

JAVA FOP-1 : Topics

- **Java Features**
- **Java Language Basics**
- **Java Naming Rules and Conventions**
- **Writing the code right way**
- **Java Fundamentals**
 - **Variables:**
 - **Java Primitive Types &**
 - **Reference Types**
 - **String API**
 - **Date and Time API (including JDK8 API)**
 - **Arrays**

Features of Java

- **Object-oriented**
 - Abstraction, Encapsulation , Inheritance , Polymorphism
- **Simpler language**
 - Compared to earlier OO languages like C++, it is simple
 - Designed considering the pitfalls of earlier languages
- **Robust**
 - Automatic memory management and garbage collection is the biggest contributor here. Strong type checking also helps.
- **Architecture Neutral / Portable**
 - Java compiler generates an architecture-neutral object file format which makes the compiled code to be executable on many processors, with the presence of Java runtime system.
- **Secure** : Built -in security features like absence of pointers and confinement of the java program within its runtime environment
- **Multithreading** : Multithreading is the capability for a program to perform several tasks simultaneously within a program
- **Distributed**: Java is designed for the distributed environment of the internet.

Java Language Basics

- Java API is a collection of packages.
- A package is a collection of related classes and interfaces.
- A Java application is a collection of one or more packages.
- Java File extension is .java
- Tester classes contain main() method:
 - main() method signature:
public static void main(String[] args)

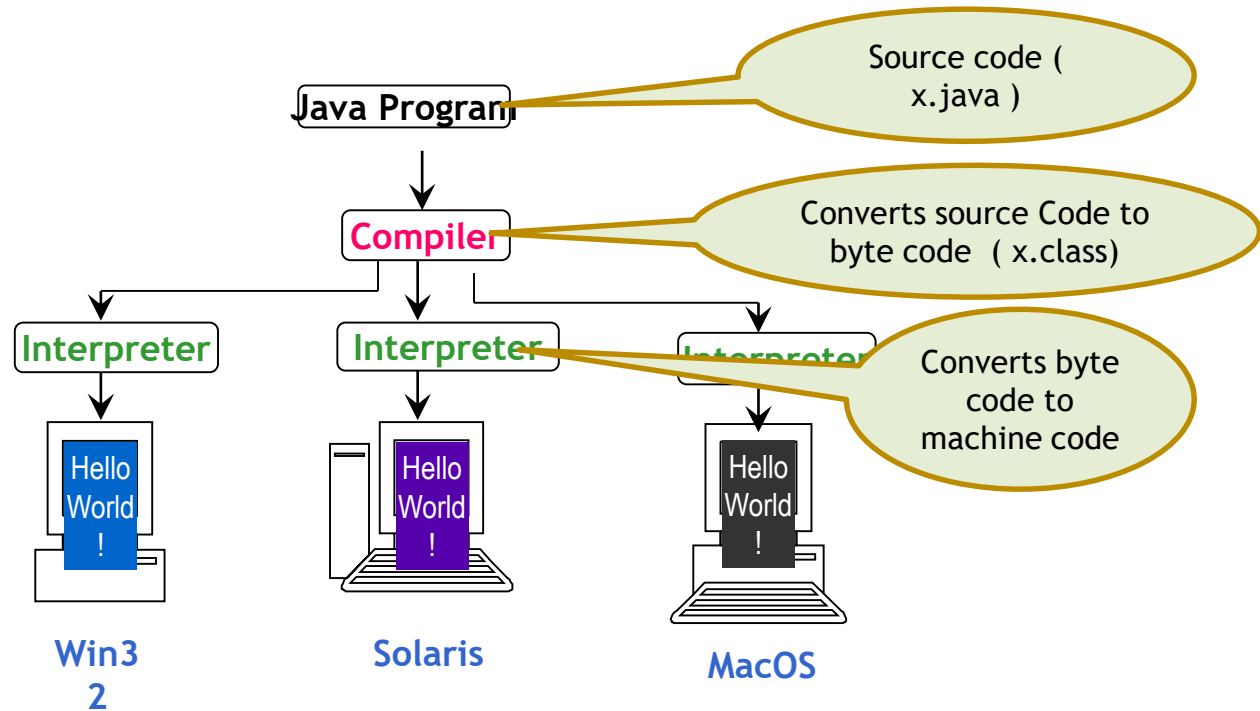
Java Language is :

- **Case-sensitivity**
- **Code Inside the Class Definition**

Running on Different Platforms

Java is a platform independent language

“Write Once Run Anywhere”, popularly known as **WORA**



The Java Platform

- Oracle provides two principal software products in the Java Platform, Standard Edition (Java SE) family:

1. Java SE Runtime Environment (JRE)

- The JRE provides the libraries(packages), Java virtual machine, and other components necessary for you to *run* applications written in the Java programming language.

2. Java SE Development Kit (JDK)

- The JDK includes the JRE *plus* command-line development tools such as *compilers and debuggers* that are necessary or useful for *developing* **applets** and **applications**.

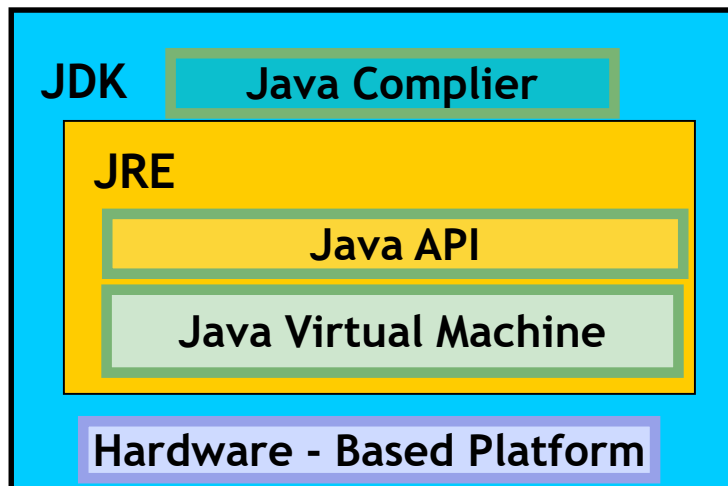
Java Platform

Java Development Kit (JDK)

- JDK consists of Java API, Java Compiler, and JVM

The Java Application Programming Interface (API)

The API is a large collection of ready-made software components that provide many useful capabilities. It is grouped into libraries of related classes and interfaces; these libraries are known as *packages*.

