



---

# Core Java

---

# Course Objective

At the end of this session, you will be able to:



- Introduce Java Architecture
- Apply Object Oriented Concepts using Java
- Package Classes & Interfaces
- Catch and Throw Exceptions
- Write Custom Exception Classes
- Use I/O Streams in Java
- Illustrate how to make use of the standard Java Class Library & create reusable classes

# Course Agenda

- Java Platform Architecture
- Java Programming Language
- Classes and Objects
- Inheritance and Polymorphism in Java
- Exception Handling
- IO Streams in Java



# Core Java

- Java Basics
-

# Objective

● At the end of this session, you will be able to:

---



- Understand the Java Platform Architecture
- Write programs using variables, expressions, console input / output and arrays
- Write Simple Object Oriented Program using static members
- Implement Composition
- Refer Java API Documentation

# Agenda

- Java Platform Architecture
- Java Programming Basics
- Classes and Objects
- Arrays - One-dimensional and Multidimensional Arrays
- Using Java API Documentation

# Introduction to Java

- A high level programming language
- Operating system independent
- Runs on Java Virtual Machine (JVM)
  - A secure operating environment that runs as a layer on top of the OS
  - A sandbox which protects the OS from malicious code
- Object Oriented Programming language
  - In Java, everything is a class
  - Unlike C++, OOP support is a fundamental component in Java

# Features of Java

- Object Oriented



- Simple

- Compared to earlier OO languages like C++, it is simple

- Robust

- Secure

- Absence of pointers



# Features of Java (Contd...)

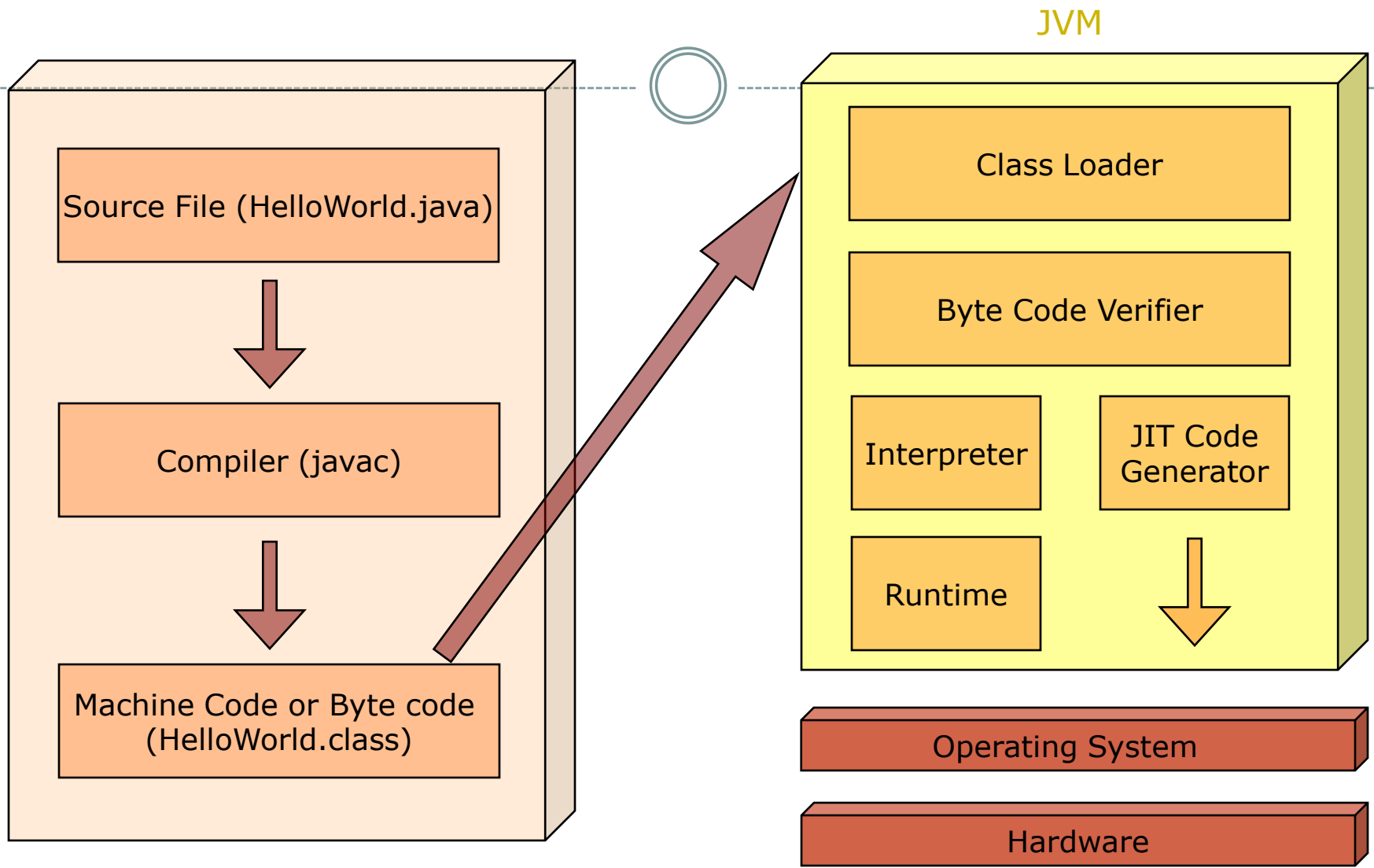
- Support for Multithreading at language level
- Designed to handle Distributed applications
- Architecture Neutral / Portable:
  - Java code compiled on Windows can be run on Unix without recompilation

# Platform Independence

- A platform is the hardware & software environment in which a program runs
- Once compiled, java code runs on any platform without recompiling or any kind of modification

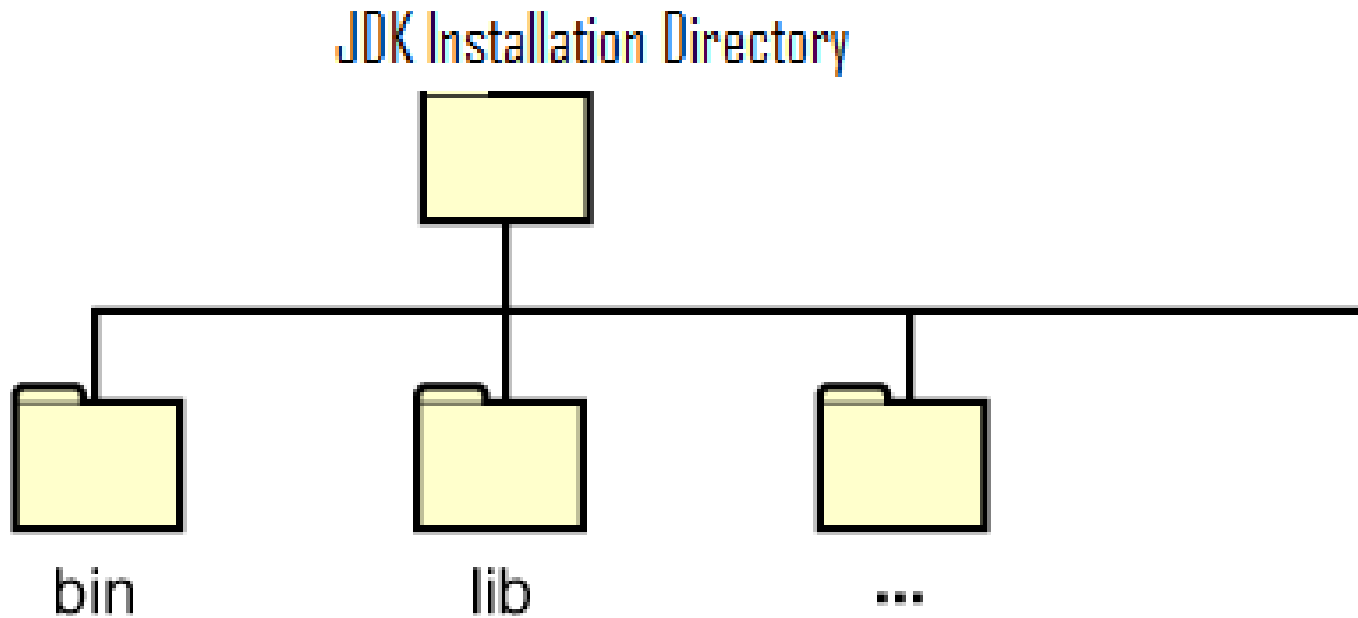
“Write Once Run Anywhere”
- This is made possible by the Java Virtual Machine (JVM)

