Yadab Raj Ojha

Sr. Java Developer / Lecture
Email: yadabrajojha@gmail.com
Blog: http://yro-tech.blogspot.com/
Java Tutorial: https://github.com/yrojha4ever/JavaStud
LinkedIn: https://www.linkedin.com/in/yrojha
Twitter: https://twitter.com/yrojha4ever

Part: 1

Courses Outline- Core Java

Introduction

- Java language & platform
- History and features
- Java PC set up, about JDK/JRE/JVM
- Environment variable and command interface
- Compiling and Running first Hello World Program

Syntax & Grammar

- Lexical Structure
- Naming Conventions
- Comments and Formatting
- Command Line Arguments
- Data Types and Variables
- Literal Constants
- Operators and Expressions
- Getter/Setter method concept

Class Libraries

- Type Wrappers
- Type Conversion and Casting
- System Class & Math Class
- Locale, Date & Calendar Class
- DateFormat, SimpleDateFormat Class
- NumberFormat Class
- DecimalFormat Class
- BigInteger, BigDecimal class

String Manipulation

- String Class and its Methods
- StringBuffer Class
- StringBuilder Class

- String Tokenizers
- Regular Expressions
- String Applications
- Object class and override its methods.

Control Flow and Enum Constants

- Conditions, Statements, Blocks
- Conditional Statements
- Loops and Switches
- Continue, Break and Return
- Array and Arrays class
- Enum Types and its application

Chapter 1

Introduction, Feature and History

What is Java

Java is a **programming language** and computing **platform**.

Java is a high level, robust, secured and object-oriented programming language.

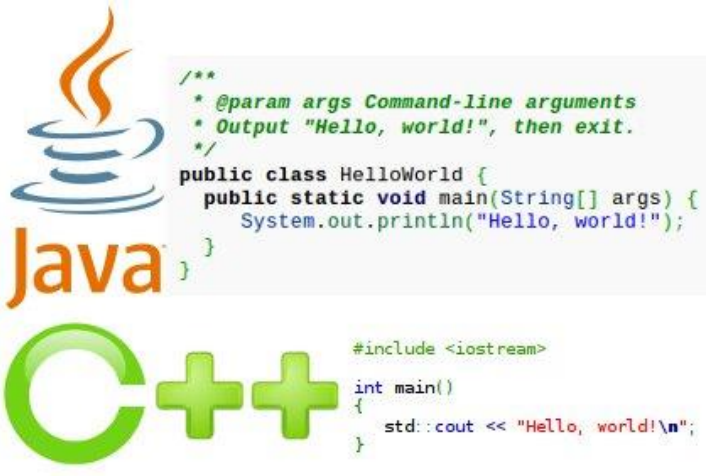
Platform: Any hardware or software environment in which a program runs, is known as a platform. Since Java has its own runtime environment (JRE) and API, it is called platform.

Java is intended to let application developers "write once, run anywhere" . Compiled Java code can run on all platforms that support Java without the need for recompilation. Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of computer architecture.

Advantage of Java:

Platform Independence: You can use the same code on Windows, Solaris, Linux, Macintosh, and so on.

Syntax: Java has a syntax similar to that of C++, making it easy for C and C++ programmers to learn.



Java is also fully object oriented—even more so than C++. Everything in Java, except for a few basic types like numbers, is an object.

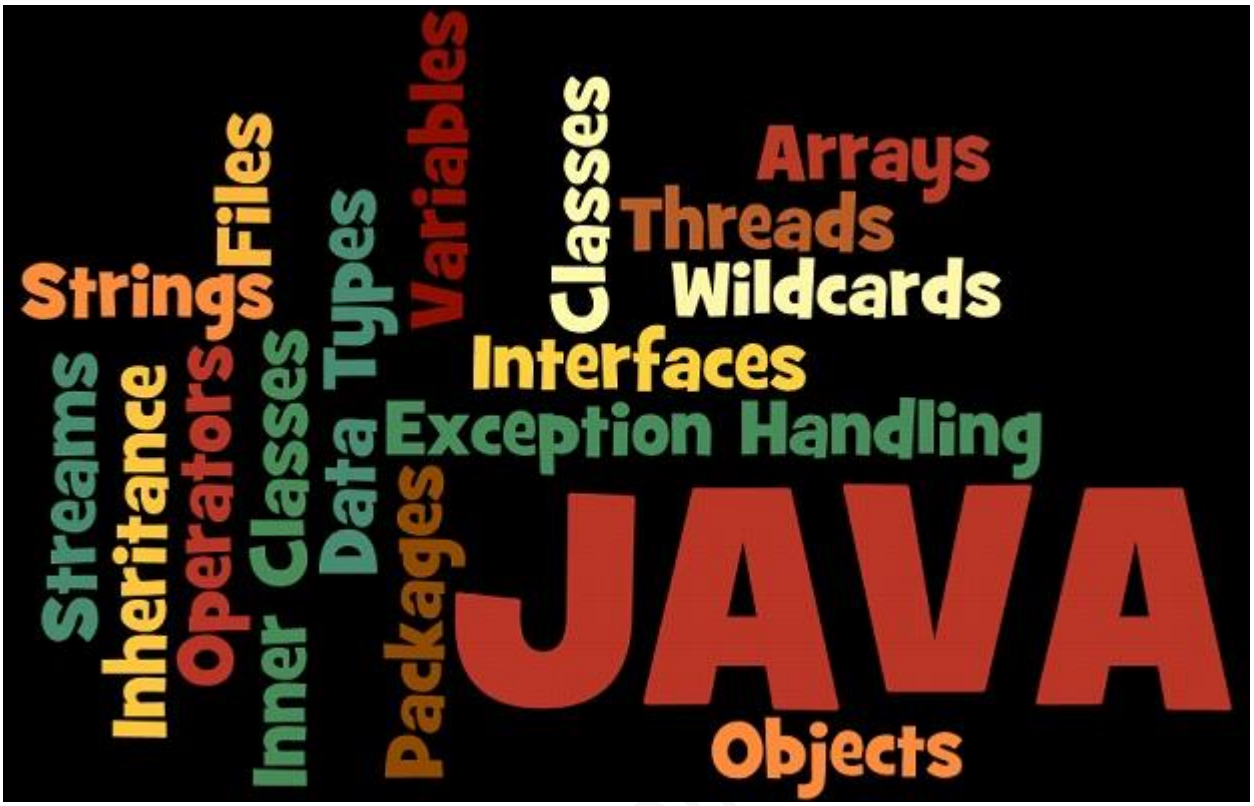
Bug Free: *It is far easier to turn out bug-free code using Java than using C++.*

1. Eliminated manual memory allocation and deallocation. Memory in Java is automatically garbage collected. You *never* have to worry about memory corruption.
2. Eliminated multiple inheritance, replacing it with a new notion of *interface (*Interfaces give you most of what you want from multiple inheritance, without the complexity that comes with managing multiple inheritance hierarchies.)

Where it is used?

According to Sun, 3 billion devices run java. There are many devices where java is currently used. Some of them are as follows:

1. Desktop Applications such as acrobat reader, media player, antivirus etc.
2. Web Applications such as irctc.co.in, javatpoint.com etc.
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System
6. Smart Card
7. Robotics
8. Games etc.



Java SE stands for Java standard edition and is normally for developing desktop applications, forms the core/base API.(Banking software)

Java EE stands for Java enterprise edition for applications which run on servers, for example web sites. (Twitter)

Java ME stands for Java micro edition for applications which run on resource constrained devices (small scale devices) like cell phones, for example mobile games (Nokia), micro-oven.