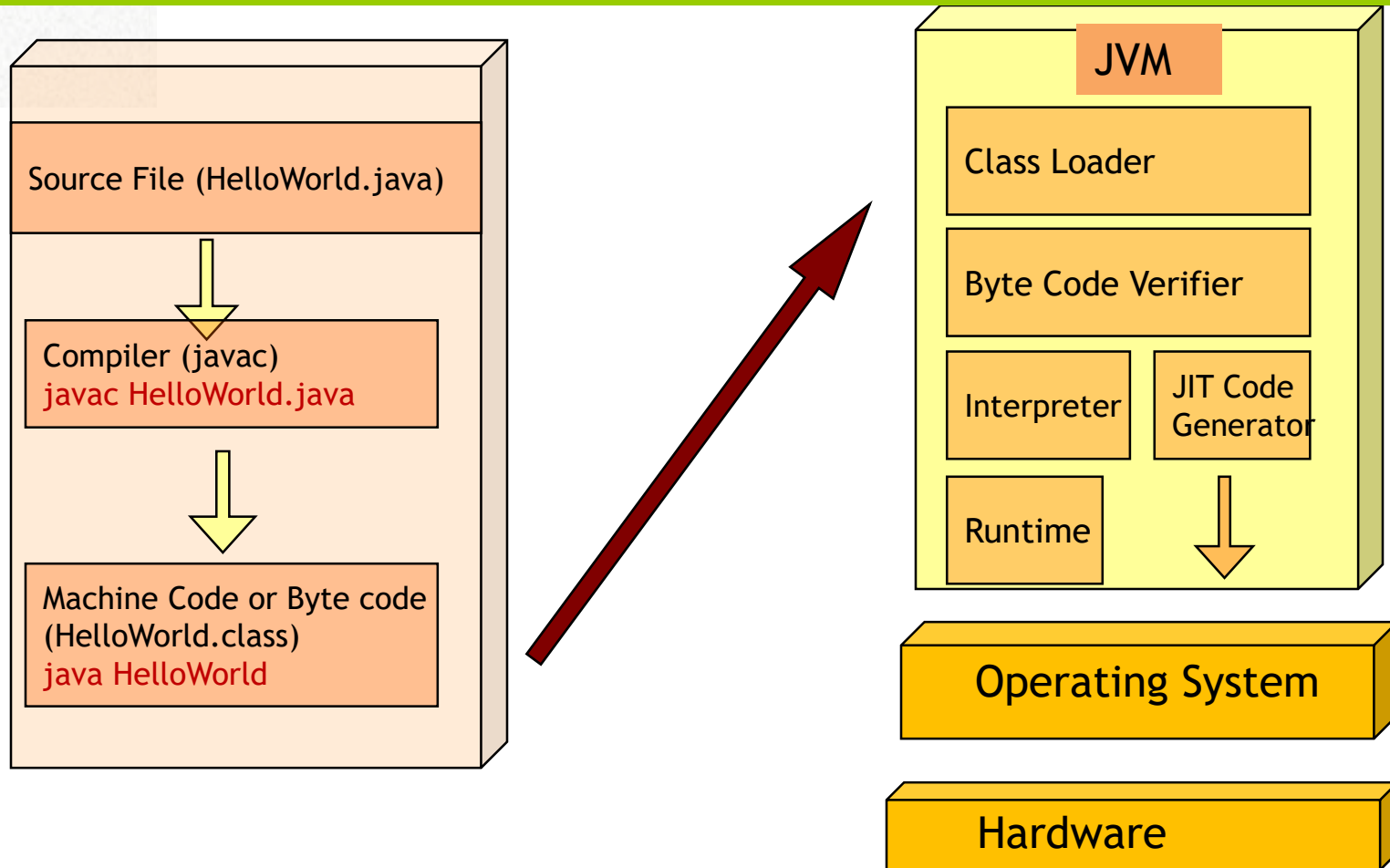


The JVM is by definition a virtual machine, that is a software machine that simulates what a real machine does.

Like real machines it has an instruction set (the bytecodes), a virtual computer architecture and an execution model.

It is capable of running code written with this virtual instruction set, pretty much like a real machine can run machine code.

Java Code Compilation and Execution



In computing, **just-in-time (JIT) compilation**, also known as **dynamic translation**, is compilation done during execution of a program - at run time - rather than prior to execution.

Most often this consists of translation to machine code, which is then executed directly.

Class loader hierarchy

Whenever a new JVM is started the bootstrap class loader is responsible to load key Java classes (from java.lang package) and other runtime classes to the memory first. The bootstrap class loader is a parent of all other class loaders

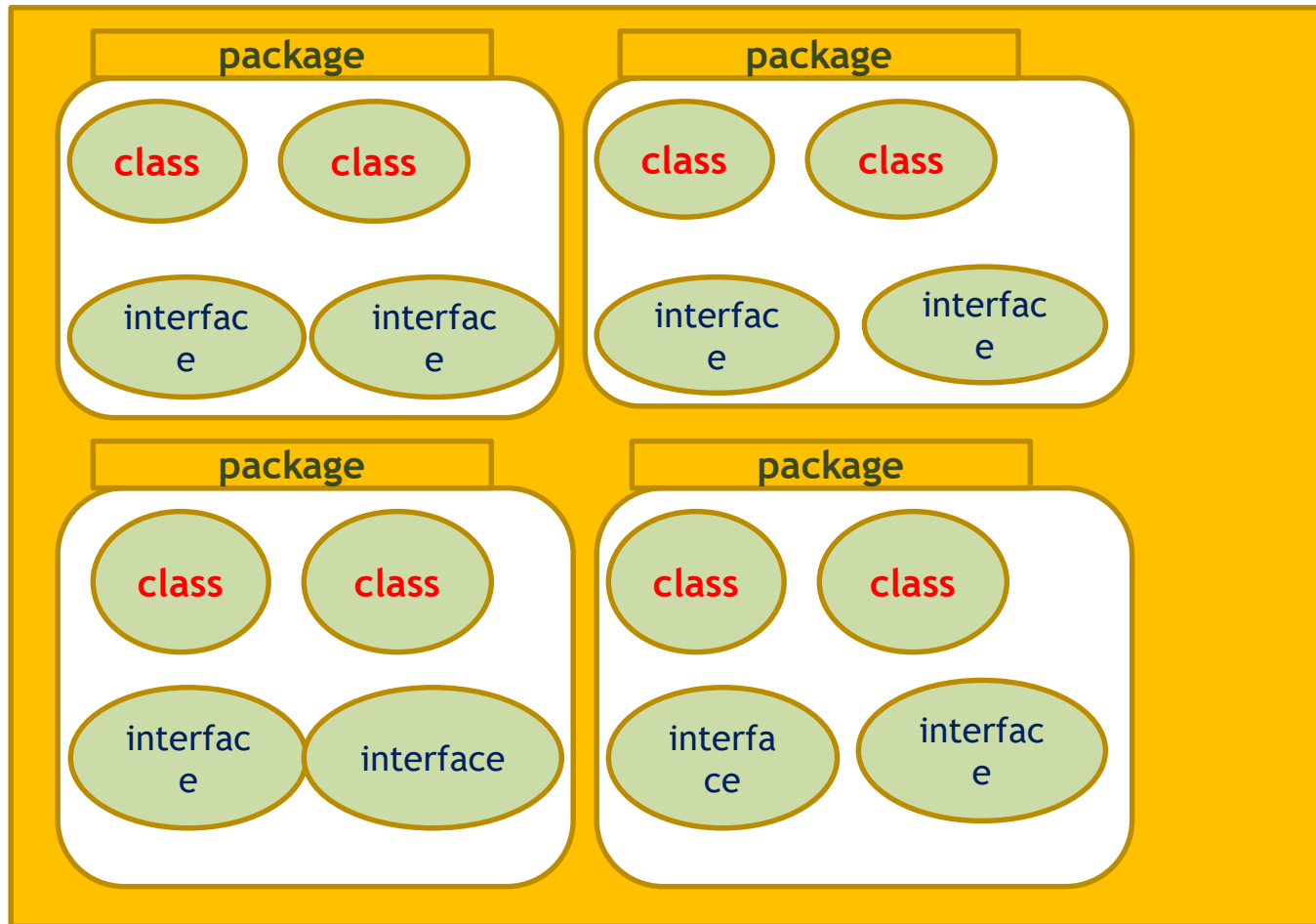


Next comes the extension classloader. It has the bootstrap class loader as parent and is responsible for loading classes from all .jar files kept in the `../jre/lib/ext` directory path—these are available regardless of the JVM's classpath.

The third and most important class loader from a developer's perspective is the system classpath class loader, which is an immediate child of the extension class loader.

It loads classes from directories and jar files specified by the CLASSPATH environment variable, java.class.path system property or **-classpath** command line option.

JAVA API



Structure of a java program

```
[package packagename; ]
```

```
import packagename.classname;
```

```
.....
```

```
public class ClassName [extends SuperClassName [ implements InterfaceName,... ] ]
```

```
{
```

instance variables (instance fields)
class variables (static fields)

instance methods
class methods(static methods)

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

optional

```
}
```

Java Sample Program

HelloWorld.java

//Hello World example.

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

Why public before class name?

What is public static void before main() method?

Keywords

The keywords/ reserved words defined in the Java language has **50 keywords and 3 literals (*true, false & null*)**.

Identifiers

- Identifiers are the **names** of variables, methods, classes, packages and interfaces.
- An identifier is a sequence of one or more characters.
- Identifier is unlimited-length sequence of Unicode letters and digits, beginning with *a letter, dollar sign "\$", or underscore character "_"*.
- Keywords cannot be declared as identifiers
- Identifiers in Java are case-sensitive

Java Naming Conventions

Variable Names and Method Names

Ex. String title; int bookCode; String myFamilyDetails;

- **Types of variables: field Variables & local variables**
- **About Local Variables**
 - *To be initialized before usage*

Class/Interface Names

Ex. Book , AccountMaster, EmployeeContract

Built-in classes:

System, PrintStream , WindowListener , MouseMotionListener etc.

Package Names

Ex. java.io.FileInputStream;
 java.util.Date;

Writing the code right way

The best practices are primarily targeted towards improvement in the readability and maintainability of code with keen attention to performance enhancements.

By employing such practices the application development team can demonstrate their proficiency, profound knowledge and professionalism.

Environment variables

Create system variable, JAVA_HOME which will be the Java Installation directory

System Properties -> Advanced -> Environment Variables -> Click on New button of System variables

Variable Name : `JAVA_HOME`

Variable Value : `C:\Program Files (x86)\Java\jdk1.8.0_05` (installation path of java)

Set the PATH variable : Path variable is used by OS to locate executable files.