# Java Architecture



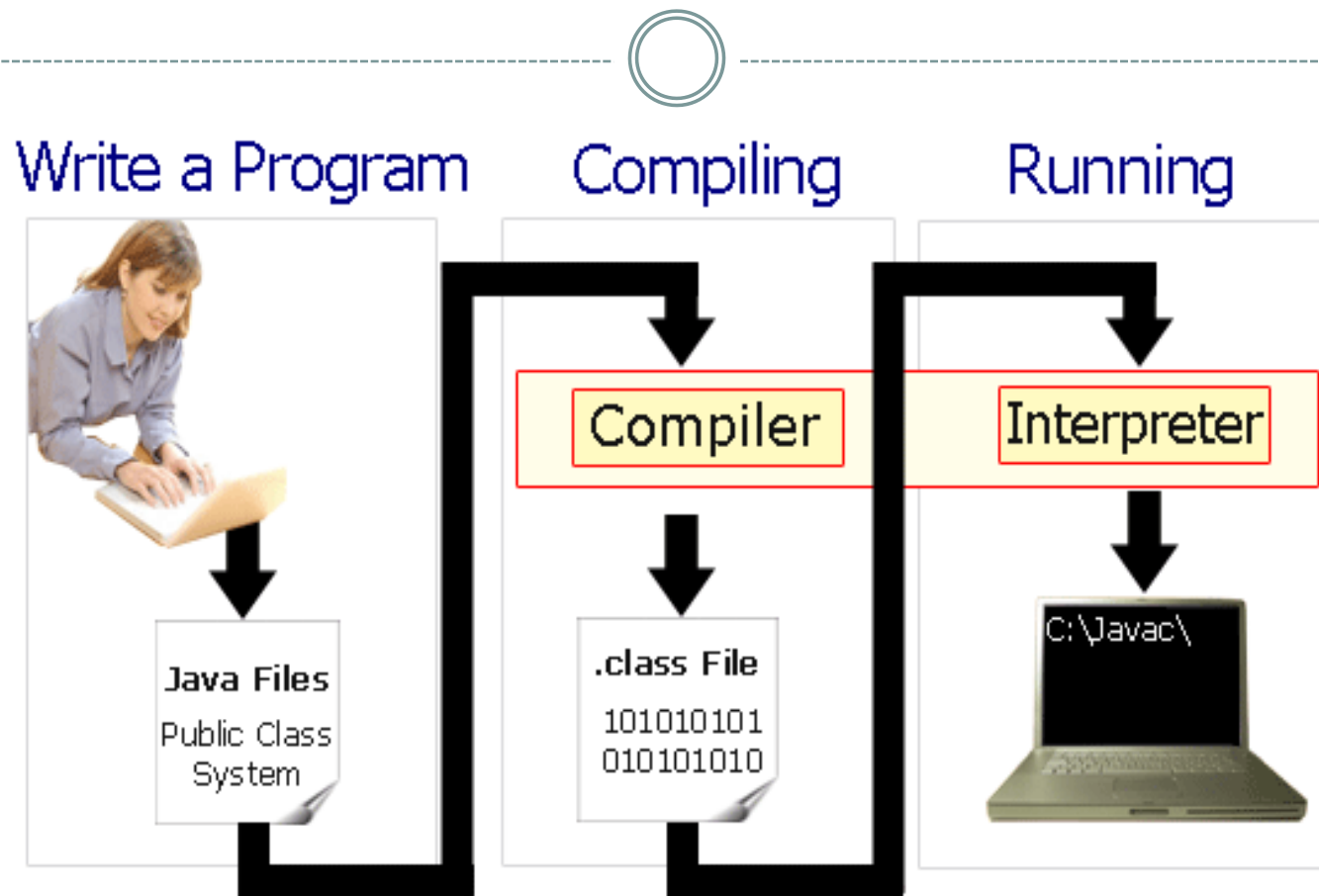
# JDK Directory Structure

- After installing the software, the JDK directory will have the structure as shown



- The *bin* directory contains both, the compiler and the interpreter

# Java Development Process



# Java Virtual Machine (JVM)

- The source code of Java is stored in a text file with the extension **.java**
- The Java compiler compiles a .java file into **byte code**
- The byte code will be in a file with extension **.class**
- The generated .class file is the machine code of this processor
  - Byte code is in binary language
- The byte code is **interpreted** by the JVM

# Java Virtual Machine (JVM) (Contd...)

- JVM makes Java platform independent
- The JVM interprets the .class file to the **machine language** of the underlying platform
- The underlying platform processes the commands given by the JVM

# Environment Variables in JVM

- **JAVA\_HOME:** Java Installation Directory

- Used to derive all other environment variables used by JVM

<b>In Windows</b>	<code>set JAVA_HOME=C:\jdk1.4.3</code>
<b>In UNIX</b>	<code>export JAVA_HOME=/var/usr/java</code>

- **CLASSPATH:**

- Used to locate class files

<b>In Windows</b>	<code>set CLASSPATH=%CLASSPATH%;%JAVA_HOME%\lib\tools.jar;.</code>
<b>In UNIX</b>	<code>set CLASSPATH=\$CLASSPATH:\$JAVA_HOME/lib/tools.jar</code>



# (Contd...)

## ● PATH

- Used by OS to locate executable files

<b>In Windows</b>	<code>set PATH=%PATH%;%JAVA_HOME%\bin</code>
<b>In UNIX</b>	<code>export PATH=\$PATH:\$JAVA_HOME/bin</code>

# Source File Layout - Hello World

- Type the source code using any text editor

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

- Save this file as *HelloWorldApp.java*

# To Compile

- Open the command prompt
- Set the environment variables
- Go to the directory in which the program is saved
- Type - **javac HelloWorldApp.java**
  - If it says, “bad command or file name” then check the path setting
  - If it returns to prompt without giving any message, it means that compilation is successful