Core Java

Core Java/ Java SE

Core Java Course Outline

Syntax & Grammar

- Lexical Structure
- Data Types
- Literal Constants
- Variables and Arrays
- Operators and Expressions
- Conditions,Statements,Blocks

Control Flow

- Conditional Statements
- Loops and Switches
- Continue, Break and Return
- Command Line Arguments

Encapsulation and Classes

- Encapsulation
- Class Specifications
- Properties and Methods
- Types of Methods
- Creation and Destruction
- Accessing Objects
- Inner Classes

Inheritance & Polymorphism

- Inheritance
- Inheritance in Java
- Abstract Classes
- Interfaces
- Polymorphism:Dynamic Binding
- Arrays of Objects
- Casting Objects

Class Libraries

Swing GUI Widgets

- Containers, Windows, Panes
- JFrame and JPanel
- Dialog Boxes
- Labels, Icons and Buttons
- Basic Event Listeners
- Bounded-Range Components

Intermediate Swing

- Checkboxes
- Radio Buttons
- Lists,ComboBoxes,Spinners
- Text and Password Fields
- TextAreas
- 3rd Party Widgets

Advanced Swing

- Advanced Event Listeners
- Model-View-Controller
- Menu Bars & Tool Bars
- Panes: Scrolled and Layered
- Panes: Split and Tabbed
- Tables & Hierarchical Trees

Designing GUIs

- Colors and Fonts
- Borders and Separators
- Component Sizing
- Look And Feel
- Layout Managers
- Layered Design

GUI Views and Studies

Advance Java

Advance Java/ Java EE

Advance Java Course Outline

JSP Actions

- Introduction to JSP Pages
- Basic Elements
- Action Elements
- Implicit JSP objects
- Scriptlets
- Error Handling
- Data Control
- User Recognition
- Creating Java Beans
- Custon JSP Actions
- MySql Database
- Connection JSP to MySQL
- A JSP Online shop(Project Work)

Servlets:

- Introduction to Web Server
- Methods of Servlet
- Life Cycle of Servlet
- Feautre of Servlet 2.5
- Server Side Programming
- Installing and Configuring Servlets

JDBC:

- What is JDBC?
- Product Component of JDBC
- Understanding JDBC Architecture
- JDBC Driver and its types
- Features of JDBC 3.0
- Accessing Database using Java and JDBC
- Enhanced SQL Exception Handling
- Relational Dtabase Concepts
- Understanding Common SQL Statements
- Important JDBC Concepts(Transaction, Logging, Concurrency Concepts) RMI (Remote Method Invocation)
- Introduction to RMI
- Java RMI
- Create an RMI System
- Accelerate your RMI Programming
- Get Smart with proxies and RMI EJB (Enterprise Java Beans) 3.0
- What is EJB Container
- Enterprise Beans
- Features of EJB 3.0
- Annotations
- Session Beans
- Message Driven Beans
- Introduction to POJO(Plain Old Java Project)
- Java Persistence API
- EJB Services

Spring Framework

- Downloading and Installing Spring
- Spring Injection Example
- Spring MVC
- Spring Persistence
- Spring Web Service and Remoting
- Spring AOP and AspectJ
- Hibernate Framework
- Introduction to Hibernate 3.0
- Understanding Hibernate O/R Mapping
- Hibernate Query Language (HQL)
- Hibernate Criteria Query Example
- Hibernate Mapping
- Hibernate Projections Tutorials and Examples
- **OR**

Hibernate Framework

- Introduction
- Who Uses It?
- Installing
- Example Database
- Model Classes
- Configuration
- Mapped Statements
- Inserting Data
- Queries
- Logging
- Classpath

LOG4J

- OverView
- Installation
- Architecture
- Configuration
- Sample Program
- Logging Methods
- Logging Level
- Log Formatting

Maven:

- Introduction
- Ant Installation
- Ant Basics
- Flow of Control

Types of Java Applications

There are mainly 4 type of applications that can be created using java programming:

1) Standalone Application

It is also known as desktop application or window-based application. An application that we need to install on every machine such as media player, antivirus etc. AWT and Swing are used in java for creating standalone applications.

2) Web Application

An application that runs on the server side and creates dynamic page, is called web application. Currently, servlet, jsp, struts, jsf etc. technologies are used for creating web applications in java.

3) Enterprise Application

An application that is distributed in nature, such as banking applications etc. It has the advantage of high level security, load balancing and clustering. In java, EJB is used for creating enterprise applications.

4) Mobile Application

An application that is created for mobile devices. Currently Android and Java ME are used for creating mobile applications.

History of Java

1. Brief history of Java
2. Java Version History

Java history is interesting to know. The history of java starts from Green Team. Java team members (also known as **Green Team**), initiated a revolutionary task to develop a language for digital devices such as set-top boxes, televisions etc.

For the green team members, it was an advance concept at that time. But, it was suited for internet programming. Later, Java technology as incorporated by Netscape.

James Gosling

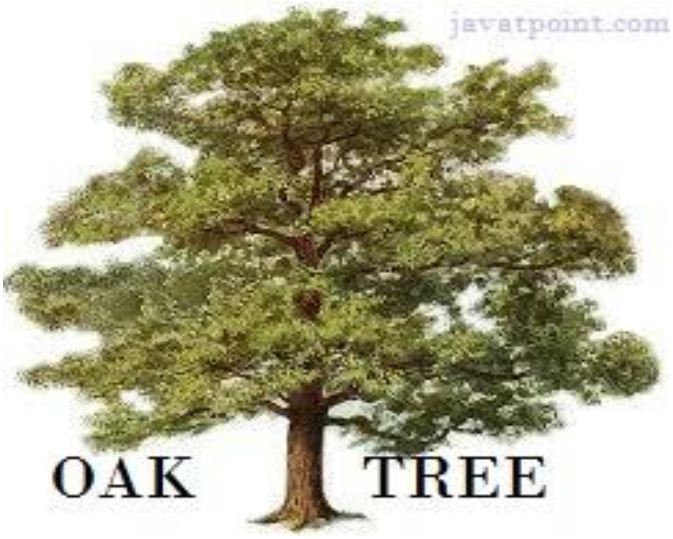
Currently, Java is used in internet programming, mobile devices, games, e-business solutions etc. There are given the major points that describes the history of java.

1) **James Gosling**, **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.

2) Originally designed for small, embedded systems in electronic appliances like set-top boxes.

3) Firstly, it was called **"Greentalk"** by James Gosling and file extension was .gt.

4) After that, it was called **Oak** and was developed as a part of the Green project.



Why Oak name for java language?

5) **Why Oak?** Oak is a symbol of strength and choosen as a national tree of many countries like U.S.A., France, Germany, Romania etc.
6) In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.

Why Java name for java language?

7) **Why they choosed java name for java language?** The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA" etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell and fun to say.

According to James Gosling "Java was one of the top choices along with **Silk**". Since java was so unique, most of the team members preferred java.

8) Java is an island of Indonesia where first coffee was produced (called java coffee).

9) Notice that Java is just a name not an acronym.

10) Originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.

11) In 1995, Time magazine called **Java one of the Ten Best Products of 1995**.

12) JDK 1.0 released in(January 23, 1996).

Java Version History

There are many java versions that has been released. Current stable release of Java is Java SE 8.

1. JDK Alpha and Beta (1995)
2. JDK 1.0 (23rd Jan, 1996)
3. JDK 1.1 (19th Feb, 1997)
4. J2SE 1.2 (8th Dec, 1998)
5. J2SE 1.3 (8th May, 2000)
6. J2SE 1.4 (6th Feb, 2002)
7. J2SE 5.0 (30th Sep, 2004)
8. Java SE 6 (11th Dec, 2006)
9. Java SE 7 (28th July, 2011)
10. Java SE 8 (18th March, 2014)

Features of Java

1. Features of Java
    1. Simple
    2. Object-Oriented
    3. Platform Independent
    4. secured
    5. Robust
    6. Architecture Neutral
    7. Portable
    8. High Performance
    9. Distributed
    10. Multi-threaded

There is given many features of java. They are also known as java buzzwords. The Java Features given below are simple and easy to understand.