In System Variables, select PATH and click on Edit button. Append the following :

`%JAVA_HOME%\bin`

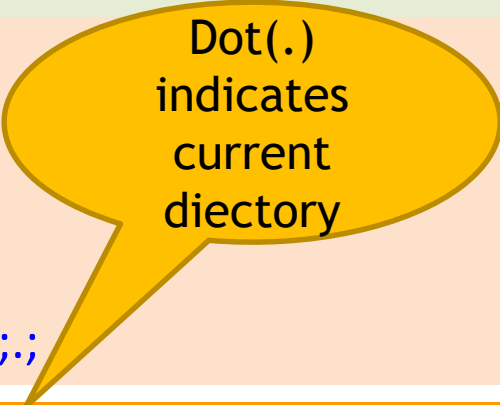
This approach helps in managing multiple versions of Java - Changing JAVA_HOME will reflect on CLASSPATH and PATH as well

Set the CLASSPATH

Click New button of User Variable and enter:

Variable name : CLASSPATH

Variable Value : `%JAVA_HOME%\lib\tools.jar;.`



Dot(.)
indicates
current
diectory

Variables

Two types of variables

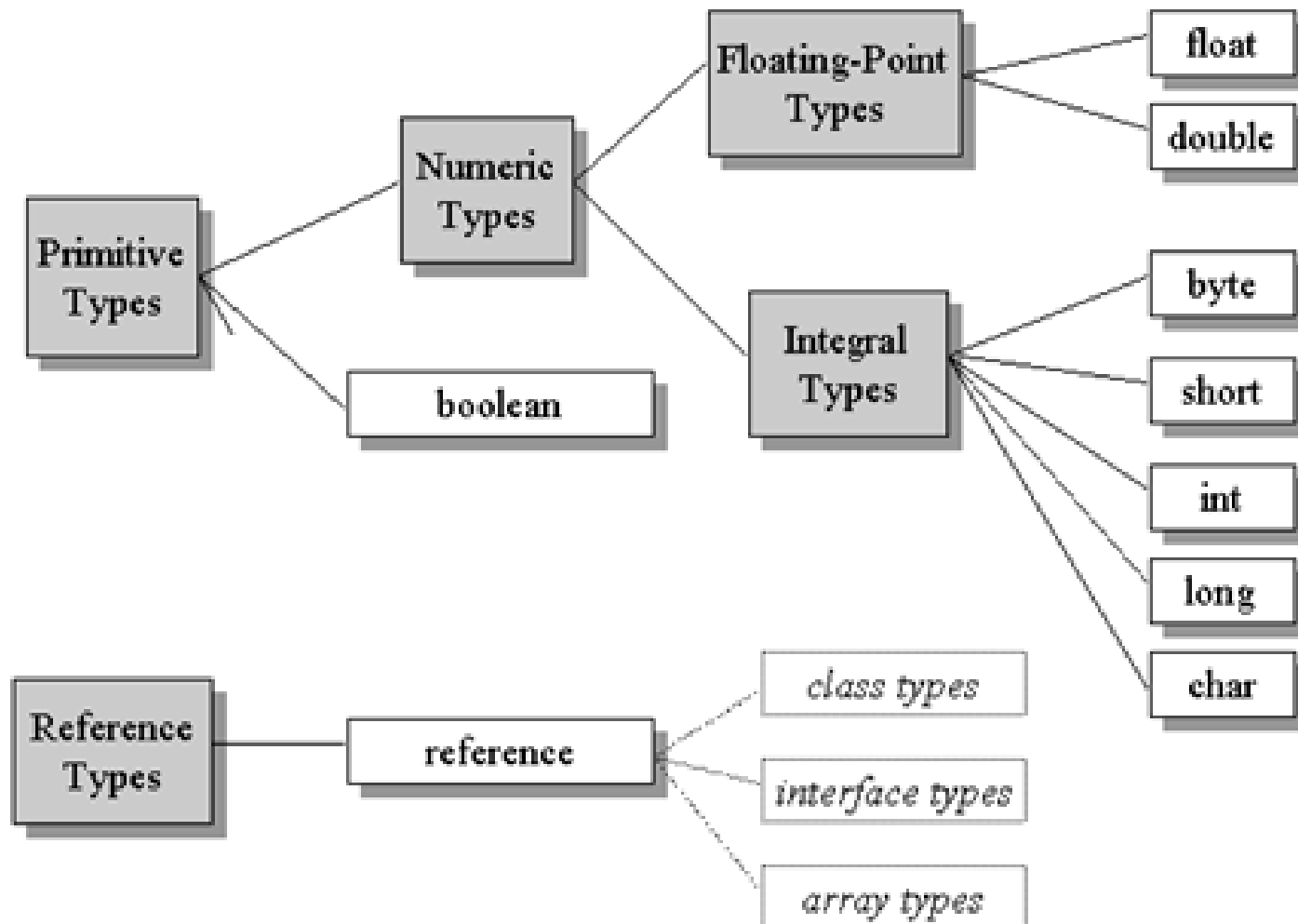
- Primitive type
- Reference type

A variable is a place holder that can hold single value at a time.

Declaration of a variable:

datatype variableName [=value];

Data Types in Java



Primitive Data Types in Java

Integer data types

- byte (1 byte)
- short (2 bytes) -32768 to 32767
- int (4 bytes)
- long (8 bytes)

Floating Type

- float (4 bytes)
 - Ex. float price = 25.67f;
- double (8 bytes)
 - Ex. double salary = 2345.6556;

Textual

- char (2 bytes) 65536
 - Ex. char operator = '+' ;

Logical

- boolean (1 bit (true/false))
 - Ex. boolean flag = true;

Notes:

- @ All numeric data types are signed
- @ The size of data type remains the same on all platforms (standardized)
- @ char data type in Java is 2 bytes because it uses UNICODE character set.
- @ *Unicode* is a industry standard designed to uniquely encode characters used in different human languages.
- @ The other character sets are ASCII and EBCDIC character sets

Declaration of a variable:

datatype variableName(s);
Ex. int radius;
double area, circumference;