## To Execute

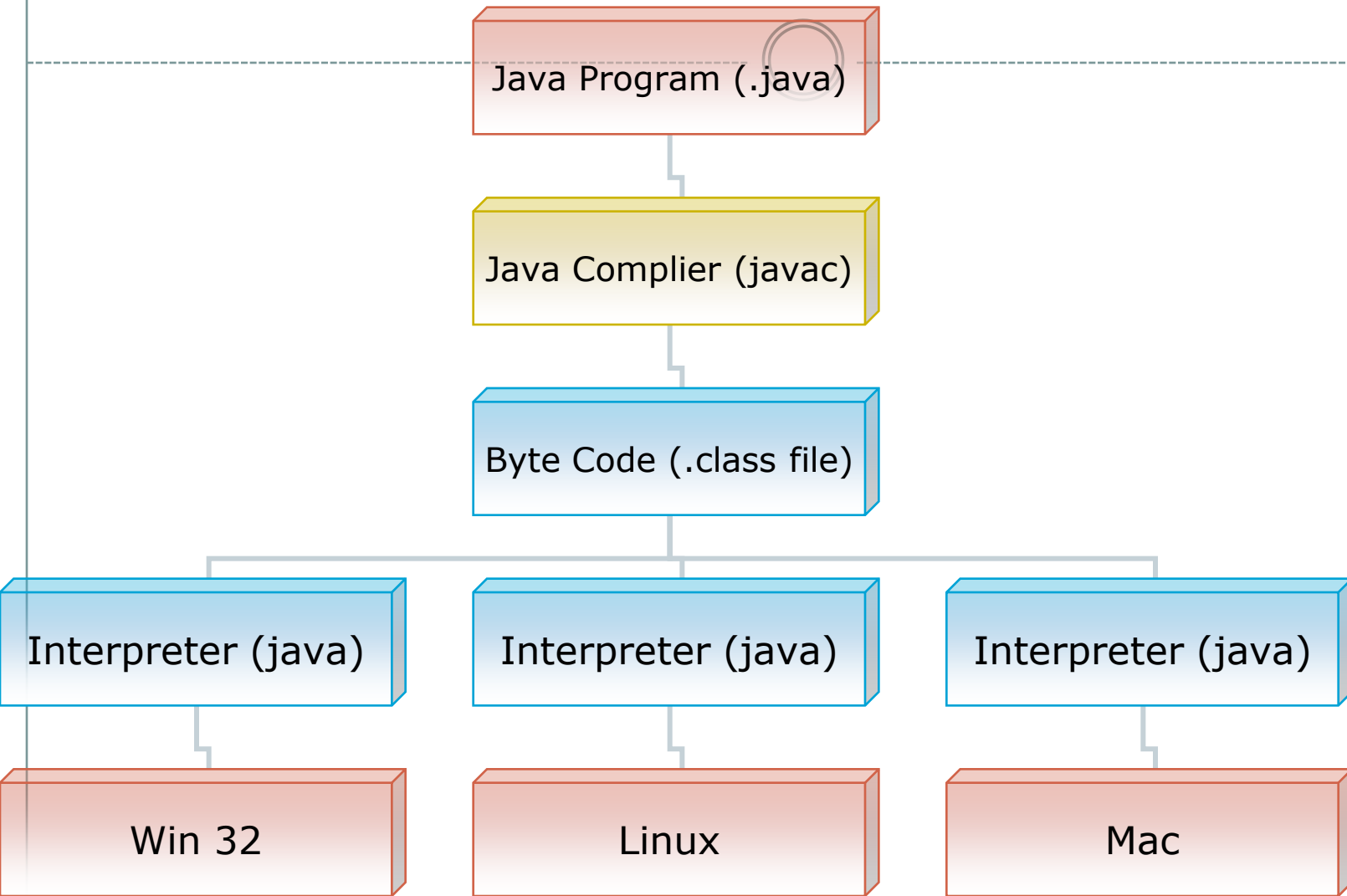
- Type the command - **java HelloWorldApp**
- The result will be



The image shows a screenshot of an MS-DOS Prompt window. The title bar is blue and contains the text 'MS-DOS Prompt' and standard window control buttons (minimize, maximize, close). Below the title bar is a menu bar with icons for file operations (New, Open, Save, Print, etc.). The main area of the window is black with white text. It shows the command prompt 'C:\java>' followed by the command 'java HelloWorldApp'. The output of the command is 'Hello World!'. The prompt 'C:\java>' is visible again at the bottom of the window.

```
C:\java>java HelloWorldApp
Hello World!
C:\java>
```

# Compilation & Execution



# Best Practices

- Only put one class in one source file
- Provide adequate comments in the program
- Properly indent the program
- Follow coding standards for identifiers



Java Coding  
Standards

# Java Keywords

<code>abstract</code>	<code>*const</code>	<code>finally</code>	<code>implements</code>	<code>public</code>	<code>this</code>
<code>boolean</code>	<code>continue</code>	<code>for</code>	<code>instanceof</code>	<code>throw</code>	<code>transient</code>
<code>break</code>	<code>float</code>	<code>if</code>	<code>null</code>	<code>short</code>	<code>void</code>
<code>byte</code>	<code>default</code>	<code>import</code>	<code>int</code>	<code>super</code>	<code>volatile</code>
<code>case</code>	<code>do</code>	<code>false</code>	<code>return</code>	<code>switch</code>	<code>while</code>
<code>catch</code>	<code>double</code>	<code>interface</code>	<code>package</code>	<code>synchronized</code>	
<code>char</code>	<code>else</code>	<code>long</code>	<code>private</code>	<code>static</code>	
<code>class</code>	<code>extends</code>	<code>*goto</code>	<code>protected</code>	<code>try</code>	
<code>true</code>	<code>final</code>	<code>new</code>	<code>native</code>	<code>throws</code>	

\* Keywords not in use now

# Java Identifiers

- Declared entities such as variables, methods, classes & interfaces are Java Identifiers
- Must begin with a letter, underscore (\_) or dollar sign (\$)
- May contain letters, digits, underscore(\_) & dollar sign (\$)



# Data Types in Java

- Java is a **strongly typed** language
  - Unlike C, type checking is strictly enforced at run time
  - Impossible to typecast incompatible types
- Data types may be:
  - Primitive data types
  - Reference data types

# Primitive Data Types in Java



## ***Integer Data Types***

byte	(1 byte)
short	(2 bytes)
int	(4 bytes)
long	(8 bytes)

## ***Floating Data Types***

float	(4 bytes)
double	(8 bytes)

## ***Character Data Types***

char	(2 bytes)
------	-----------

## ***Logical Data Types***

boolean	(1 bit) (true/false)
---------	----------------------

- All numeric data types are signed
- The size of data types remain same on all platforms
- *char* data type is 2 bytes as it uses the UNICODE character set. And so, Java supports internationalization

# Variables

- A named storage location in the computer's memory that stores a value of a particular type for use by program.



- Example of variable declaration:

<code>DataType</code>	<code>variableName</code>
<code>int</code>	<code>myAge, cellPhone;</code>
<code>double</code>	<code>salary;</code>
<code>char</code>	<code>tempChar;</code>

- The data type can either be:

- built-in *primitive* types (e.g. int, double, char object classes)
- *reference* data types (e.g. String, BufferedReader)

- Naming Convention →

**Variable Name:** First word lowercase & rest initial capitalized (**Camel Casing**)

e.g. `thisIsALongVariableName`

# Variables (Contd...)

- Using primitive data types is similar to other languages

```
int count;  
int max=100;
```



- Variables can be declared anywhere in the program

```
for (int count=0; count < max; count++) {  
    int z = count * 10;  
}
```

## BEST PRACTICE

Declare a variable in program only when required

Do not declare variables upfront like in C

- In Java, if a local variable is used without initializing it, the compiler will show an error

# Give this a Try...

## How many of these are valid Java Identifiers?



78class

User\$ID

False

Hello!

Class87

Jump\_Up\_

Private

First One

sixDogs

DEFAULT\_VAL

Average-Age

String

A. 5

B. 6

C. 7

D. 8

E. 9

## Give this a Try...

- What will be the output of the following code snippet when you try to compile and run it?

```
class Sample{  
    public static void main (String args[]){  
        int count;  
        System.out.println(count);  
    }  
}
```

# Comments in Java


- A single line comment in Java starts with //

// This is a single line comment in Java

- A multi line comment starts with /\* & ends with \*/

```
/* This is a multi line  
comment  
in Java */
```

# Reference Data Types

- Hold the reference of dynamically created objects which are in the heap 
- Can hold three kinds of values:
  - **Class type:** Points to an object / class instance
  - **Interface type:** Points to an object, which is implementing the corresponding interface
  - **Array type:** Points to an array instance or “*null*”
- Difference between Primitive & Reference data types:
  - Primitive data types hold values themselves
  - Reference data types hold reference to objects, i.e. they are not objects, but reference to objects