1. Simple
2. Object-Oriented
3. Platform independent
4. Secured
5. Robust

6. Architecture neutral
7. Portable
8. Dynamic
9. Interpreted
10. High Performance
11. Multithreaded
12. Distributed

Simple

According to Sun, Java language is simple because:

Syntax is based on C++ (so easier for programmers to learn it after C++).

removed many confusing and/or rarely-used features e.g., explicit pointers, operator overloading etc.

No need to remove unreferenced objects because there is Automatic Garbage Collection in java.

Object-oriented

Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behaviour.

Object-oriented programming (OOPs) is a methodology that simplify software development and maintenance by providing some rules.

Basic concepts of OOPs are:

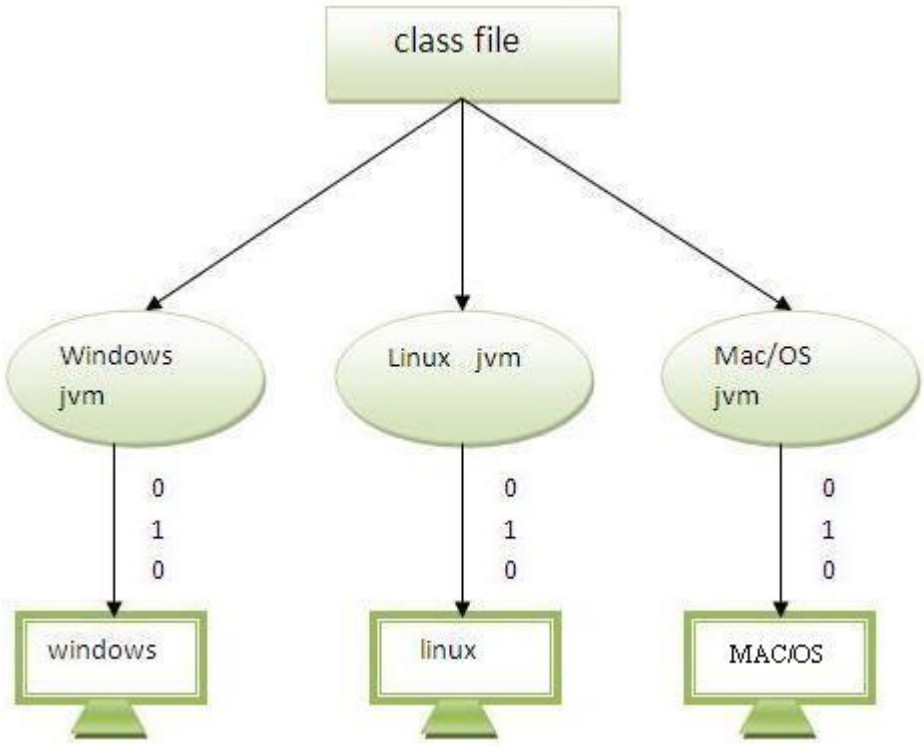
1. Object
2. Class
3. Inheritance

4. Polymorphism
5. Abstraction
6. Encapsulation

Platform Independent

A platform is the hardware or software environment in which a program runs. There are two types of platforms software-based and hardware-based. Java provides software-based platform. The Java platform differs from most other platforms in the sense that it's a software-based platform that runs on top of other hardware-based platforms.It has two components:

1. Runtime Environment
2. API(Application Programming Interface)



Java code can be run on multiple platforms e.g.Windows,Linux,Sun Solaris,Mac/OS etc.

Java code is compiled by the compiler and converted into bytecode.This bytecode is a platform independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere (WORA).

Secured

Java is secured because:

- No explicit pointer
- Programs run inside virtual machine sandbox.

- **Classloader-** adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier-** checks the code fragments for illegal code that can violate access right to objects.
- **Security Manager-** determines what resources a class can access such as reading and writing to the local disk.

These security are provided by java language. Some security can also be provided by application developer through SSL, JAAS, cryptography etc.

Robust

Robust simply means strong. Java uses strong memory management. There are lack of pointers that avoids security problem. There is automatic garbage collection in java. There is exception handling and type checking mechanism in java. All these points makes java robust.

Architecture-neutral

There is no implementation dependent features e.g. size of primitive types is set.

Portable

We may carry the java bytecode to any platform.

High-performance

Java is faster than traditional interpretation since byte code is "close" to native code still somewhat slower than a compiled language (e.g., C++)

Distributed

We can create distributed applications in java. RMI and EJB are used for creating distributed applications. We may access files by calling the methods from any machine on the internet.

Multi-threaded

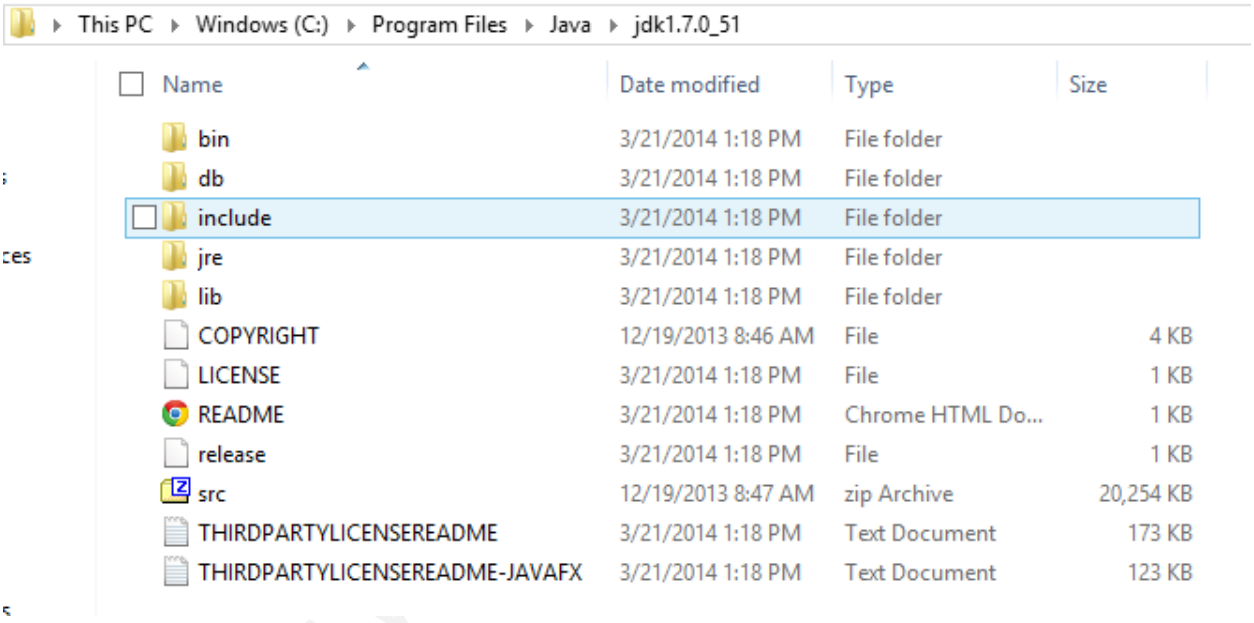
A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it shares the same memory. Threads are important for multi-media, Web applications etc.