Note: datatype can be primitive type or reference type

Variables in Java

Java programming language defines the following types of variables

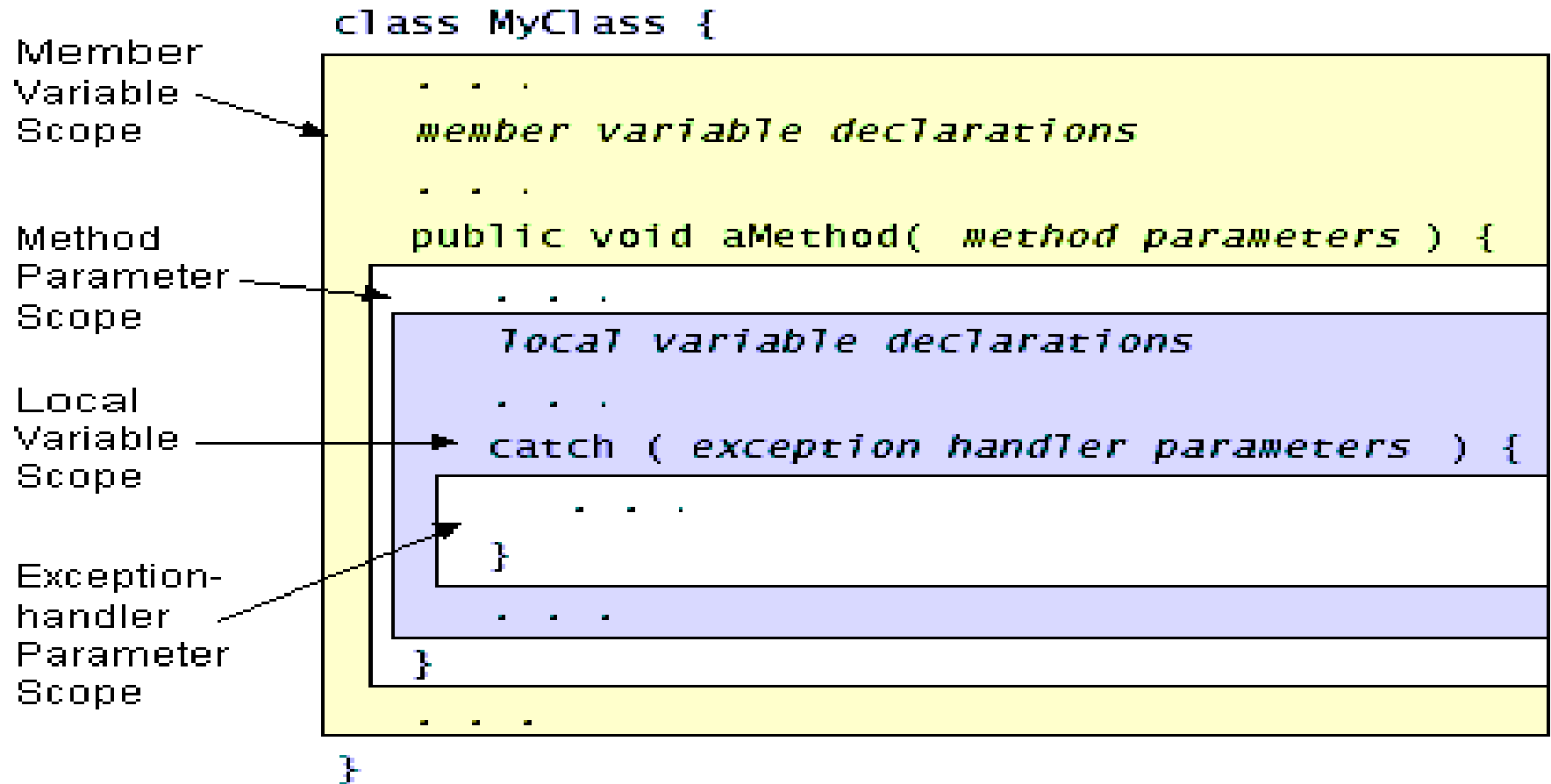
- **Data members/fields of a class:**

- instance Variables
- static/class Variables

- **Local variables : declared within a method**

- **Method parameters/arguments : receive data and they are local to the method**

Variables in Java



JAVA FOP : Topics

- Type casting
- Command-line Arguments

Type Casting

- Java is a Strongly typed language
- Unlike C, at runtime type checking is strictly enforced

In Java, type casting is classified into two types,

- **Widening Casting(Implicit)**

byte → short → int → long → float → double



A red arrow points from the 'byte' type to the 'double' type, indicating a widening conversion.

widening

- **Narrowing Casting(Explicit)**

double → float → long → int → short → byte



A red arrow points from the 'double' type to the 'byte' type, indicating a narrowing conversion.

Narrowing

Invalid Conversions

- Any primitive type to any reference type
- The null value to any primitive type
- Any primitive to **boolean**
- A **boolean** to any primitive

Command-line Arguments

A Java application can accept any number of arguments from the command line.

This allows the user to specify configuration information when the application is launched.

When an application is launched, the runtime system passes the command-line arguments to the application's main method via an array of Strings.

```
public class SumOfDigits {  
    public static void main(String[] args) {  
        try{  
            double r1=Double.parseDouble(args[0]);  
            double r2=Double.parseDouble(args[1]);  
            System.out.println((r1+r2));  
        }  
        catch(NumberFormatException nfe){  
            System.out.println("Invalid data");  
        }  
    }  
}
```


JAVA FOP : Topics

- **Conditional Controls**
 - if statement
 - if else statement
 - Nested if statement
 - To test for more than one condition for a true expression
 - if else if else ladder
 - Usage:
 - Multiple-choice: To select one out of many
 - To check for ranges
 - switch statement

JAVA FOP : Topics

- Iterative Controls
 - Types
 - while loop
 - do while loop
 - for loop
 - Nested Loop
- Break and continue statements in a loop
- About Increment and decrement operators

JAVA FOP : Topics

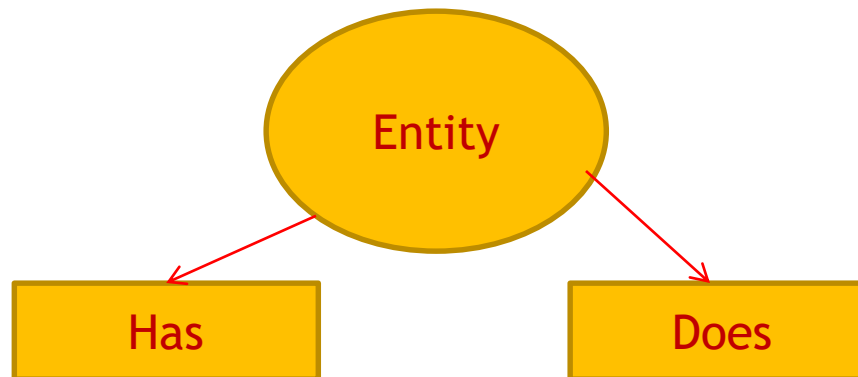
- Introduction to classes and objects
- Explain object scope
- Java reference types
- Object scope
 - Explanation using memory structures

Introduction to classes and objects

A class is a set of specifications that describes the **characteristics (state)** and **behavior(functionality)** of an **entity**.

Entities are real-world objects which are distinguishable from other objects and can be transformed into programming-world classes.

Ex. Account, Employee, Supplier, Student etc



Introduction to classes and objects

Creating an instance/object:

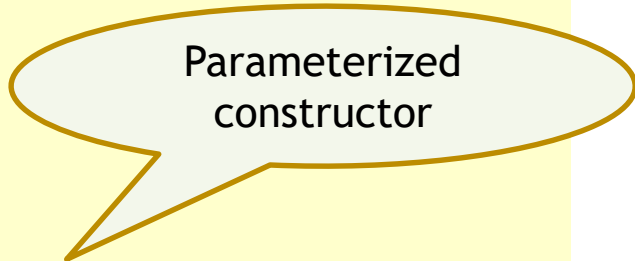
ClassName objectName = new **ClassName()**;

OR

ClassName objectName = new **ClassName(<parameters>)**;



Default constructor



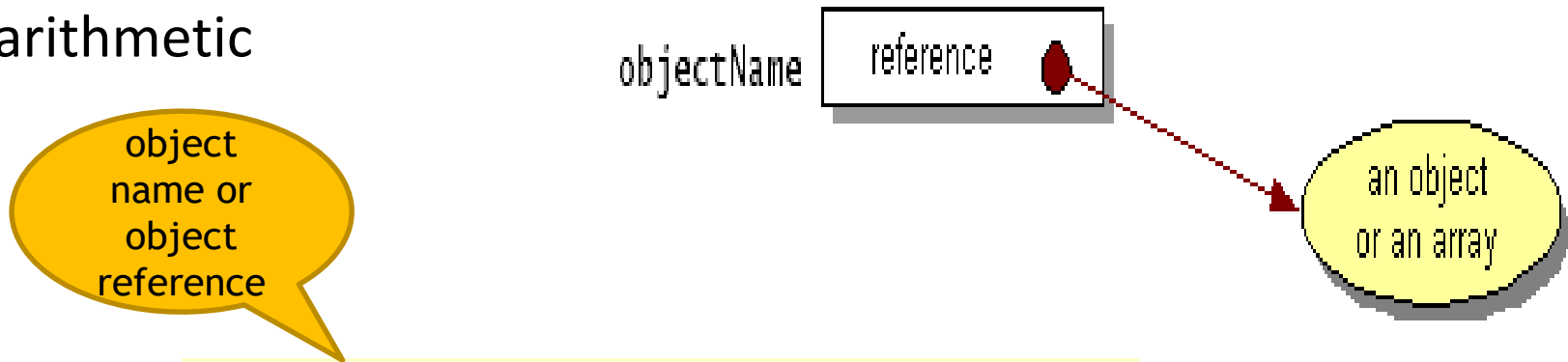
Parameterized
constructor

Note: object name also called as object reference is a reference variable

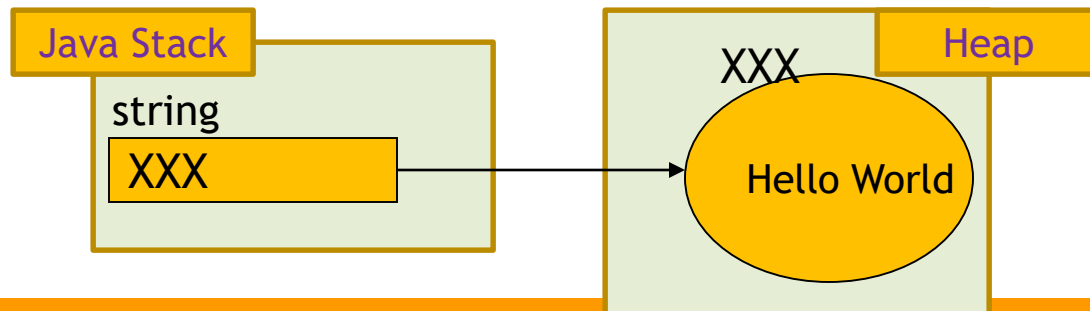
Note: Every instance/object of a class has separate memory allocated in the **Heap**.