# Reference Data Types (Contd...)

- Objects & Arrays are accessed using *reference variables* in Java

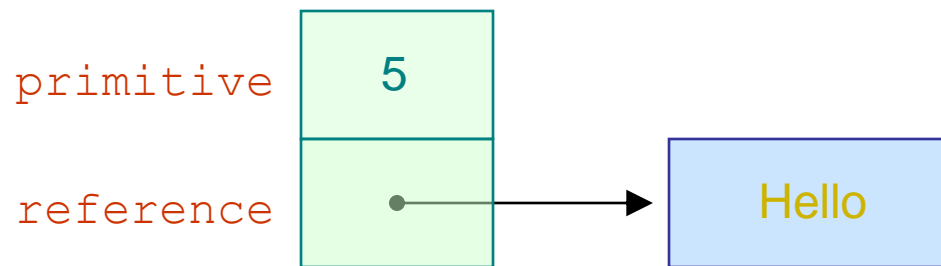
- A reference variable is similar to a pointer (stores memory address of an object)

- Java does not support the explicit use of addresses like other languages

- Java does not allow pointer manipulation or pointer arithmetic

```
int primitive = 5;  
String reference = "Hello" ;
```

- **Memory Representation:**



## Reference Data Types (Contd...)

- A reference type cannot be cast to primitive type
- A reference type can be assigned 'null' to show that it is not referring to any object

# Typecasting Primitive Data Types

- Automatic type changing is known as *Implicit Conversion*

- A variable of smaller capacity can be assigned to another variable of bigger capacity

```
int i = 10;  
double d;  
d = i;
```

- Whenever a larger type is converted to a smaller type, we have to explicitly specify the *type cast operator*

```
double d = 10  
int i;  
i = (int) d;
```

- This prevents *accidental loss* of data

# Java Operators

- Used to manipulate primitive data types
- Classified as unary, binary or ternary
- Following are different operators in Java:
  - Assignment
  - Arithmetic
  - Relational
  - Logical
  - Bitwise
  - Compound assignment
  - Conditional

# Java Operators (Contd...)

Assignment Operators

=

Arithmetic Operators

-   +   \*   /   %   ++  
--

Relational Operators

> <   >=   <=   ==   !=

Logical Operators

&&   ||   &   |   !   ^

Bit wise Operator

&   |   ^   >>   >>>

Compound Assignment Operators

+=   -=   \*=   /=   %=  
<<=   >>=   >>>=

Conditional Operator

?:

## Precedence & Associativity of Java Operators

- Decides the order of evaluation of operators



- Click below to check all Java operators from highest to lowest precedence, along with their associativity



Precedence and  
Operators in Java

## Give this a Try...

- What is the result of the following code fragment?



```
int x = 5;  
int y = 10;  
int z = ++x * y--;
```

# Control Structures

- Work the same as in C / C++



if/else, for, while, do/while, switch

```
i = 0;
while(i < 10) {
    a += i;
    i++;
}
```

```
for(i = 0; i < 10; i++) {
    a += i;
}
```

```
i = 0;
do {
    a += i;
    i++;
} while(i < 10);
```

```
if(a > 3) {
    a = 3;
}
else {
    a = 0;
}
```

```
switch(i) {
    case 1:
        string = "foo";
    case 2:
        string = "bar";
    default:
        string = "";
}
```



# Control Structures (Contd...)

- Java supports **continue** & **break** keywords also
- Again, work very similar to as in C / C++
- Switch statements require the condition variable to be a char, byte, short or int

```
for(i = 0; i < 10; i++) {  
    if(i == 5)  
        continue;  
    a += i;  
}
```

```
for(i = 0; i < 10; i++) {  
    a += i;  
    if(a > 100)  
        break;  
}
```

## Give this a Try...

What do you think is the output if aNumber is 3?



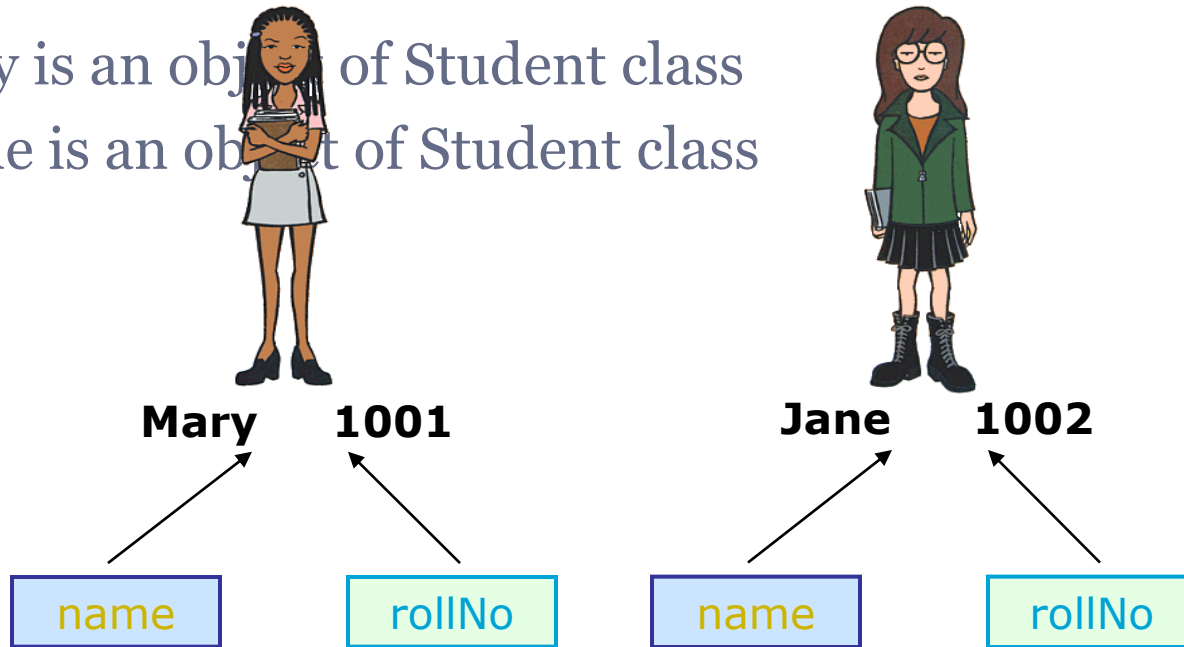
```
if (aNumber >= 0) {  
  
    if (aNumber == 0)  
        System.out.println("first string");  
    else  
        System.out.println("second string");  
        System.out.println("third string");  
}
```

# Concept of Class

- A **class** is a description of a group of objects with common properties (attributes) & behavior (operations)

- An object is an instance of a class

e.g. Mary is an object of Student class  
Jane is an object of Student class



# Constituents of a Class

```
public class Student {  
    private int rollNo;  
    private String name;
```

Data Members  
(State)

```
        Student() {  
            //initialize data members  
        }  
        Student(String nameParam) {  
            name = nameParam;  
        }  
        public int getrollNo () {  
            return rollNo;  
        }  
    }
```

Constructor

Method  
(Behavior)

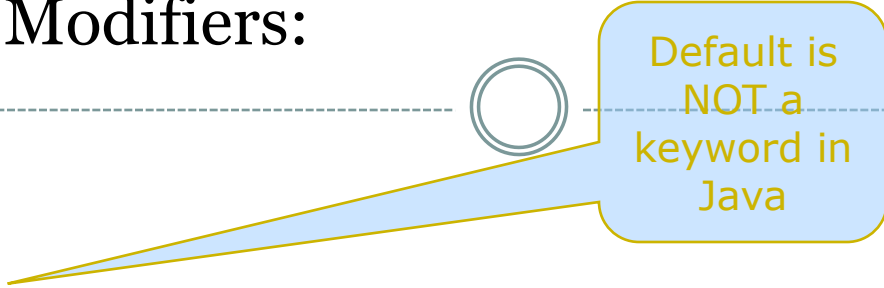
The main method may or may not be present depending on whether the class is a starter class

**Naming Convention** → Class Name: First letter Capital

# Access Modifiers – Private & Public

- Four Access Modifiers:

- Private
- Protected
- Public
- Default



Default is  
NOT a  
keyword in  
Java

- Data members are always kept **private**

- Accessible only within the class

- The methods which expose the behavior of the object are kept **public**

- However, we can have helper methods which are private

- Key features of Object Oriented Programs

- Encapsulation (code & data bound together)
- State (data) is hidden & Behavior (methods) is exposed to external world

# Creating Objects

- The *new* operator creates a object & returns a reference to it
- Memory allocation of objects happens in the heap area
- Reference returned can be stored in reference variables

```
Student obj1;
```

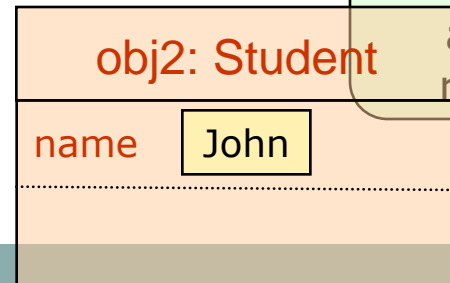
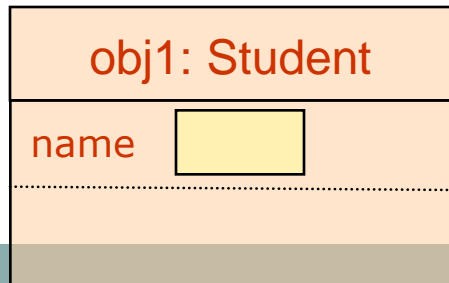
```
obj1 = new Student();
```

Or

```
Student obj2 = new  
Student("John");
```

*obj1* is a reference variable

*new* keyword creates an object and returns a reference to it



# Constructors

- Special methods used to initialize a newly created object
- Called just after memory is allocated for an object
- Initialize objects to required or default values at the time of object creation
- Not mandatory to write a constructor for each class
- A constructor
  - Has the same name as that of the class
  - Doesn't return any value, not even *void*
  - May or may not have parameters (arguments)
- If a class does not have any constructor, the default constructor is automatically added

## Constructors (Contd...)

- In the absence of a user defined constructor, the compiler initializes member variables to its default values
  - Numeric data types are set to 0
  - Char data types are set to null character ('\0')
  - Reference variables are set to *null*



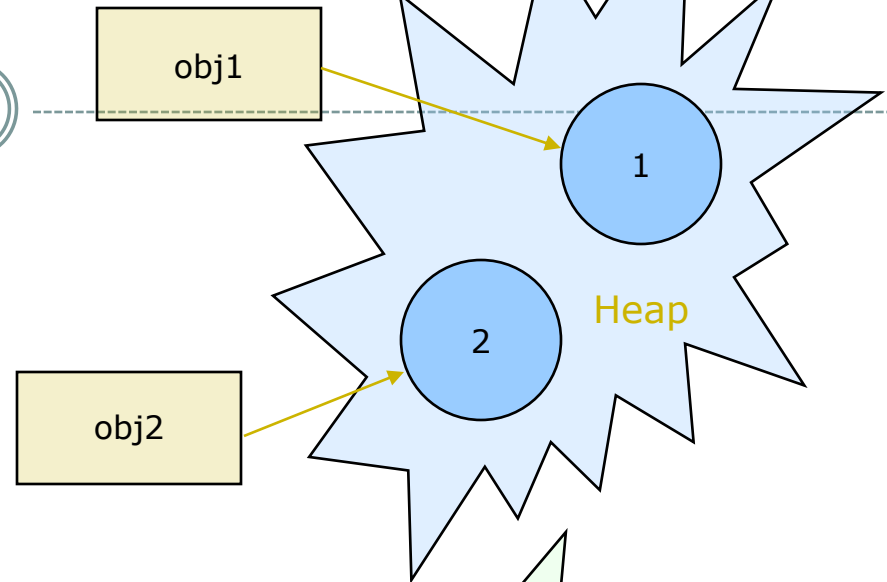
# Lifetime of Objects

```
Student obj1 = new student();  
Student obj2 = new student();
```

Both Student objects now live on the heap

→ References : 2

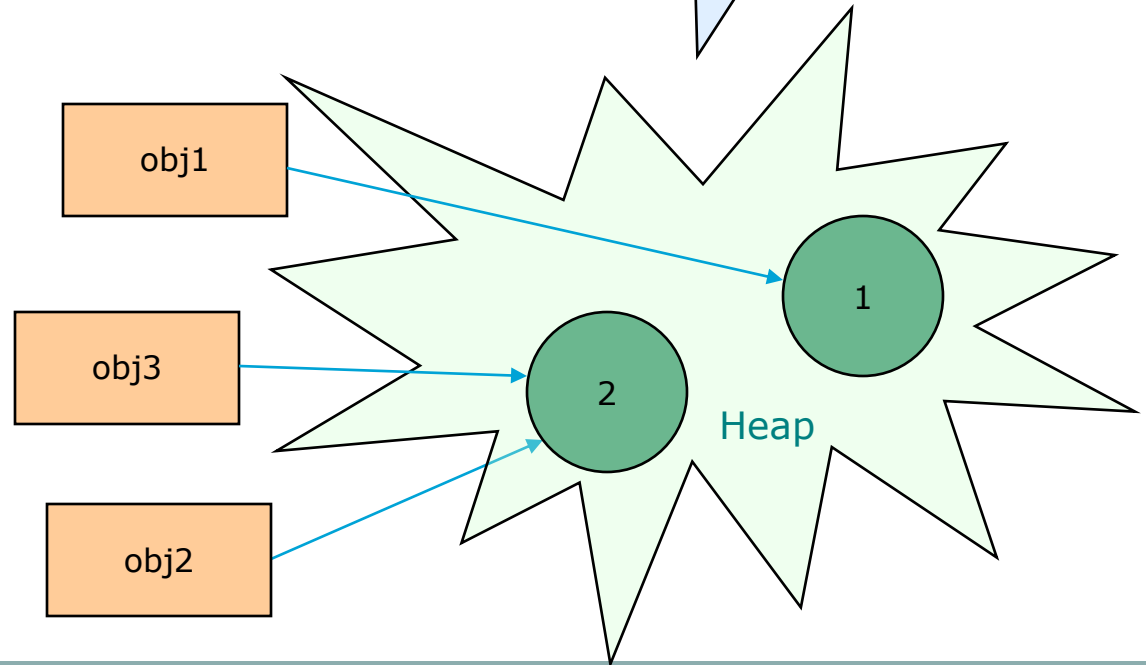
→ Objects : 2



```
Student obj3 = obj2;
```