Chapter 2

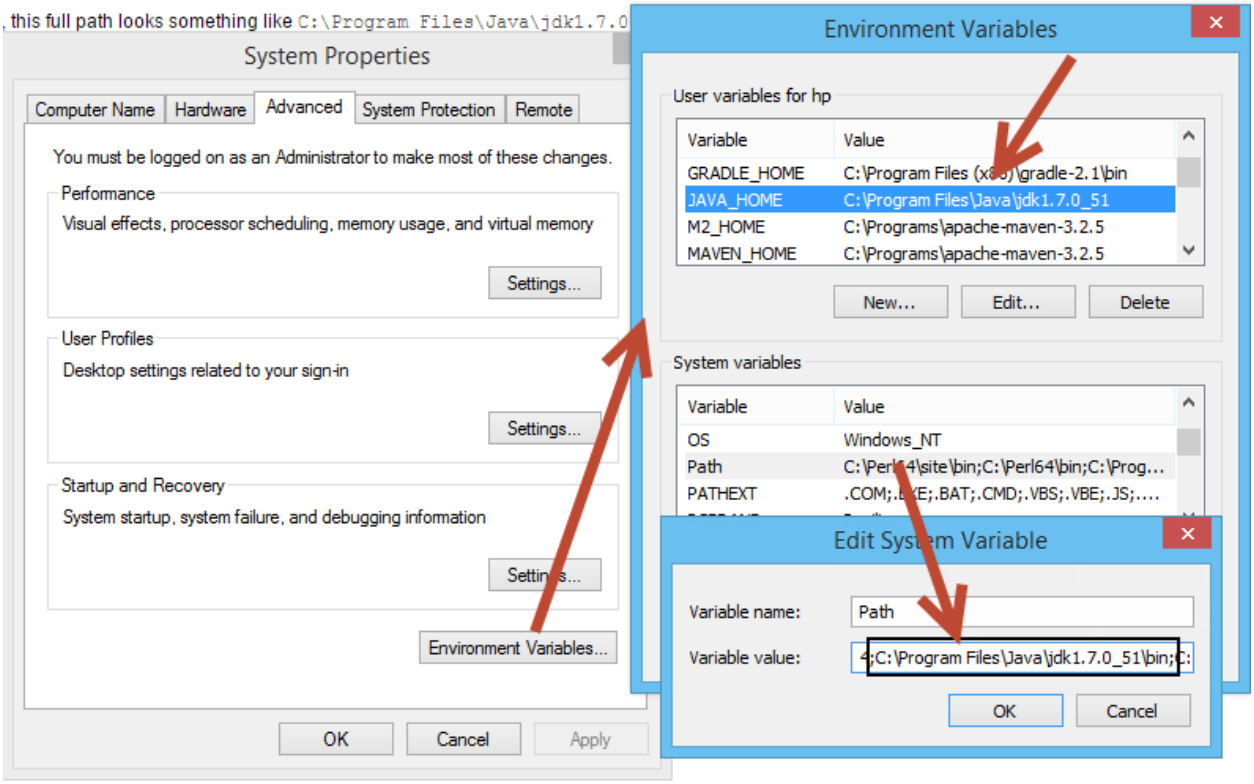
Set Up and Run Java Program

➢ Set up/Install Java

1. Make sure the JDK is installed and that the *jdk*/bin directory is on the executable path.

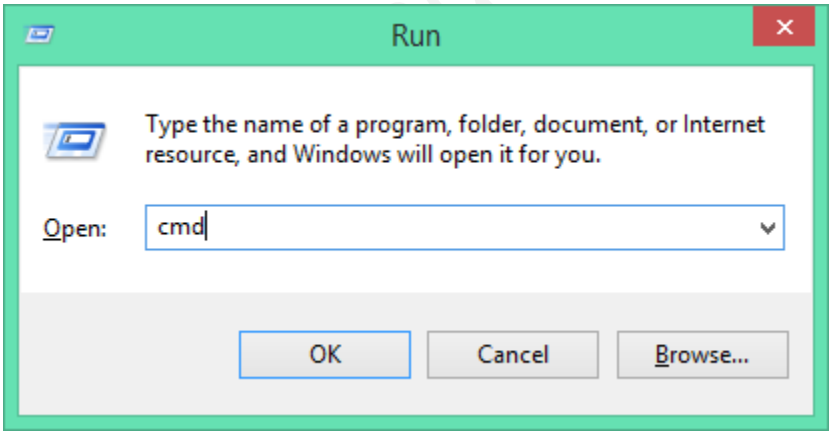


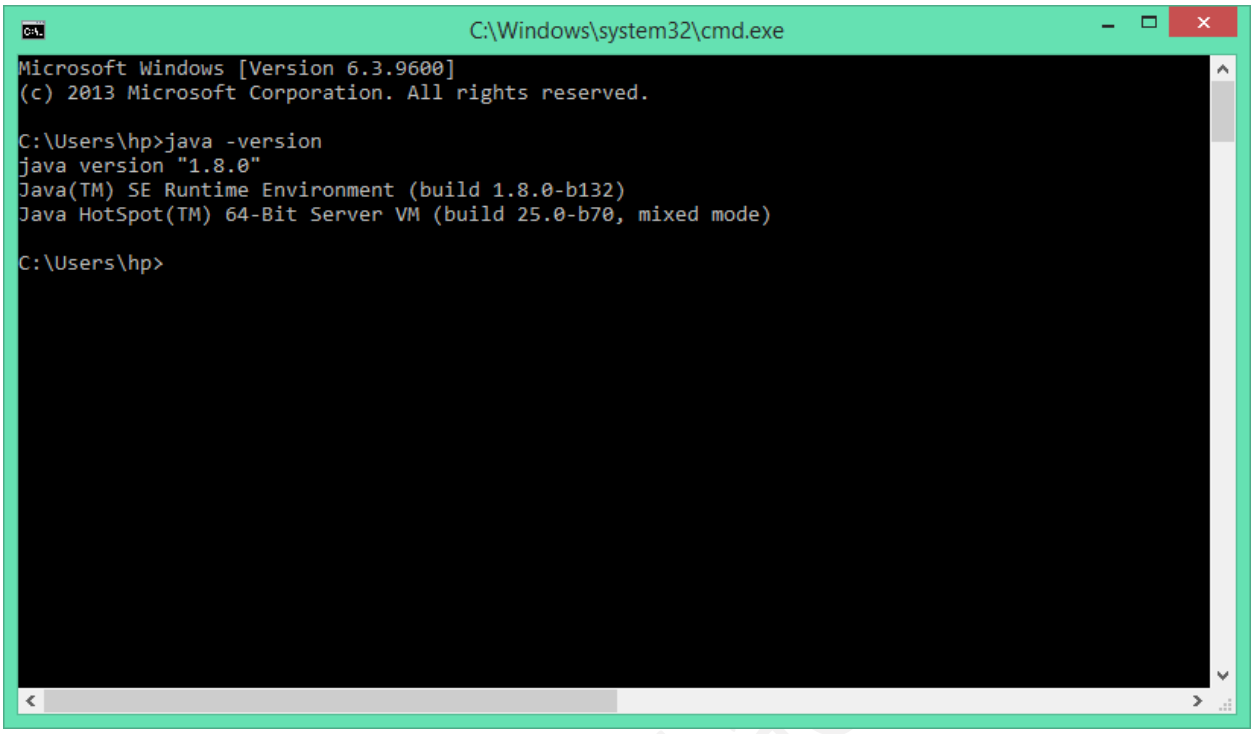
2.Updating the PATH Environment Variable:



3.Verify Java Installed Correctly

Windows + R = Open Run





Simple Program of Java

In this page, we will learn how to write the simple program of java. We can write a simple hello java program easily after installing the JDK.

