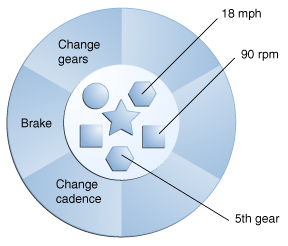
They all have state and behavior. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

Take a minute right now to observe the real-world objects that are in your immediate area. For each object that you see, ask yourself two questions: "What possible states can this object be in?" and "What possible behavior can this object perform?". Make sure to write down your observations. As you do, you'll notice that real-world objects vary in complexity; your desktop lamp may have only two possible states (on and off) and two possible behaviors (turn on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and behavior (turn on, turn off, increase volume, decrease volume, seek, scan, and tune). You may also notice that some objects, in turn, will also contain other objects. These real-world observations all translate into the world of object-oriented programming.

**A software object.**



**A bicycle modeled as a software object.**

****

By attributing state (current speed, current pedal cadence, and current gear) and providing methods for changing that state, the object remains in control of how the outside world is allowed to use it. For example, if the bicycle only has 6 gears, a method to change gears could reject any value that is less than 1 or greater than 6.

Bundling code into individual software objects provides a number of benefits, including:

**Modularity**: The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.

**Information-hiding**: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.

**Code re-use**: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.

**Pluggability and debugging ease**: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace it, not the entire machine

### What Is a Class?

A class is a blueprint or prototype from which objects are created. This section defines a class that models the state and behavior of a real-world object. It intentionally focuses on the basics, showing how even a simple class can cleanly model state and behavior.

In the real world, you'll often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an instance of the class of objects known as bicycles. A class is the blueprint from which individual objects are created.

The following Bicycle class is one possible implementation of a bicycle:

*class Bicycle {*

*int cadence = 0;*

*int speed = 0;*

*int gear = 1;*

*void changeCadence(int newValue) {*

*cadence = newValue;*

*}*

*void changeGear(int newValue) {*

*gear = newValue;*

*}*

*void speedUp(int increment) {*

*speed = speed + increment;*

*}*

*void applyBrakes(int decrement) {*

*speed = speed - decrement;*

*}*

*void printStates() {*

*System.out.println("cadence:" +*

*cadence + " speed:" +*

*speed + " gear:" + gear);*

*}*

*}*

The syntax of the Java programming language will look new to you, but the design of this class is based on the previous discussion of bicycle objects. The fields cadence, speed, and gear represent the object's state, and the methods (changeCadence, changeGear, speedUp etc.) define its interaction with the outside world.

You may have noticed that the Bicycle class does not contain a main method. That's because it's not a complete application; it's just the blueprint for bicycles that might be used in an application. The responsibility of creating and using new Bicycle objects belongs to some other class in your application.

Here's a BicycleDemo class that creates two separate Bicycle objects and invokes their methods:

*class BicycleDemo {*

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

*// Create two different*

*// Bicycle objects*

*Bicycle bike1 = new Bicycle();*

*Bicycle bike2 = new Bicycle();*

*// Invoke methods on*

*// those objects*

*bike1.changeCadence(50);*

*bike1.speedUp(10);*

*bike1.changeGear(2);*

*bike1.printStates();*

*bike2.changeCadence(50);*

*bike2.speedUp(10);*

*bike2.changeGear(2);*

*bike2.changeCadence(40);*

*bike2.speedUp(10);*

*bike2.changeGear(3);*

*bike2.printStates();*

*}*

*}*

The output of this test prints the ending pedal cadence, speed, and gear for the two bicycles:

*cadence:50 speed:10 gear:2*

*cadence:40 speed:20 gear:3*

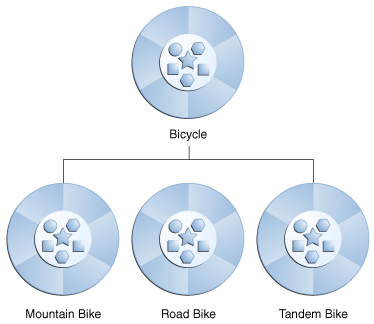
### What Is Inheritance?

Inheritance provides a powerful and natural mechanism for organizing and structuring your software. This section explains how classes inherit state and behavior from their superclasses, and explains how to derive one class from another using the simple syntax provided by the Java programming language.