Reference Types in Java

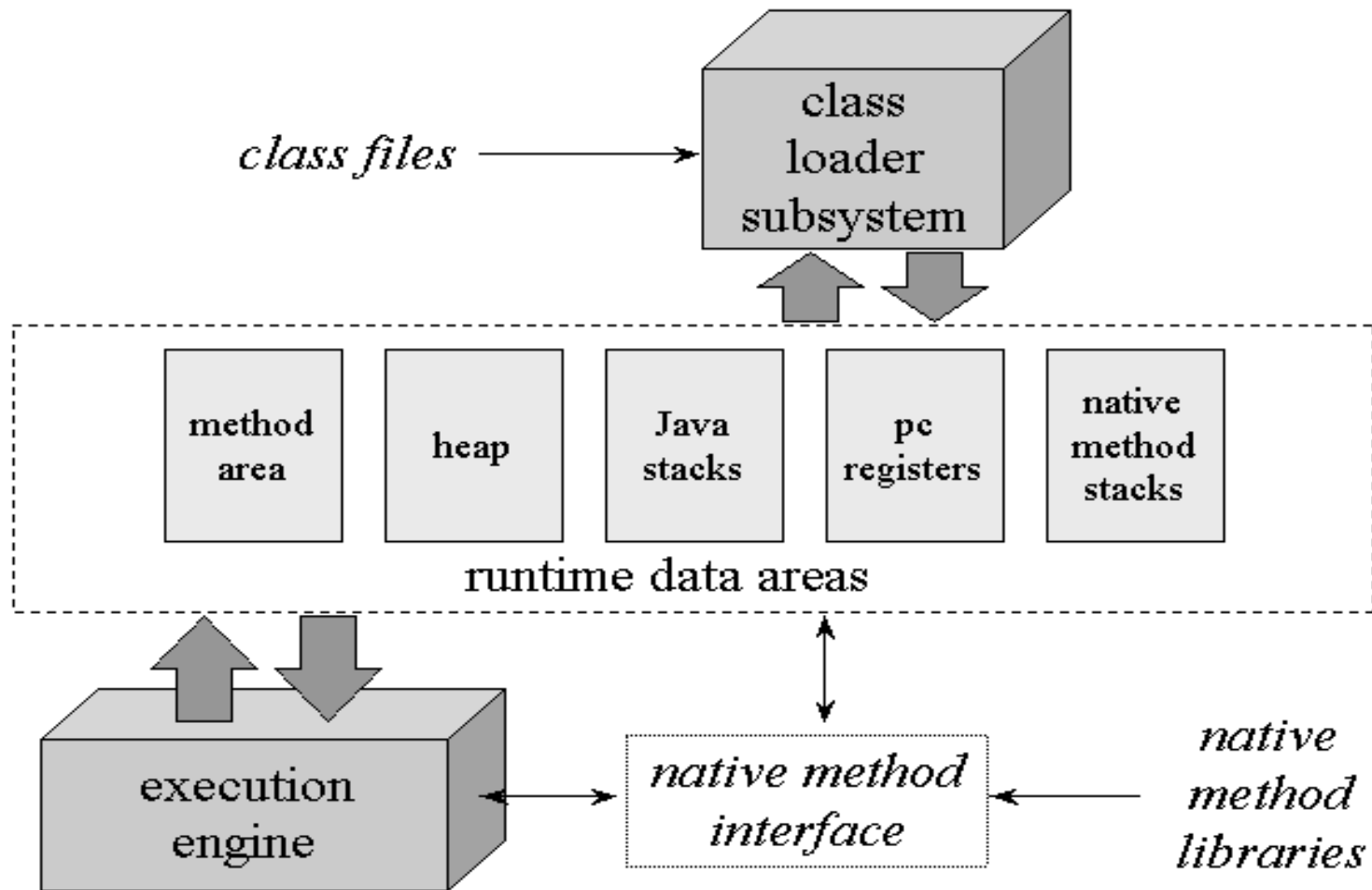
- objects and arrays are accessed using *reference variables* in Java
- A reference variable is similar to a pointer (stores memory address of an object) but Java does not allow pointer manipulation or pointer arithmetic



```
String string = new String("Hello World");
```



JVM Memory Structures



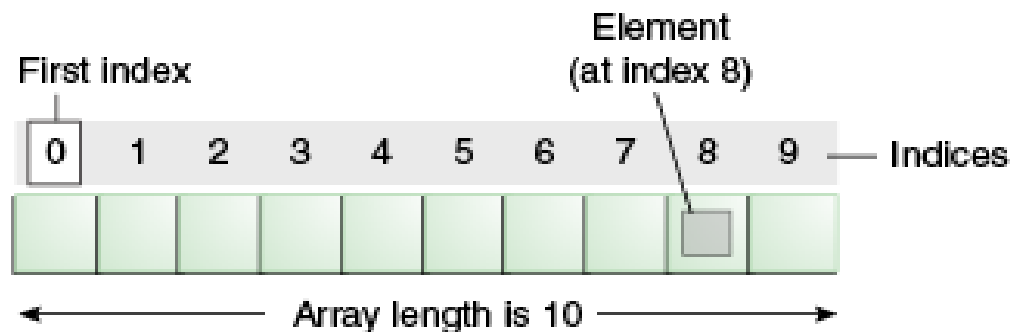
JAVA FOP : Topics

- Arrays
 - Single-dimensional
 - find minimum, maximum, sum and average marks
- Array manipulation:
 - Insert & delete
- Passing arrays to methods
- Methods returning Arrays
- Implementation of stack and queue using arrays
- Arrays class

Arrays

An *array* is a container object that holds a fixed number of values of a single type.

The length of an array is established when the array is created. After creation, its length is fixed.



Each item in an array is called an *element*, and each element is accessed by its numerical *index*.

As shown above, indexing begins with 0.

Creating, Initializing, and Accessing an Array

I. Declaring array name:

`datatype [] arrayName ;`

or

`datatype arrayName[]`

Ex. `int [] list; or int list[];`

Reference
variable

II. Allocating memory for holding data values

`arrayName = new datatype[NE];`

Ex. `list = new int[10];`

NE indicates Number
of Elements(can be
constant or variable)

III. Combine I and II

`datatype arrayName [] = new datatype[NE];`

Ex. `int list[] = new int[10];`

Ex. `String names[] = new String[10];`

Note : data type can be primitive type or reference type.

Traversing an array

It is convenient to use for each loop

The For-Each Loop : *Can iterate through collection of objects*

```
for(Type iterator-variable : iterableObject) {  
    statement-block  
}
```

Methods receiving & returning **reference** types.

Array Manipulation

- Java SE provides several methods for performing array manipulations (common tasks, such as copying, sorting and searching arrays) in the [java.util.Arrays](#) class.
- *Commonly used methods of Arrays class:*
 - static <T> [List](#)<T> [asList](#)(T... a) :Returns a fixed-size list backed by the specified array.
 - static int [binarySearch](#)(int[] a, int key) Searches the specified array of ints for the specified value using the binary search algorithm.
 - static char[] [copyOfRange](#)(char[] original, int from, int to) Copies the specified range of the specified array into a new array.
 - static int[] [copyOfRange](#)(int[] original, int from, int to) Copies the specified range of the specified array into a new array.
 - static boolean [equals](#)(float[] a, float[] a2) Returns true if the two specified arrays of floats are *equal* to one another.
 - static void [fill](#)(double[] a, double val) Assigns the specified double value to each element of the specified array of doubles.
 - static void [sort](#)(double[] a) Sorts the specified array into ascending numerical order.
 - static void [sort](#)([Object](#)[] a) Sorts the specified array of objects into ascending order, according to the [natural ordering](#) of its elements.

Note: The above methods are overloaded to handle all types of arrays including Object type

2D Arrays

Declarations:

```
datatype [ ][ ] arrayName ;
```

```
arrayName = new datatype[NR][NC];
```

OR

```
datatype arrayName [ ] [ ] = new datatype[NR][NC];
```

Note : NR- No. of Rows, NC-No. of columns(can be constant or variable). data type can be primitive type or reference type.