→ References : 3

→ Objects : 2

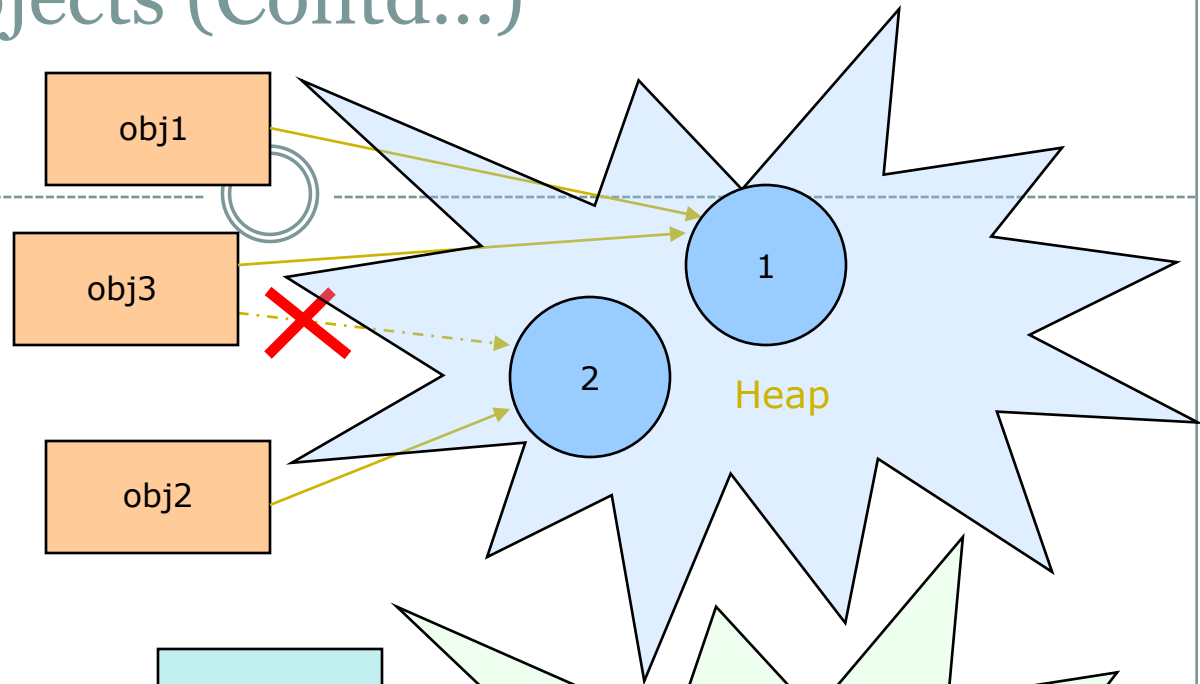


# Lifetime of Objects (Contd...)

```
obj3 = obj1;
```

→ References : 3

→ Objects : 2



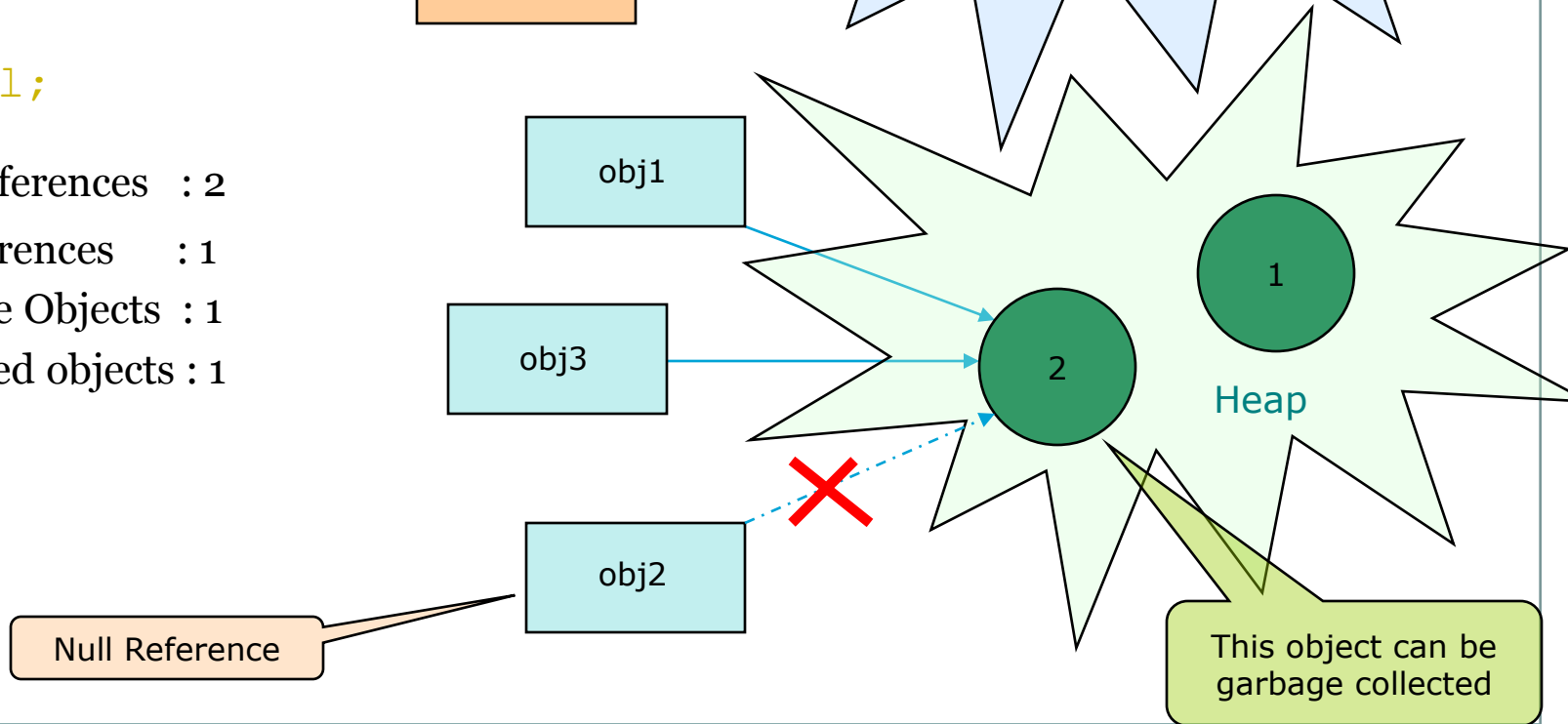
```
obj2 = null;
```

→ Active References : 2

→ Null References : 1

→ Reachable Objects : 1

→ Abandoned objects : 1



# Garbage Collection

- In C, it is the programmer's responsibility to de-allocate the dynamically allocated memory using the *free()* function
- JVM automatically de-allocates memory (Garbage Collection)
- An object which is not referred by any reference variable is removed from memory by the Garbage Collector
- Primitive types are not objects & cannot be assigned *null*

# Scope of Variables

- **Instance Variables (also called Member Variables)**

- Declared inside a class
- Outside any method or constructor
- Belong to the object
- Stored in heap area with the object to which they belong to
- Lifetime depends on the lifetime of object

- **Local Variables (also called Stack Variables)**

- Declared inside a method
- Method parameters are also local variables
- Stored in the program stack along with method calls and live until the call ends

# Scope of Variables (Contd...)

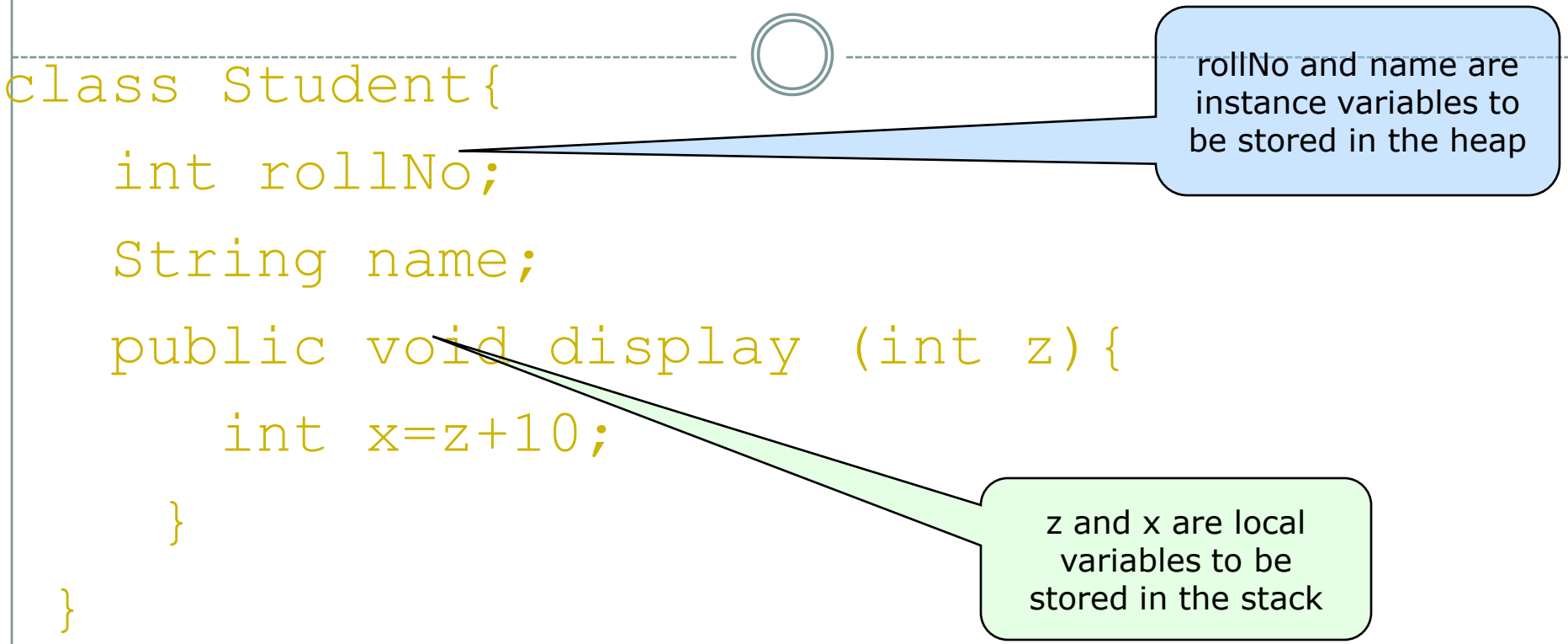
- If we don't initialize instance variables explicitly, they are awarded predictable *default initial values*, based only on the type of the variable