Different kinds of objects often have a certain amount in common with each other. Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics of bicycles (current speed, current pedal cadence, current gear). Yet each also defines additional features that make them different: tandem bicycles have two seats and two sets of handlebars; road bikes have drop handlebars; some mountain bikes have an additional chain ring, giving them a lower gear ratio.

Object-oriented programming allows classes to inherit commonly used state and behavior from other classes. In this example, Bicycle now becomes the superclass of MountainBike, RoadBike, and TandemBike. In the Java programming language, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of subclasses:

**A hierarchy of bicycle classes**.



The syntax for creating a subclass is simple. At the beginning of your class declaration, use the extends keyword, followed by the name of the class to inherit from:

*class MountainBike extends Bicycle {*

*// new fields and methods defining*

*// a mountain bike would go here*

*}*

This gives MountainBike all the same fields and methods as Bicycle, yet allows its code to focus exclusively on the features that make it unique. This makes code for your subclasses easy to read. However, you must take care to properly document the state and behavior that each superclass defines, since that code will not appear in the source file of each subclass.

### What Is an Interface?

An interface is a contract between a class and the outside world. When a class implements an interface, it promises to provide the behavior published by that interface. This section defines a simple interface and explains the necessary changes for any class that implements it.

Methods form the object's interface with the outside world; the buttons on the front of your television set, for example, are the interface between you and the electrical wiring on the other side of its plastic casing. You press the "power" button to turn the television on and off.

In its most common form, an interface is a group of related methods with empty bodies. A bicycle's behavior, if specified as an interface, might appear as follows:

*interface Bicycle {*

*// wheel revolutions per minute*

*void changeCadence(int newValue);*

*void changeGear(int newValue);*

*void speedUp(int increment);*

*void applyBrakes(int decrement);*

*}*

To implement this interface, the name of your class would change (to a particular brand of bicycle, for example, such as ACMEBicycle), and you'd use the implements keyword in the class declaration:

*class ACMEBicycle implements Bicycle {*

*int cadence = 0;*

*int speed = 0;*

*int gear = 1;*

*// The compiler will now require that methods*

*// changeCadence, changeGear, speedUp, and applyBrakes*

*// all be implemented. Compilation will fail if those*

*// methods are missing from this class.*

*void changeCadence(int newValue) {*

*cadence = newValue;*

*}*

*void changeGear(int newValue) {*

*gear = newValue;*

*}*

*void speedUp(int increment) {*

*speed = speed + increment;*

*}*

*void applyBrakes(int decrement) {*

*speed = speed - decrement;*

*}*

*void printStates() {*

*System.out.println("cadence:" +*

*cadence + " speed:" +*

*speed + " gear:" + gear);*

*}*

*}*

Implementing an interface allows a class to become more formal about the behavior it promises to provide. Interfaces form a contract between the class and the outside world, and this contract is enforced at build time by the compiler. If your class claims to implement an interface, all methods defined by that interface must appear in its source code before the class will successfully compile.

### What Is a Package?

A package is a namespace for organizing classes and interfaces in a logical manner. Placing your code into packages makes large software projects easier to manage. This section explains why this is useful, and introduces you to the Application Programming Interface (API) provided by the Java platform.

Its packages represent the tasks most commonly associated with general-purpose programming. For example, a String object contains state and behavior for character strings; a File object allows a programmer to easily create, delete, inspect, compare, or modify a file on the filesystem; a Socket object allows for the creation and use of network sockets; various GUI objects control buttons and checkboxes and anything else related to graphical user interfaces. There are literally thousands of classes to choose from. This allows you, the programmer, to focus on the design of your particular application, rather than the infrastructure required to make it work.

The Java Platform API Specification contains the complete listing for all packages, interfaces, classes, fields, and methods supplied by the Java SE platform. Load the page in your browser and bookmark it. As a programmer, it will become your single most important piece of reference documentation.

## Define the scope of variables

**Variables**

A variable is a container that holds values that are used in a Java program. Every variable must be declared to use a data type. For example, a variable could be declared to use one of the eight primitive data types: byte, short, int, long, float, double, char or Boolean. And, every variable must be given an initial value before it can be used.

**Examples:**

*int myAge = 21;*

The variable "myAge" is declared to be an int data type and initialized to a value of 21.