Ex. `int m[][] = new int[4][3];`

Ex. `int [][] list = new int[4][];`

`list[0] = new int[3]; // First row has 3 columns`

`list[1] = new int[4]; // Second row has 4 columns`

Note: rows are allowed to vary in length hence called as jagged arrays.

2D Array Demo

```
class ArrayDemo {  
    public static void main(String[] args)  
    {  
        String[][] names = { {"Mr. ", "Mrs. ", "Ms. "},  
                               {"Smith", "Jones"}  
        };  
  
        System.out.println(names[0][0] + names[1][1]);  
        System.out.println(names[0][2] + names[1][0]);  
    }  
}
```

Mr.Jones

Ms.Smith

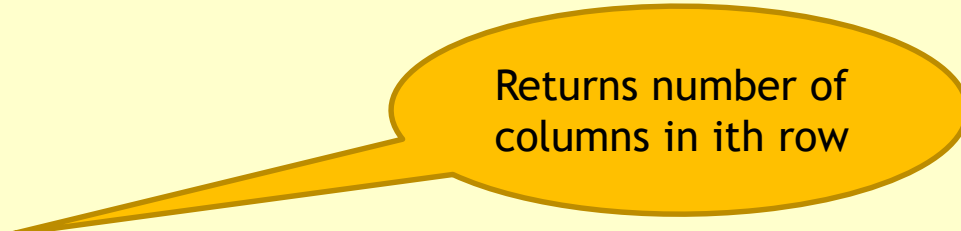
2D Arrays in Java

Class Name: TwoDArrayDemo

Initialization :

```
int marks[][]= { {85,68,90} , {56,67,78,5},{98,89,85},{84,85} };
```

```
for(int i=0;i<4;i++){  
    System.out.println();  
    for(int j=0;j<marks[i].length;j++){  
        System.out.print(marks[i][j]+" ");  
    }  
}
```



Returns number of
columns in ith row

Note: When initializing 2D array, both the square brackets must be empty.

JAVA FOP : Topics

- **String API**
- **User-defined methods manipulating String objects.**

String API

Java API has three classes to work on strings:

- String
- StringBuffer
- StringBuilder

String class

String is a built-in class in Java. The fully qualified name of String class in Java is **java.lang.String**

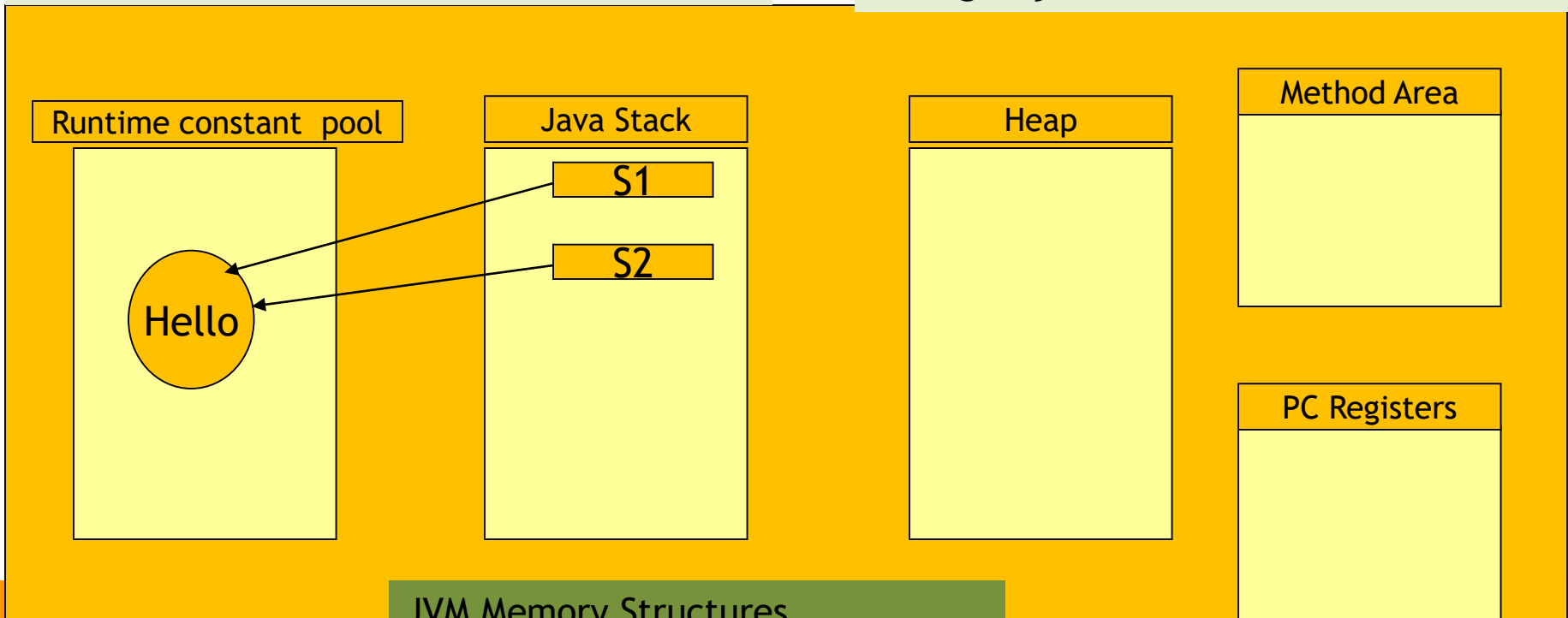
Strings are immutable objects i.e. their values cannot be changed after they are created. Since String objects are immutable they can only be shared but not modified.

String Pool or Run time Constant Pool

- String pool is a fixed block of memory within heap where the string objects are held by JVM.
- If a string object is created directly, using assignment operator as `String s1 = "Hello"`, then it is stored in String constant pool.
- `String s2 = "Hello"`, then `s1 == s2` will be true or false ? **true**

```
System.out.println(s1.hashCode());  
System.out.println(s2.hashCode());
```

Return same hashcode. Two equal String objects return same hashcode



String objects

If a String object is created using new operator. then memory is allocated in the heap.

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

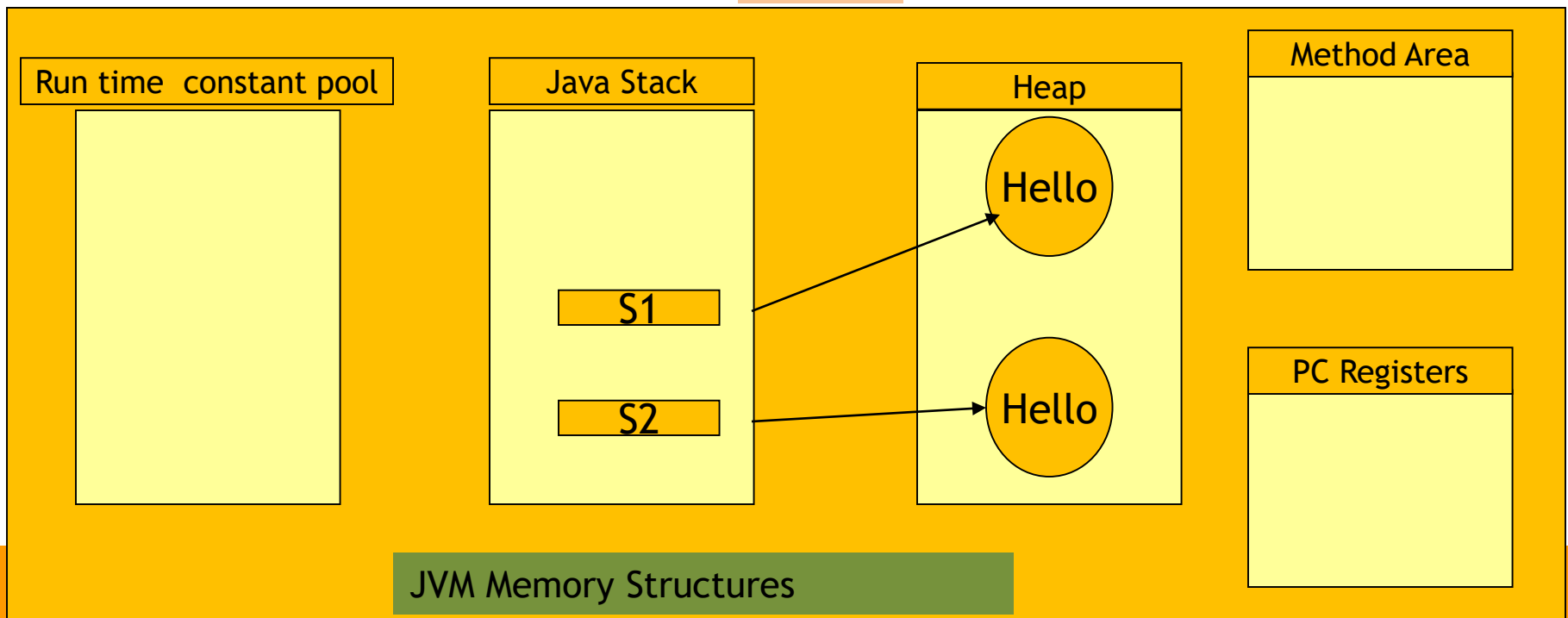
```
String s2 = new String("Hello");
```

```
System.out.println(s1.hashCode());  
System.out.println(s2.hashCode());
```