To create a simple java program, you need to create a class that contains main method. Let's understand the requirement first.

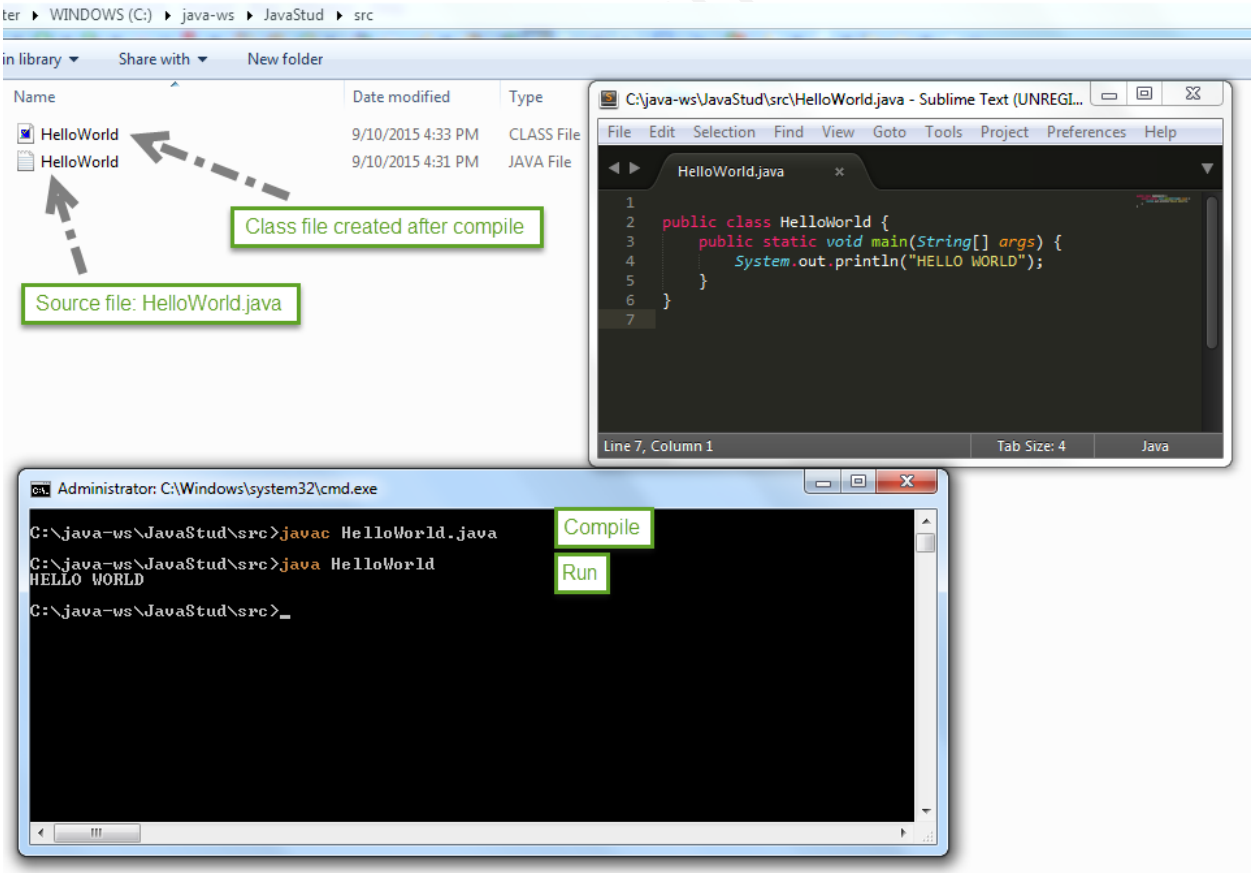
Requirement for Hello Java Example

For executing any java program, you need to

- Install the JDK if you don't have installed it, download the JDK and install it.
- Set path of the jdk/bin directory.
- create the java program
- compile and run the java program

Creating hello java example

Run HelloWorld Program from console window(terminal):



To compile: javac HelloWorld.java

To execute: java HelloWorld

Output: HELLO WORLD

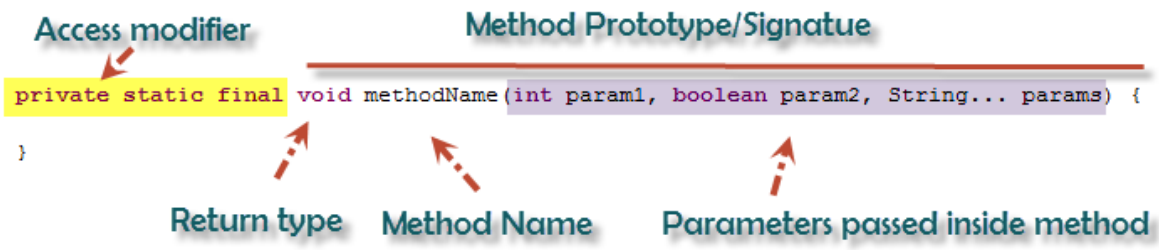
Understanding first java program

Let's see what is the meaning of class, public, static, void, main, String[], System.out.println().

- **class** keyword is used to declare a class in java.
- **public** keyword is an access modifier which represents visibility, it means it is visible to all.
- **static** is a keyword, if we declare any method as static, it is known as static method. The core advantage of static method is that there is no need to create object to invoke the static method. The main method is executed by the JVM, so it doesn't require to create object to invoke the main method. So it saves memory.
- **void** is the return type of the method, it means it doesn't return any value.
- **main** represents startup of the program.
- **String[] args** is used for command line argument. We will learn it later.
- **System.out.println()** is used print statement. We will learn about the internal working of System.out.println statement later.

How many ways can we write a java program

There are many ways to write a java program. The modifications that can be done in a java program are given below:



1) By changing sequence of the modifiers, method prototype is not changed.

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

2) Subscript notation in java array can be used after type, before variable or after variable.

Let's see the different codes to write the main method.

1. public static void main(String[] args)
2. public static void main(String []args)
3. public static void main(String args[])

3) You can provide var-args support to main method by passing 3 ellipses (dots)

We will learn about var-args later in Java New Features chapter.

1. public static void main(String... args)

4) Having semicolon at the end of class in java is optional.

Let's see the simple code.

```
1. class A{
2. static public void main(String... args){
3. System.out.println("hello world!");
4. }
5. };
```

Valid java main method signature

1. public static void main(String[] args)
2. public static void main(String []args)
3. public static void main(String args[])
4. public static void main(String... args)

5. static public void main(String[] args)
6. public static final void main(String[] args)
7. final public static void main(String[] args)

Invalid java main method signature

1. public void main(String[] args): (signature not wrong but you can't run non static main)
2. static void main(String[] args): (signature not wrong but you can't run private main)
3. public void static main(String[] args) : Method signature modified
4. abstract public static void main(String[] args) : (abstract method can't be static because abstract method have no body. It's not good idea to call method without body)

Internal Details of Hello Java Program

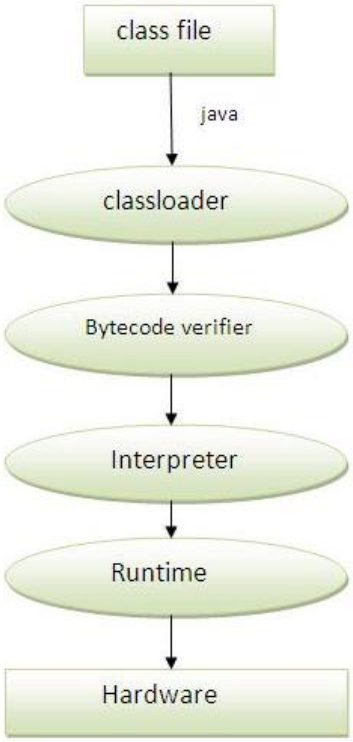
Here, we are going to learn, what happens while compiling and running the java program. Moreover, we will see some question based on the first program.