## Define the structure of a Java class

Now that you've seen the "Hello World!" application (and perhaps even compiled and run it), you might be wondering how it works. Here again is its code:

*class HelloWorldApp {*

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

*System.out.println("Hello World!"); // Display the string.*

*}*

*}*

## Create executable Java applications with a main method

"Hello World!" for the NetBeans IDE

"Hello World!" for Microsoft Windows

"Hello World!" for Solaris OS and Linux

A Closer Look at the "Hello World!" Application

## Declare and initialize variables.

## Initializing Fields

We often provide an initial value for a field in its declaration:

For example:

*public class BedAndBreakfast {*

*// initialize to 10*

*public static int capacity = 10;*

*// initialize to false*

*private boolean full = false;*

*}*

## Differentiate between object reference variables and primitive variables.

Primitive Data Types

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

*int gear = 1;*

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

* Byte
* Short
* Int
* Long
* Float
* Double
* Boolean
* Char

## Read or write to object fields

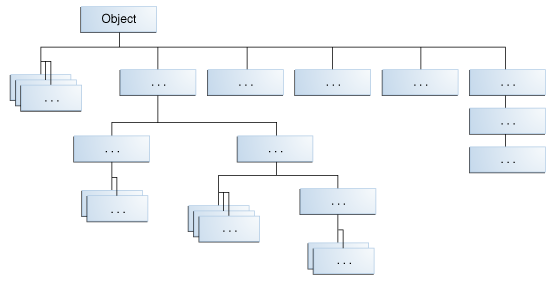
### Inheritance

In the Java language, classes can be derived from other classes, thereby inheriting fields and methods from those classes.

#### The Java Platform Class Hierarchy

The Object class, defined in the java.lang package, defines and implements behavior common to all classes—including the ones that you write. In the Java platform, many classes derive directly from Object, other classes derive from some of those classes, and so on, forming a hierarchy of classes.

**All Classes in the Java Platform are Descendants of Object**

****

At the top of the hierarchy, Object is the most general of all classes. Classes near the bottom of the hierarchy provide more specialized behavior.

#### An Example of Inheritance

Here is the sample code for a possible implementation of a Bicycle class that was presented in the Classes and Objects lesson:

*public class Bicycle {*

*// the Bicycle class has three fields*

*public int cadence;*

*public int gear;*

*public int speed;*

*// the Bicycle class has one constructor*

*public Bicycle(int startCadence, int startSpeed, int startGear) {*

*gear = startGear;*

*cadence = startCadence;*

*speed = startSpeed;*

*}*

*// the Bicycle class has four methods*

*public void setCadence(int newValue) {*

*cadence = newValue;*

*}*

*public void setGear(int newValue) {*

*gear = newValue;*

*}*

*public void applyBrake(int decrement) {*

*speed -= decrement;*

*}*

*public void speedUp(int increment) {*

*speed += increment;*

*}*

*}*

A class declaration for a MountainBike class that is a subclass of Bicycle might look like this:

*public class MountainBike extends Bicycle {*

*// the MountainBike subclass adds one field*

*public int seatHeight;*

*// the MountainBike subclass has one constructor*

*public MountainBike(int startHeight,*

*int startCadence,*

*int startSpeed,*

*int startGear) {*

*super(startCadence, startSpeed, startGear);*

*seatHeight = startHeight;*

*}*

*// the MountainBike subclass adds one method*

*public void setHeight(int newValue) {*

*seatHeight = newValue;*

*}*

*}*

MountainBike inherits all the fields and methods of Bicycle and adds the field seatHeight and a method to set it. Except for the constructor, it is as if you had written a new MountainBike class entirely from scratch, with four fields and five methods. However, you didn't have to do all the work. This would be especially valuable if the methods in the Bicycle class were complex and had taken substantial time to debug.

* + - 1. **What You Can Do in a Subclass**
* A subclass inherits all of the public and protected members of its parent, no matter what package the subclass is in. If the subclass is in the same package as its parent, it also inherits the package-private members of the parent. You can use the inherited members as is, replace them, hide them, or supplement them with new members:
* he inherited fields can be used directly, just like any other fields.
* You can declare a field in the subclass with the same name as the one in the superclass, thus hiding it (not recommended).
* You can declare new fields in the subclass that are not in the superclass.
* The inherited methods can be used directly as they are.
* You can write a new instance method in the subclass that has the same signature as the one in the superclass, thus overriding it.
* You can write a new static method in the subclass that has the same signature as the one in the superclass, thus hiding it.
* You can declare new methods in the subclass that are not in the superclass.
* You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.
* **Declaring Member Variables**

There are several kinds of variables:

* + Member variables in a class—these are called fields.
  + Variables in a method or block of code—these are called local variables.
  + Variables in method declarations—these are called parameters.