Return same hashcode. Two equal String objects return same hashcode

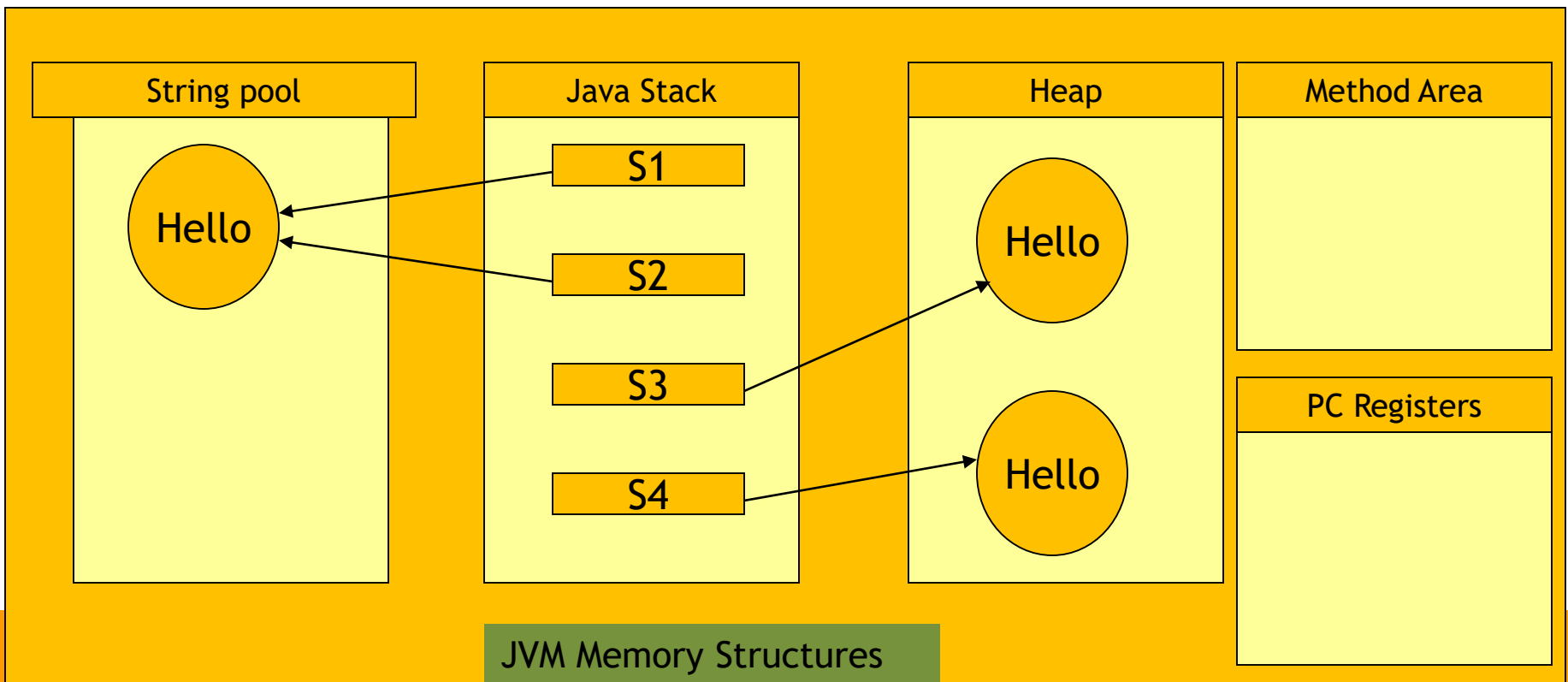
`s1 == s2` will be true or false ?

false



String objects

```
public static void main(String[] args) {  
    String s1 = "Hello"; String s2 = "Hello";  
    System.out.println(s1==s2);  
    System.out.println(s1.equals(s2));  
    String s3 = new String("Hello"); String s4 = new String("Hello");  
    System.out.println(s3==s4);  
    System.out.println(s3.equals(s4));  
    System.out.println(s1.hashCode()+" "+s2.hashCode());  
    System.out.println(s3.hashCode()+" "+s4.hashCode());  
}
```

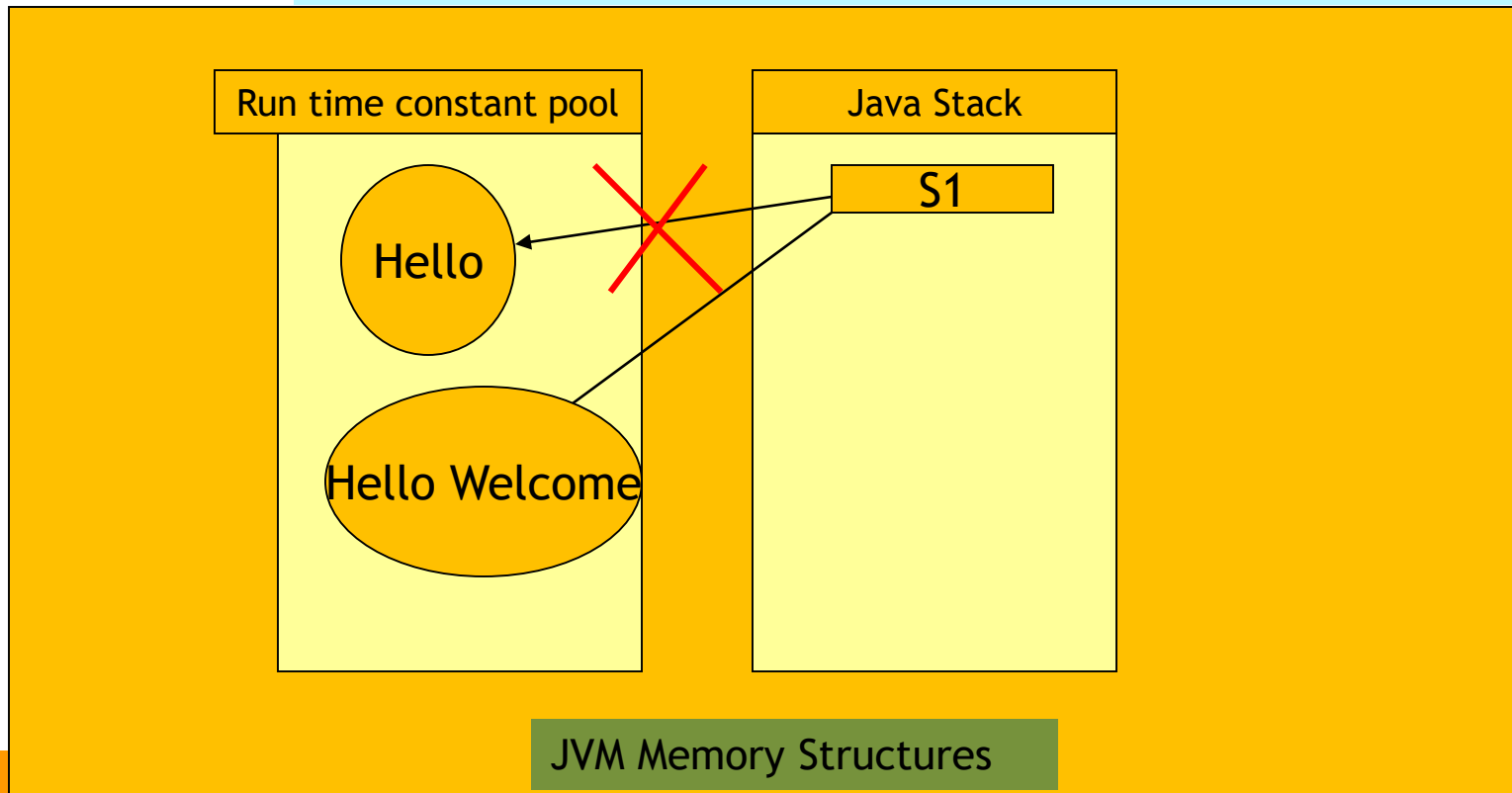


String objects

```
String s1 = "Hello";  
System.out.println(s1);
```

```
s1 = s1 + "welcome";  
System.out.println(s1)
```

If Strings are immutable, how do we justify these statements ?