What happens at compile time?

At compile time, java file is compiled by Java Compiler (It does not interact with OS) and converts the java code into bytecode.

What happens at runtime?

At runtime, following steps are performed:



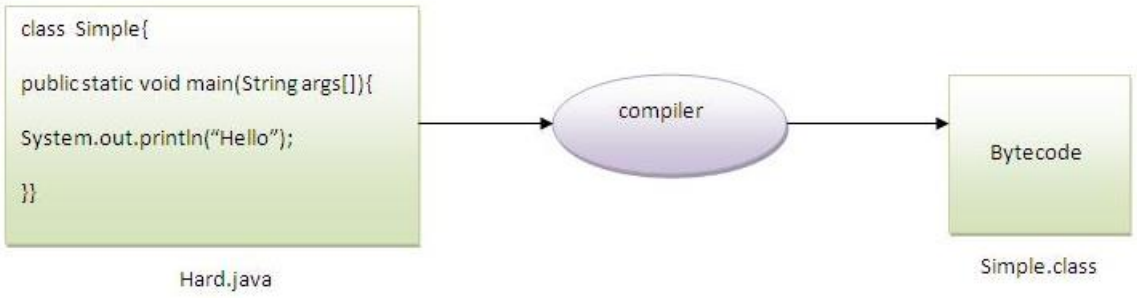
Classloader: is the subsystem of JVM that is used to load class files.

Bytecode Verifier: checks the code fragments for illegal code that can violate access right to objects.

Interpreter: read bytecode stream then execute the instructions.

Q) Can you save a java source file by other name than the class name?

Yes, if the class is not public. It is explained in the figure given below:



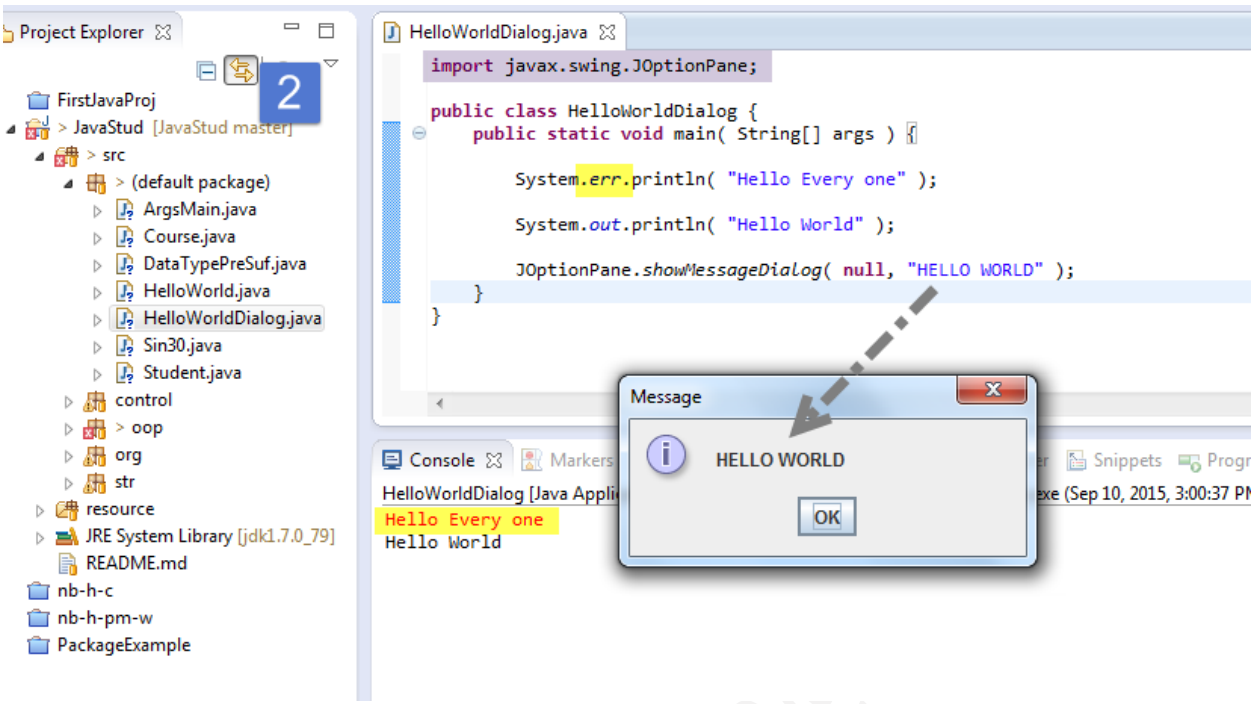
To compile: javac Hard.java

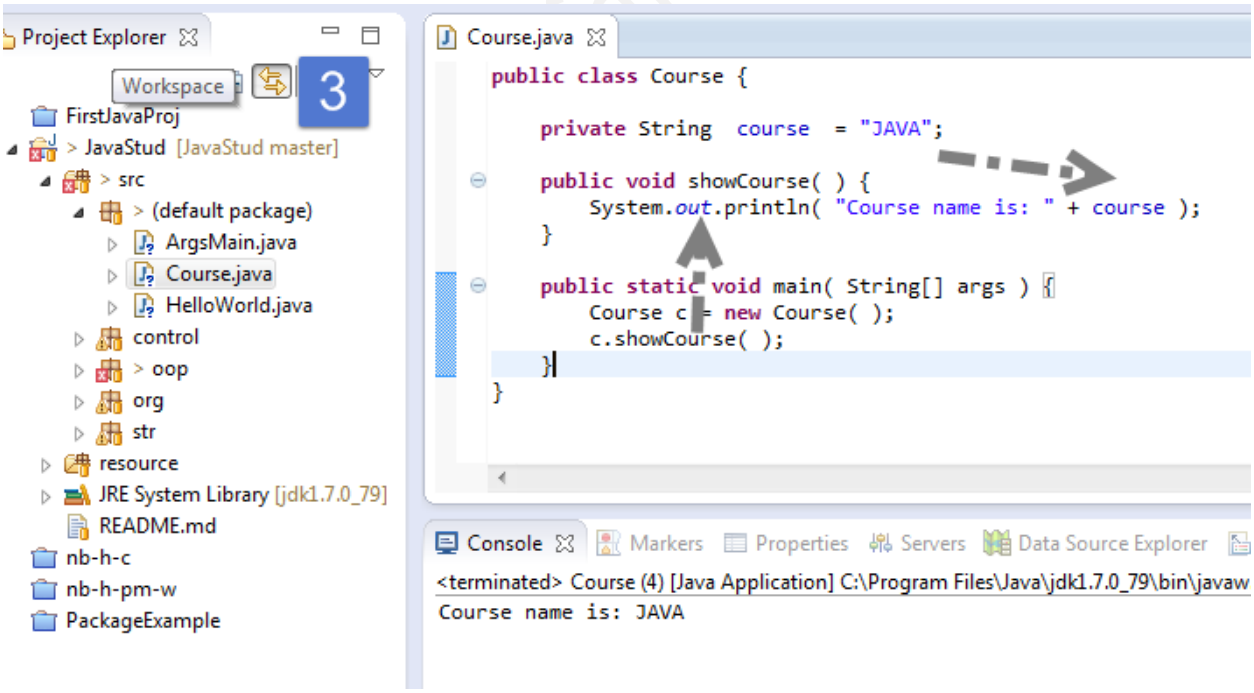
To execute: java Simple

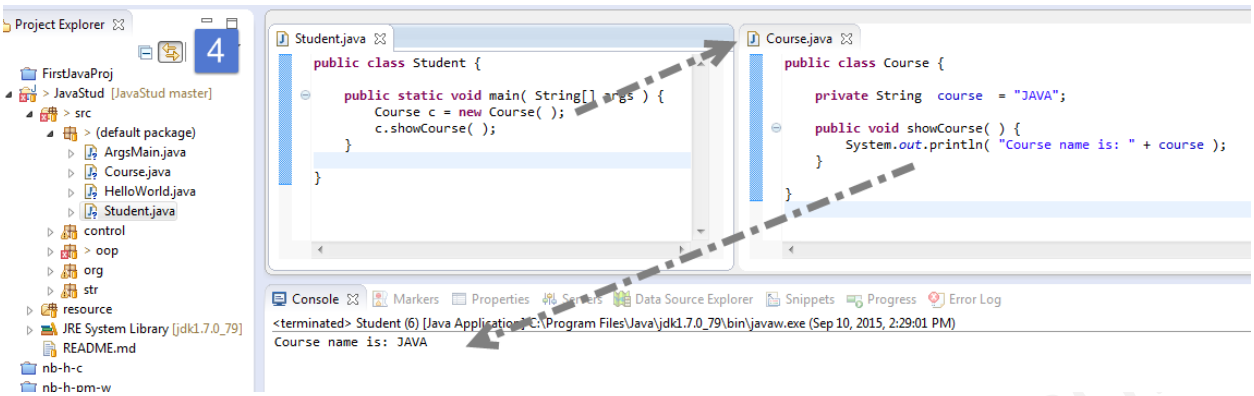
Q. Create Object of Java Class.

```
public class Course {

}

Course course = new Course();
```

Class Object new operator create object

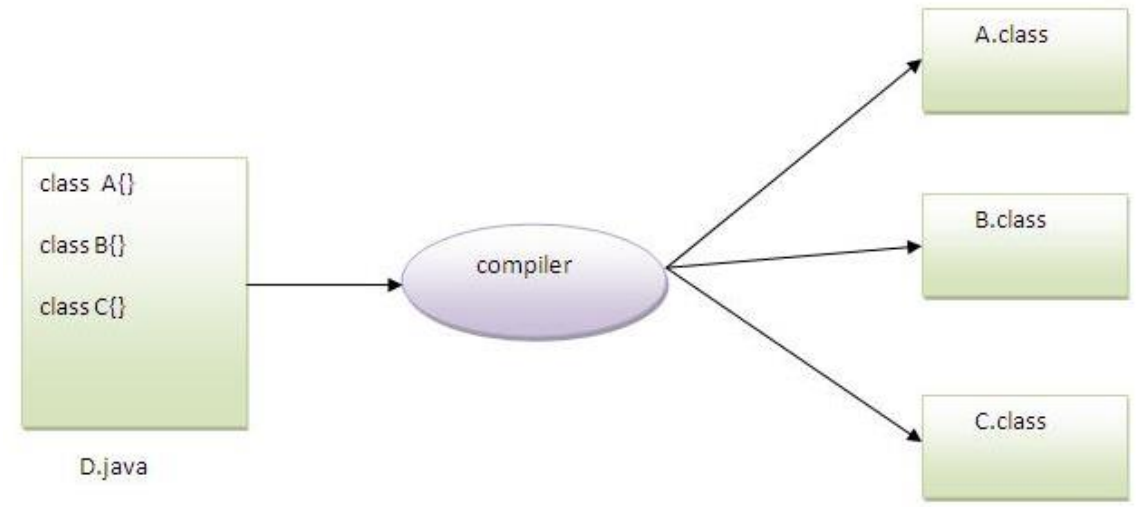

Project Explorer
2
FirstJavaProj
JavaStud [JavaStud master]
src
(default package)
ArgsMain.java
Course.java
DataTypePreSuf.java
HelloWorld.java
HelloWorldDialog.java
Sin30.java
Student.java
control
oop
org
str
resource
JRE System Library [jdk1.7.0_79]
README.md
nb-h-c
nb-h-pm-w
PackageExample
HelloWorldDialog.java
import javax.swing.JOptionPane;
public class HelloWorldDialog {
public static void main(String[] args) {
System.err.println("Hello Every one");
System.out.println("Hello World");
JOptionPane.showMessageDialog(null, "HELLO WORLD");
}
}
Message
HELLO WORLD
OK
Console
Markers
Snippets
HelloWorldDialog [Java Appli
Hello Every one
Hello World


Project Explorer
Workspace
3
FirstJavaProj
JavaStud [JavaStud master]
src
(default package)
ArgsMain.java
Course.java
HelloWorld.java
control
oop
org
str
resource
JRE System Library [jdk1.7.0_79]
README.md
nb-h-c
nb-h-pm-w
PackageExample
Course.java
public class Course {
private String course = "JAVA";
public void showCourse( ) {
System.out.println("Course name is: " + course);
}
public static void main(String[] args) {
Course c = new Course( );
c.showCourse( );
}
}
Console
Markers
Properties
Servers
Data Source Explorer
<terminated> Course (4) [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw
Course name is: JAVA



Q) Can you have multiple classes in a java source file?

Yes, like the figure given below illustrates:



Chapter 3
Programming Structures

Difference between JDK, JRE and JVM

JRE (Java Runtime Environment)

Java Runtime Environment contains JVM, class libraries, and other supporting files. It does not contain any development tools such as compiler, debugger, etc. Actually JVM runs the program, and it uses the class libraries, and other supporting files provided in JRE. If you want to run any java program, you need to have JRE installed in the system.

JDK (Java Development Kit)

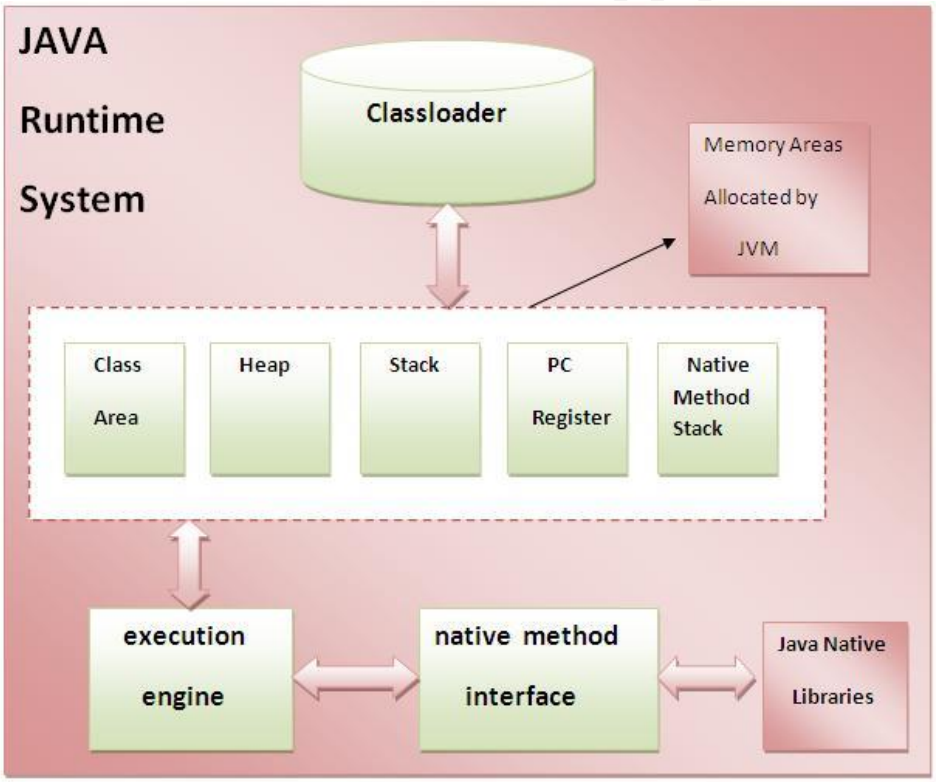
Java Development Kit (JDK) The JDK is a superset of the JRE, and contains everything that is in the JRE, plus tools such as the compilers and debuggers necessary for developing applets and applications.