The Bicycle class uses the following lines of code to define its fields:

* public int cadence;
* public int gear;
* public int speed;

Field declarations are composed of three components, in order:

* Zero or more modifiers, such as public or private.
* The field's type.
* The field's name.

The fields of Bicycle are named cadence, gear, and speed and are all of data type integer (int). The public keyword identifies these fields as public members, accessible by any object that can access the class.

* **Creating Objects**

A class provides the blueprint for objects; you create an object from a class. Each of the following statements taken from the CreateObjectDemo program creates an object and assigns it to a variable:

Point originOne = new Point(23, 94);

Rectangle rectOne = new Rectangle(originOne, 100, 200);

Rectangle rectTwo = new Rectangle(50, 100);

The first line creates an object of the Point class, and the second and third lines each create an object of the Rectangle class.

Each of these statements has three parts (discussed in detail below):

* Declaration: The code set in bold are all variable declarations that associate a variable name with an object type.
* Instantiation: The new keyword is a Java operator that creates the object.
* Initialization: The new operator is followed by a call to a constructor, which initializes the new object.

**Declaring a Variable to Refer to an Object**

Previously, you learned that to declare a variable, you write:

type name;

This notifies the compiler that you will use name to refer to data whose type is type. With a primitive variable, this declaration also reserves the proper amount of memory for the variable.

You can also declare a reference variable on its own line. For example:

Point originOne;

If you declare originOne like this, its value will be undetermined until an object is actually created and assigned to it. Simply declaring a reference variable does not create an object. For that, you need to use the new operator, as described in the next section. You must assign an object to originOne before you use it in your code. Otherwise, you will get a compiler error.

**Instantiating a Class**

The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the object constructor.

The new operator requires a single, postfix argument: a call to a constructor. The name of the constructor provides the name of the class to instantiate.

The new operator returns a reference to the object it created. This reference is usually assigned to a variable of the appropriate type, like:

Point originOne = new Point(23, 94);

The reference returned by the new operator does not have to be assigned to a variable. It can also be used directly in an expression. For example:

*int height = new Rectangle().height;*

**Initializing an Object**

Here's the code for the Point class:

*public class Point {*

*public int x = 0;*

*public int y = 0;*

*//constructor*

*public Point(int a, int b) {*

*x = a;*

*y = b;*

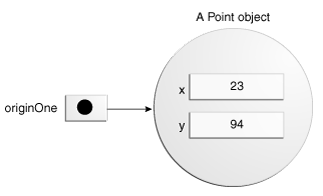
*}*

*}*

This class contains a single constructor. You can recognize a constructor because its declaration uses the same name as the class and it has no return type. The constructor in the Point class takes two integer arguments, as declared by the code (int a, int b). The following statement provides 23 and 94 as values for those arguments:

*Point originOne = new Point(23, 94);*

The result of executing this statement can be illustrated in the next figure:



Here's the code for the Rectangle class, which contains four constructors:

*public class Rectangle {*

*public int width = 0;*

*public int height = 0;*

*public Point origin;*

*// four constructors*

*public Rectangle() {*

*origin = new Point(0, 0);*

*}*

*public Rectangle(Point p) {*

*origin = p;*

*}*

*public Rectangle(int w, int h) {*

*origin = new Point(0, 0);*

*width = w;*

*height = h;*

*}*

*public Rectangle(Point p, int w, int h) {*

*origin = p;*

*width = w;*

*height = h;*

*}*

*// a method for moving the rectangle*

*public void move(int x, int y) {*

*origin.x = x;*

*origin.y = y;*

*}*

*// a method for computing the area of the rectangle*

*public int getArea() {*

*return width \* height;*

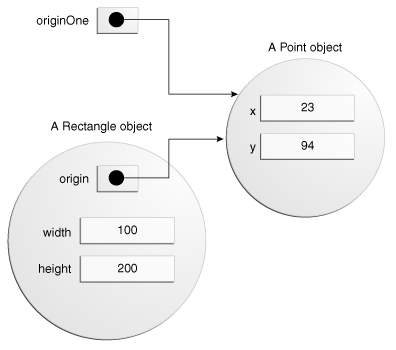
*}*

*}*

Each constructor lets you provide initial values for the rectangle's origin, width, and height, using both primitive and reference types. If a class has multiple constructors, they must have different signatures. The Java compiler differentiates the constructors based on the number and the type of the arguments. When the Java compiler encounters the following code, it knows to call the constructor in the Rectangle class that requires a Point argument followed by two integer arguments:

*Rectangle rectOne = new Rectangle(originOne, 100, 200);*

This calls one of Rectangle's constructors that initializes origin to originOne. Also, the constructor sets width to 100 and height to 200. Now there are two references to the same Point object—an object can have multiple references to it, as shown in the next figure:



The following line of code calls the Rectangle constructor that requires two integer arguments, which provide the initial values for width and height. If you inspect the code within the constructor, you will see that it creates a new Point object whose x and y values are initialized to 0:

*Rectangle rectTwo = new Rectangle(50, 100);*

The Rectangle constructor used in the following statement doesn't take any arguments, so it's called a no-argument constructor:

*Rectangle rect = new Rectangle();*

All classes have at least one constructor. If a class does not explicitly declare any, the Java compiler automatically provides a no-argument constructor, called the default constructor. This default constructor calls the class parent's no-argument constructor, or the Object constructor if the class has no other parent. If the parent has no constructor (Object does have one), the compiler will reject the program.

* Using Objects

## Explain an object's lifecycle

* Objects
* Creating Objects
* Using Objects

## Call methods on objects.

* Using Objects

## Manipulate data using the String Builder class and its methods.

* **The String Builder Class**

**Difference Between String, StringBuilder And StringBuffer Classes With Example: Java**

There are minor differences between the above mentioned classes.

* **String**

String is immutable (once created cannot be changed) object. The object created as a String is stored in the Constant String Pool.

Every immutable object in Java is thread safe that implies String is also thread safe. String cannot be used by two threads simultaneously.

String once assigned cannot be changed.

*String demo = “hello " ;*

*// The above object is stored in constant string pool and its value cannot be modified.*

*demo="Bye" ; //new "Bye" string is created in constant pool and referenced by the demo variable*

*// "hello" string still exists in string constant pool and its value is not overrided but we lost reference to the "hello"string*

* **String Buffer**

String Buffer is mutable means one can change the value of the object. The object created through String Buffer is stored in the heap. String Buffer has the same methods as the String Builder, but each method in String Buffer is synchronized that is String Buffer is thread safe.

Due to this it does not allow two threads to simultaneously access the same method. Each method can be accessed by one thread at a time.

But being thread safe has disadvantages too as the performance of the String Buffer hits due to thread safe property. Thus String Builder is faster than the String Buffer when calling the same methods of each class.

String Buffer value can be changed, it means it can be assigned to the new value. Nowadays it’s a most common interview question, the differences between the above classes.

String Buffer can be converted to the string by using

*toString() method.*

*StringBuffer demo1 = new StringBuffer("Hello") ;*

*// The above object stored in heap and its value can be changed.*

*demo1=new StringBuffer("Bye");*

*// Above statement is right as it modifies the value which is allowed in the StringBuffer*

* **String Builder**

String Builder is same as the String Buffer that is it stores the object in heap and it can also be modified. The main difference between the String Buffer and String Builder is that String Builder is also not thread safe.

String Builder is fast as it is not thread safe.

*StringBuilder demo2= new StringBuilder("Hello");*

*// The above object too is stored in the heap and its value can be modified*

*demo2=new StringBuilder("Bye");*

*// Above statement is right as it modifies the value which is allowed in the StringBuilder*

|  |  |  |  |
| --- | --- | --- | --- |
|  | String | StringBuffer | StringBuilder |
| Storage Area  Modifiable  Thread Safe  Performance | Constant String Pool  No (immutable)  Yes  Fast | Heap  Yes( mutable )  Yes  Very slow | Heap  Yes( mutable )  No  Fast |

* Summary of Characters and Strings

Most of the time, if you are using a single character value, you will use the primitive char type. There are times, however, when you need to use a char as an object—for example, as a method argument where an object is expected. The Java programming language provides a wrapper class that "wraps" the char in a Character object for this purpose. An object of type Character contains a single field whose type is char. This Character class also offers a number of useful class (i.e., static) methods for manipulating characters.

Strings are a sequence of characters and are widely used in Java programming. In the Java programming language, strings are objects. The String class has over 60 methods and 13 constructors.