String Example

```
public class StringTest {  
    public static void main(String[] args) {  
        foo("foo");  
        foo(new String("foo"));  
        foo("bar");  
    }  
  
    private static void foo(String string) {  
        if(string=="foo"){  
            string="1";  
        }else if(string.equals("foo")){  
            string="2";  
        }else if(string.equals("baR")){  
            string="3";  
        }else{  
            string="4";  
        }  
        System.out.println(string);  
    }  
}
```

Common String class methods

char	<u>charAt</u> (int index) Returns the char value at the specified index.
boolean	<u>contains</u> (<u>CharSequence</u> s) Returns true if and only if this string contains the specified sequence of char values.
boolean	<u>equals</u> (<u>Object</u> anObject) Compares this string to the specified object.
boolean	<u>equalsIgnoreCase</u> (<u>String</u> anotherString) Compares this String to another String, ignoring case considerations.
int	<u>indexOf</u> (<u>String</u> str) Returns the index within this string of the first occurrence of the specified substring.
int	<u>length</u> () Returns the length of this string.
<u>String</u>	<u>replace</u> (char oldChar, char newChar) Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
<u>String</u> []	<u>split</u> (<u>String</u> regex) Splits this string around matches of the given <u>regular expression</u> .
<u>String</u>	<u>substring</u> (int beginIndex, int endIndex) Returns a new string that is a substring of this string.
boolean	matches() : Pattern matching

Regular expression

Subexpression	Matches
<code>^</code>	Matches beginning of line.
<code>\$</code>	Matches end of line.
<code>.</code>	Matches any single character except newline. Using <code>m</code> option allows it to match newline as well.
<code>[...]</code>	Matches any single character in brackets.
<code>[^...]</code>	Matches any single character not in brackets
<code>\A</code>	Beginning of entire string
<code>\Z</code>	End of entire string
<code>\Z</code>	End of entire string except allowable final line terminator.
<code>re*</code>	Matches 0 or more occurrences of preceding expression.
<code>re+</code>	Matches 1 or more of the previous thing
<code>re?</code>	Matches 0 or 1 occurrence of preceding expression.
<code>re{ n}</code>	Matches exactly <code>n</code> number of occurrences of preceding expression.
<code>re{ n,}</code>	Matches <code>n</code> or more occurrences of preceding expression.
<code>re{ n, m}</code>	Matches at least <code>n</code> and at most <code>m</code> occurrences of preceding expression.
<code>a b</code>	Matches either <code>a</code> or <code>b</code> .
<code>\w</code>	Matches word characters.
<code>\W</code>	Matches nonword characters.
<code>\s</code>	Matches whitespace. Equivalent to <code>[\t\n\r\f]</code> .
<code>\S</code>	Matches nonwhitespace.
<code>\d</code>	Matches digits. Equivalent to <code>[0-9]</code> .
<code>\D</code>	Matches nondigits.
<code>\A</code>	Matches beginning of string.
<code>\Z</code>	Matches end of string. If a newline exists, it matches just before newline.
<code>\z</code>	Matches end of string.

StringBuffer class

- A **thread-safe, mutable** sequence of characters. A string buffer is like a String, but **can be** modified.
- **StringBuffer** objects are safe for use by multiple threads. This class methods are synchronized.
- The principal operations on StringBuffer are **append** , **insert** and **delete** methods, which are overloaded so as to accept data of any type.

Creating StringBuffer object:

```
StringBuffer customerName=new StringBuffer("Ravi Kumar");
```

StringBuilder class

- StringBuilder class is a copy of StringBuffer class except that StringBuilder class is not thread-safe.
- Hence all the methods of StringBuffer class apply to StringBuilder class.
- This class is designed to replace StringBuffer in place where the string buffer is used by a single thread (as is generally the case).
- When to use what:
 - Use String if you require immutability
 - Use StringBuffer if you need mutable + thread-safety
 - Use StringBuilder if you require mutable + without thread-safety

Date and Time API

Date class

Date class represents a specific instant in time, with millisecond precision.

Constructors

public Date()

Allocates a Date object and initializes it so that it represents the time at which it was allocated, measured to the nearest millisecond.

Ex.

```
Date today = new Date();  
System.out.println(today); //Wed Sep 26 12:18:43 IST 2012
```

public Date(long date)

Allocates a Date object and initializes it to represent the specified number of milliseconds since the standard base time known as “**the epoch**”, namely **January 1, 1970, 00:00:00 GMT**.

Parameter: date in long in milliseconds since January 1, 1970, 00:00:00 GMT

Common Date class methods

- **boolean after(Date when)** Tests if this date is after the specified date
- **boolean before(Date when)** Tests if this date is before the specified date
- **int compareTo(Date anotherDate)** Compares two Dates for ordering
- **boolean equals(Object obj)** Compares two dates for equality.
- **long getTime()** Returns the number of milliseconds since January 1, 1970 00:00:00 GMT represented by this Date object.

SimpleDateFormat class

Date Formatting using SimpleDateFormat:

SimpleDateFormat converts *java.util.Date* to *String* and vice-versa
SimpleDateFormat is a concrete class for formatting and parsing dates in a locale-sensitive manner.

This class is placed in **java.text** package

Date Formats

Letter	Date or Time Component	Presentation	Examples
G	Era designator	Text	AD
y	Year	Year	1996; 96
M	Month in year	Month	July; Jul; 07
w	Week in year	Number	27
W	Week in month	Number	2
D	Day in year	Number	189
d	Day in month	Number	10
F	Day of week in month	Number	2
E	Day in week	Text	Tuesday; Tue
a	Am/pm marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in am/pm (0-11)	Number	0
h	Hour in am/pm (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
Z	Time zone	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC 822 time zone	-0800

Date Formats

Input string	Pattern
----- 2001.07.04 AD at 12:08:56 PDT	yyyy.MM.dd G 'at' HH:mm:ss z
Wed, Jul 4, '01	EEE, MMM d, 'yy
12:08 PM	h:mm a
12 o'clock PM, Pacific Daylight Time	hh 'o''clock' a, zzzz
0:08 PM, PDT	K:mm a, z
02001.July.04 AD 12:08 PM	yyyyy.MMMM.dd GGG hh:mm aaa
Wed, 4 Jul 2001 12:08:56 -0700	EEE, d MMM yyyy HH:mm:ss Z
010704120856-0700	yMMddHHmmssZ
2001-07-04T12:08:56.235-0700	yyyy-MM-dd'T'HH:mm:ss.SSSZ
2001-07-04T12:08:56.235-07:00	yyyy-MM-dd'T'HH:mm:ss.SSSXXX
2001-W27-3	YYYY-'W'ww-u

Calendar Class

`java.util.Calendar`

- The Calendar class is an **abstract class** that provides methods for converting between a specific instant in time and a set of **calendar fields** such as YEAR, MONTH, DAY_OF_MONTH, HOUR, and so on, and for manipulating the calendar fields.
- Calendar provides a class method, `getInstance()` which returns a Calendar object whose calendar fields have been initialized with the current date and time.

GregorianCalendar class

java.util.GregorianCalendar is a concrete subclass of *Calendar* and provides the standard calendar system used by most of the world, supports both the Julian and Gregorian calendar systems.

Constructor Summary

GregorianCalendar()

Constructs a default *GregorianCalendar* using the current time in the default time zone with the default locale.

GregorianCalendar(int year, int month, int dayOfMonth)

Constructs a *GregorianCalendar* with the given date set in the default time zone with the default locale.

GregorianCalendar(int year, int month, int dayOfMonth, int hourOfDay, int minute)

Constructs a *GregorianCalendar* with the given date and time set for the default time zone with the default locale.

GregorianCalendar(int year, int month, int dayOfMonth, int hourOfDay, int minute, int second)

Constructs a *GregorianCalendar* with the given date and time set for the default time zone with the default locale.



Thank You!