Type	Default Value
boolean	false
byte	(byte) 0
short	(short) 0
int	0
long	0L
char	\u0000
float	0.0f
double	0.0d
object reference	null

- Local variables are not initialized implicitly

# Scope of Variables (Contd...)

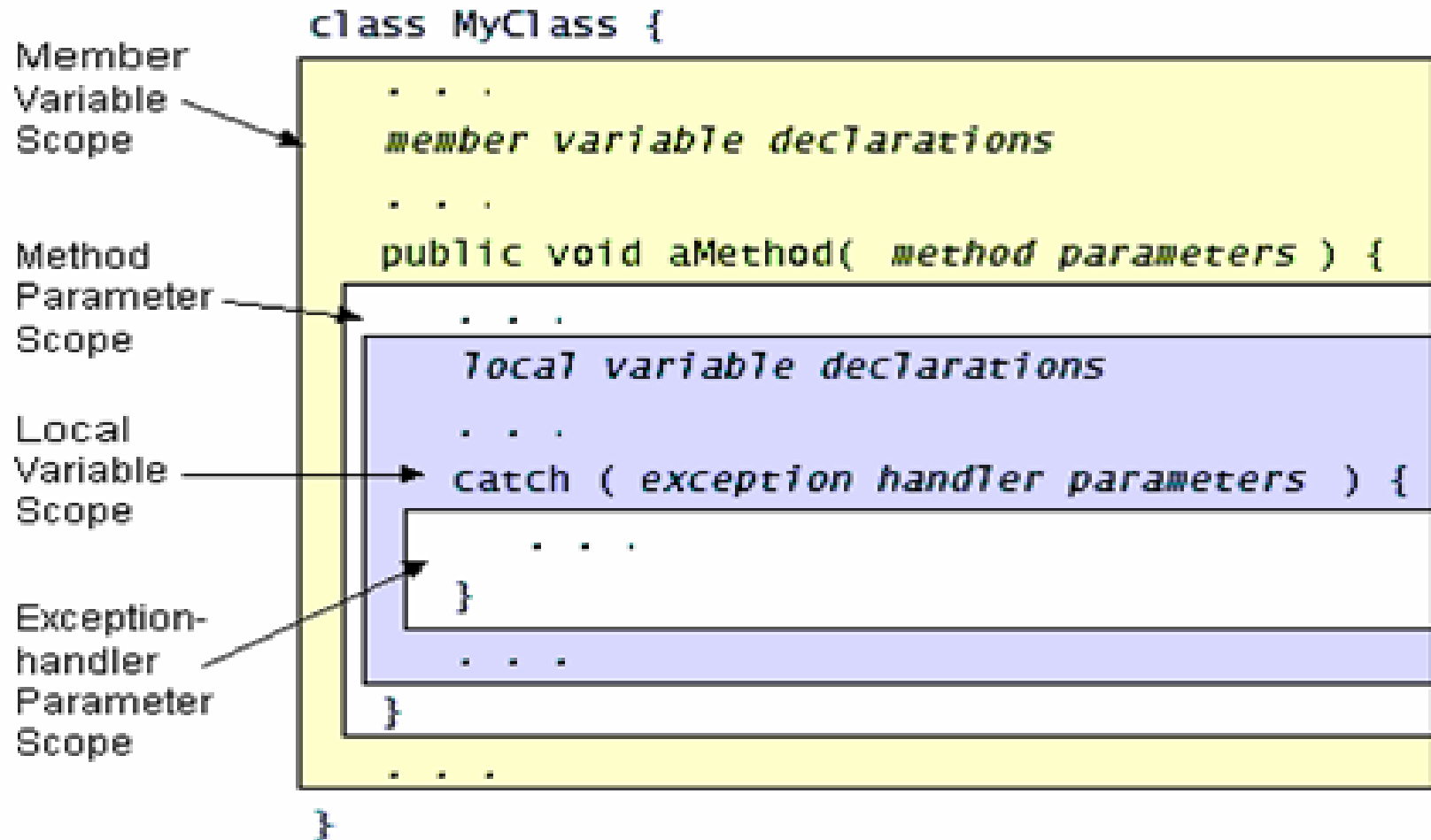
```
class Student{  
    int rollNo;  
    String name;  
    public void display (int z) {  
        int x=z+10;  
    }  
}
```



rollNo and name are  
instance variables to  
be stored in the heap

z and x are local  
variables to be  
stored in the stack

# Scope of Variables (Contd...)



# Arrays in Java

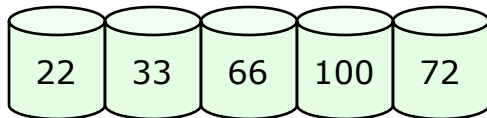
- A data structure which defines an ordered collection of a fixed number of homogeneous data elements



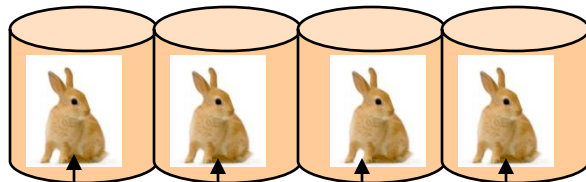
- Size is fixed and cannot increase to accommodate more elements

- Arrays in Java are objects and can be of primitive data types or reference variable type

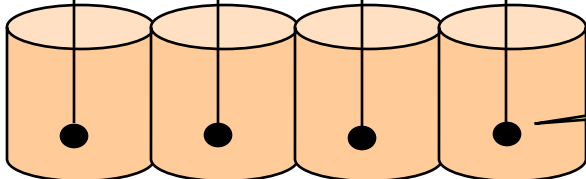
- All elements in the array must be of the same data type



An array holding 5 int elements



An array holding 4 rabbit objects



Reference to Rabbit Object



# Arrays in Java (Contd...)

- *Reference variables* are used in Java to store the references of objects created by the operator *new*

- Any one of the following syntax can be used to create a reference to an *int* array

```
int x[];  
int [] x;
```

- The reference *x* can be used for referring to any *int* array

```
//Declare a reference to an int array
```

```
int [] x;
```

```
//Create a new int array and make x refer to it
```

```
x = new int[5];
```

## Arrays in Java (Contd...)

- The following statement also creates a new *int* array and assigns its reference to x

```
int [] x = new int[5];
```

- In simple terms, references can be seen as names of an array

# Initializing Arrays

- An array can be initialized while it is created as follows:



```
int [] x = {1, 2, 3, 4};
```

```
char [] c = {'a', 'b', 'c'};
```