Most commonly, you create a string with a statement like

*String s = "Hello world!";*

rather than using one of the String constructors.

The String class has many methods to find and retrieve substrings; these can then be easily reassembled into new strings using the + concatenation operator.

The String class also includes a number of utility methods, among them *split(), toLowerCase(), toUpperCase(), and valueOf().* The latter method is indispensable in converting user input strings to numbers. The Number subclasses also have methods for converting strings to numbers and vice versa.

In addition to the String class, there is also a StringBuilder class. Working with StringBuilder objects can sometimes be more efficient than working with strings. The StringBuilder class offers a few methods that can be useful for strings, among them *reverse().* In general, however, the String class has a wider variety of methods.

A string can be converted to a string builder using a StringBuilder constructor. A string builder can be converted to a string with the *toString()* method.

## Create and manipulate strings.

* Strings

Java String provides a lot of concepts that can be performed on a string such as compare, concat, equals, split, length, replace, compareTo, intern, substring etc.

In java, string is basically an object that represents sequence of char values.

An array of characters works same as java string. For example:

*char[] ch={'j','a','v','a','t','p','o','i','n','t'};*

*String s=new String(ch);*

**What is String in java?**

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

**How to create String object?**

There are two ways to create String object:

* By string literal.
* By new keyword.

**String Literal**

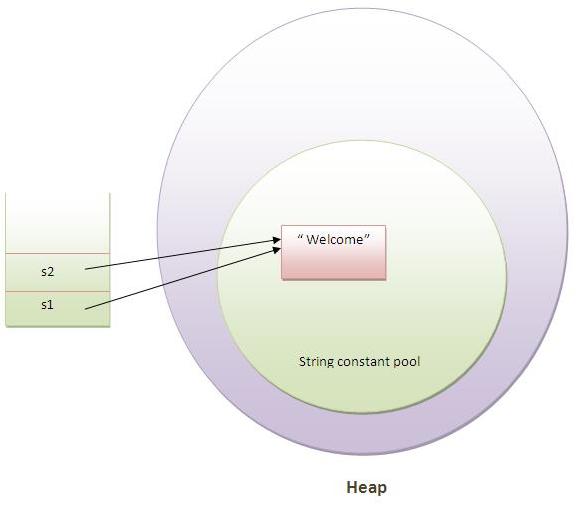
Java String literal is created by using double quotes. For Example:

*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 string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

*String s1="Welcome";*

*String s2="Welcome";//will not create 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, so it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create new object but will return the reference to the same instance.

* Converting Between Numbers and Strings.
* Comparing Strings and Portions of Strings.
* Manipulating Characters in a String.

## Use Java operators.

* Operators

The following quick reference summarizes the operators supported by the Java programming language.

**Simple Assignment Operator**

*= Simple assignment operator*

**Arithmetic Operators**

*+ Additive operator (also used*

*for String concatenation)*

*- Subtraction operator*

*\* Multiplication operator*

*/ Division operator*

*% Remainder operator*

**Unary Operators**

*+ Unary plus operator; indicates*

*positive value (numbers are*

*positive without this, however)*

*- Unary minus operator; negates*

*an expression*

*++ Increment operator; increments*

*a value by 1*

*-- Decrement operator; decrements*

*a value by 1*

*! Logical complement operator;*

*inverts the value of a boolean*

**Equality and Relational Operators**

*== Equal to*

*!= Not equal to*

*> Greater than*

*>= Greater than or equal to*

*< Less than*

*<= Less than or equal to*

**Conditional Operators**

*&& Conditional-AND*

*|| Conditional-OR*

*?: Ternary (shorthand for*

*if-then-else statement)*

* Assignment, Arithmetic, and Unary Operators
* Equality, Relational, and Conditional Operators
* Bitwise and Bit Shift

## Use parentheses to override operator precedence

* Expressions, Statements, and Blocks

## Creating and Using Arrays

* Declare, instantiate, initialize and use a one-dimensional array

## Declare and use an Array List

* The List Interface
* List Implementations

## Using Loop Constructs

* The while and do-while Statements.
* Create and use for loops including the enhanced for loop.
* The for Statement.
* The while and do-while Statements.

References

1. <https://docs.oracle.com/javase/tutorial/java/javaOO/objectcreation.html>
2. <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/opsummary.html>
3. <http://docs.oracle.com/javase/tutorial/java/concepts/>
4. <http://javahyngry.blogspot.com>