# Length of an Array

- Unlike C, Java checks the boundary of an array while accessing an element in it
- Programmer not allowed to exceed its boundary
- And so, setting a for loop as follows is very common:

```
for(int i = 0; i < x.length; ++i){  
    x[i] = 5;  
}
```

This works for  
any size array

use the **.length** attribute of an array to control the *for* loop

# Multidimensional Arrays

- A Multi-dimensional array is an array of arrays
- To declare a multidimensional array, specify each additional index using another set of square

```
bracket int [][] x;  
//x is a reference to an array of int arrays  
x = new int[3][4];  
//Create 3 new int arrays, each having 4 elements  
//x[0] refers to the first int array, x[1] to the second and  
so on  
//x[0][0] is the first element of the first array  
//x.length will be 3  
//x[0].length, x[1].length and x[2].length will be 4
```

# Command Line Arguments

- Information that follows program's name on the command line when it is executed

- This data is passed to the application in the form of String arguments

```
class Echo {  
    public static void main (String args[]) {  
        for (int i = 0; i < args.length; i++)  
            System.out.println(args[i]);  
    }  
}
```

Try this: Invoke the Echo application as follows

```
C:\> java Echo Drink Hot Java
```

```
Drink
```

```
Hot
```

```
Java
```

# Using *static*

- *static* keyword can be used in three scenarios:

- For class variables
- For methods
- For a block of code

# Using *static* (Contd...)

- *static variable*

- Belongs to a class
- A single copy to be shared by all instances of the class
- Creation of instance not necessary for using static variables
- Accessed using *<class-name>.<variable-name>* unlike instance variables which are accessed as *<object-name>.<variable-name>*

- *static method*

- It is a class method
- Accessed using *class name.method name*
- Creation of instance not necessary for using static methods
- A static method can access only other static data & methods, and not non-static members



# Using *static* (Contd...)

```
Class Student {
```

```
private int rollNo;
```

```
private static int studCount;
```

```
public Student(){
```

```
    studCount++;
```

```
}
```

```
public void setRollNo (int r){
```

```
    rollNo = r;
```

```
}
```

```
public int getRollNo (int r){
```

```
    return rollNo;
```

```
}
```

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

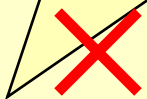
```
    System.out.println("RollNo of the Student is;" + rollNo);
```

```
}
```

The static studCount variable is initialized to 0, ONLY when the class is first loaded, NOT each time a new instance is made

Each time the constructor is invoked, i.e. an object gets created, the static variable studCount will be incremented thus keeping a count of the total no of Student objects created

Which Student? Whose rollNo? A static method cannot access anything non-static



Compilation Error

# Using *static* (Contd...)

- ***static block***: A block of statement inside a Java class that is executed when a class is first loaded & initialized
  - A class is loaded typically after the JVM starts
  - Sometimes a class is loaded when the program requires it

```
class Test{  
    static {  
        //Code goes here  
    }  
}
```

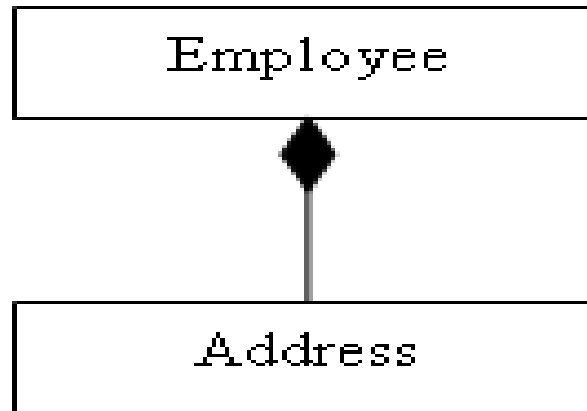
- A static block helps to initialize the static data members like constructors help to initialize instance members

# Implementing Composition

- Composition is simply using instance variables that are references to other objects

For example:

```
class Address {  
    //...}  
  
class Employee {  
    private Address addr = new  
Address();    //...}
```



# Referring Java Documentation

- Java provides a rich set of library classes
- Java API Documentation provides detailed help on all classes
- Browse Java API Documentation

The screenshot shows a Microsoft Internet Explorer window titled "Overview (Java 2 Platform SE v1.4.2) - Microsoft Internet Explorer". The address bar shows the path "C:\Documents and Settings\sj0016216\Desktop\docs\api\index.html". The page content is the "Java™ 2 Platform, Standard Edition, v 1.4.2 API Specification". The left sidebar lists "All Classes" and "Packages" including [java.applet](#), [java.awt](#), and various [Accessible](#) classes. The main content area has tabs for "Overview", "Package", "Class", "Use", "Tree", "Deprecated", "Index", and "Help". The "Overview" tab is selected, showing the title "Java™ 2 Platform, Standard Edition, v 1.4.2 API Specification" and a description: "This document is the API specification for the Java 2 Platform, Standard Edition, version 1.4.2." Below this, there is a section "See:" with a link to "Description". A table titled "Java 2 Platform Packages" lists several packages and their descriptions:

Java 2 Platform Packages	
<a href="#">java.applet</a>	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
<a href="#">java.awt</a>	Contains all of the classes for creating user interfaces and for painting graphics and images.
<a href="#">java.awt.color</a>	Provides classes for color spaces.
<a href="#">java.awt.datatransfer</a>	Provides interfaces and classes for transferring data between and within applications.
<a href="#">java.awt.dnd</a>	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
<a href="#">java.awt.event</a>	Provides interfaces and classes for dealing with different types of events fired by AWT components.

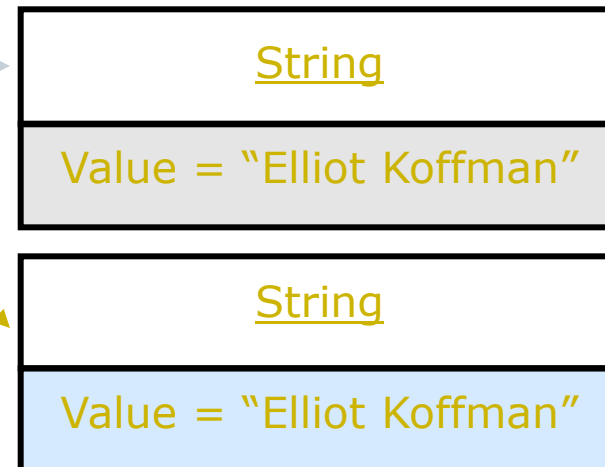
# Using *String* Class

- Present in *java.lang* package
- An object of the String class represents a fixed length, immutable sequence of characters
- Has overridden equals( ) method of the Object class that should be used to compare the actual string values
- A lot of other string manipulation methods are available
- JavaDocs can be referred for a detailed list of methods

# Using *String* Class (Contd...)


- Defines a data type used to store a sequence of characters
- Strings are objects
- **String objects can't be modified:**
  - If attempted to do so, Java creates a new object having the modified character sequence

```
String myName = "Elliot Koffman";  
myName = "Koffman, Elliot";
```



# Common String Operations

- **String concatenation**



```
String u = "Hello";  
String t = " World";  
String s = u + t; // s refers to "Hello World"  
  
int i = s.length(); // returns 11  
  
u.equals(t)          // comparison, returns false  
u.compareTo(t)       // returns negative number  
  
s.charAt(1)          // returns 'e', index runs  
                      // from 0 to length-1  
  
String x = u.toUpperCase(); // returns "HELLO"
```

- **Many more, check String class in Java Docs**

# Using *StringBuffer* Class

- Present in *java.lang* package
- Unlike class *String*, *StringBuffer* represents a string that can be *dynamically modified*
- String buffer's capacity can be dynamically increased even though its initial capacity is specified
- Should be used while manipulating strings like appending, inserting, and so on



StringBufferAppend.java



# Summary

- In this session, we have covered:



- Java Architecture
- Features of Java
- Data types and Operators in Java
- Classes and Objects
- Garbage Collection
- Using Java Arrays
- Referring Java Documentation

Thank You

---