JVM (Java Virtual Machine)

As we all aware when we compile a Java file, output is not an ‘exe’ but it’s a ‘.class’ file. ‘.class’ file consists of Java byte codes which are understandable by JVM. Java Virtual Machine interprets the byte code into the machine code depending upon the underlying operating system and hardware combination. It is responsible for all the things like garbage collection, array bounds checking, etc… JVM is platform dependent.

The ***JVM*** is called “virtual” because it provides a machine interface that does not depend on the underlying operating system and machine hardware architecture. This independence from hardware and operating system is a cornerstone of the write-once run-anywhere value of Java programs.
There are different JVM implementations are there. These may differ in things like performance, reliability, speed, etc. These implementations will differ in those areas where Java specification doesn’t mention how to implement the features, like how the garbage collection process works is JVM dependent, Java spec doesn’t define any specific way to do this.

Internal Architecture of JVM



1) Classloader:

Classloader is a subsystem of JVM that is used to load class files.

2) Class(Method) Area:

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

3) Heap:

It is the runtime data area in which objects are allocated.

4) Stack:

Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return.

Each thread has a private JVM stack, created at the same time as thread.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

5) Program Counter Register:

PC (program counter) register. It contains the address of the Java virtual machine instruction currently being executed.

6) Native Method Stack:

It contains all the native methods used in the application.

7) Execution Engine:

It contains:

1) A virtual processor

2) Interpreter: Read bytecode stream then execute the instructions.

3) Just-In-Time(JIT) compiler: It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here the term ?compiler? refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

Java Lexical Structures and Issues

Java programs are a collection of whitespace, identifiers, literals, comments, operators, separators, and keywords.

Whitespace

Java is a free-form language. This means that you do not need to follow any special indentation rules. In Java, whitespace is a space, tab, or newline.

Identifiers:

Identifiers are the names of variables, methods, classes, packages and interfaces. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters.

AvgTemp	count	a4	$test	this_is_okey

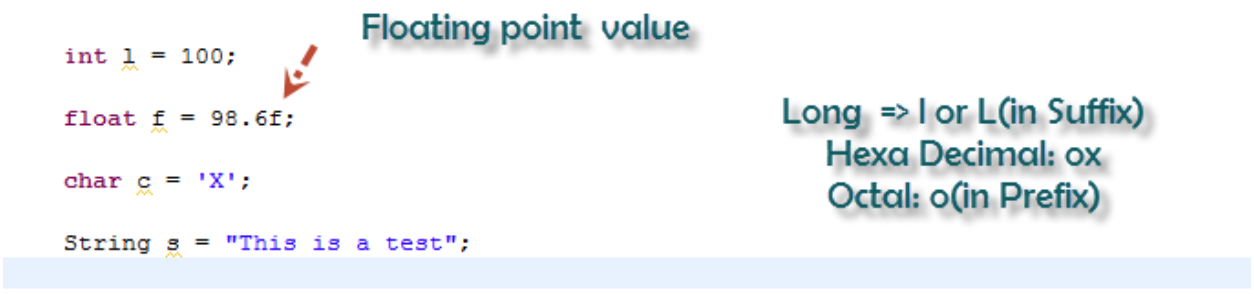
Invalid Identifiers:

2Count	high-temp	Not/ok
digit: 2	hyphen: -	slash: /

Literals

A constant value in Java is created by using a literal representation of it. For example, here are some literals:

100	98.6	'X'	"This is a String"



Standard Java Naming Conventions

The below list outlines the standard Java naming conventions for each identifier type:

- **Packages:** Names should be in lowercase. With small projects that only have a few packages it's okay to just give them simple (but meaningful!) names:

```
package broadway
package mycalculator
```

  In software companies and large projects where the packages might be imported into other classes, the names will normally be subdivided. Typically this will start with the company domain before being split into layers or features:

```
package com.mycompany.utilities
package org.xyzcompany.application.userinterface
```

- **Classes:** Names should be in CamelCase. Try to use nouns because a class is normally representing something in the real world:

```
class Customer
class Account
```

- **Interfaces:** Names should be in CamelCase. They tend to have a name that describes an operation that a class can do:

```
interface Comparable
interface Enumerable
```

  Note that some programmers like to distinguish interfaces by beginning the name with an "I":

```
interface IComparable
interface IEnumerable
```

- **Methods:** Names should be in mixed case. Use verbs to describe what the method does:

```
void calculateTax()
string getSurname()
```

- **Variables:** Names should be in mixed case. The names should represent what the value of the variable represents:

```
string firstName
int orderNumber
```

Only use very short names when the variables are short lived, such as in for loops:

```
for (int i=0; i<20;i++) {    //i only lives in here }
```

- **Constants:** Names should be in uppercase.

```
static final int DEFAULT_WIDTH
static final int MAX_HEIGHT
```

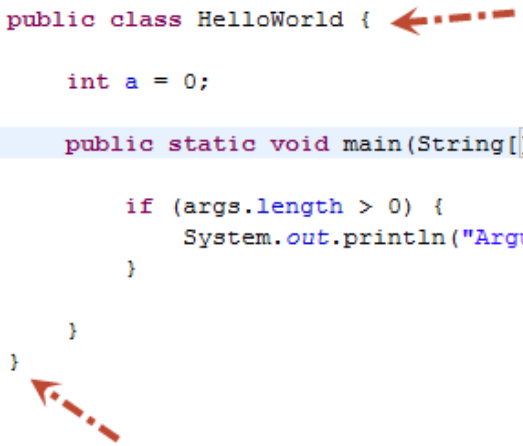
CamelCase: each new word begins with a capital letter. `Eg: JOptionPane`

Mixed case (also known as Lower CamelCase) is the same as CamelCase except the first letter of the name is in lowercase. Eg: showMessageDialog

```
public class MyClass {
  public String firstName, lastName;

  public String fullName() {
    String name =
      firstName + " " + lastName;
    return(name);
  }
}
```

Formatting Conventions .

You should **always** open a bracket at the end of a statement. You should **always** close the bracket at the beginning of a line.



```
public class HelloWorld {

    int a = 0;

    public static void main(String[] args) {

        if (args.length > 0) {
            System.out.println("Argument value is found!");
        }

    }
}
```

Java Comments .

Comments in Java, as in most programming languages, do not show up in the executable program. Thus, you can add as many comments as needed without fear of bloating the code. Java has three ways of marking comments.

Single Line Comment(//) : You use // for a comment that will run from the // to the end of the line.

```
//int number;

System.out.println("Hello world!"); //Comment at the end.
```

Multiple Line: /* */

```
/* Call showCourse method by creating object.
  1. Create Object */
Course course = new Course();
```

```
/*
 * Call showCourse method by creating object.
 * 1. Create Object
 */
Course course = new Course();

/* 2. Call Method showCourse(); */
course.showCourse();
```

Documentation Comment: /** */

```
/**
 * Class: Course
 * This class show Running Course!
 * @author YROJHA
 */
public class Course {

    public void showCourse() {
        System.out.println("Java! I am called by Object!");
    }

}
```

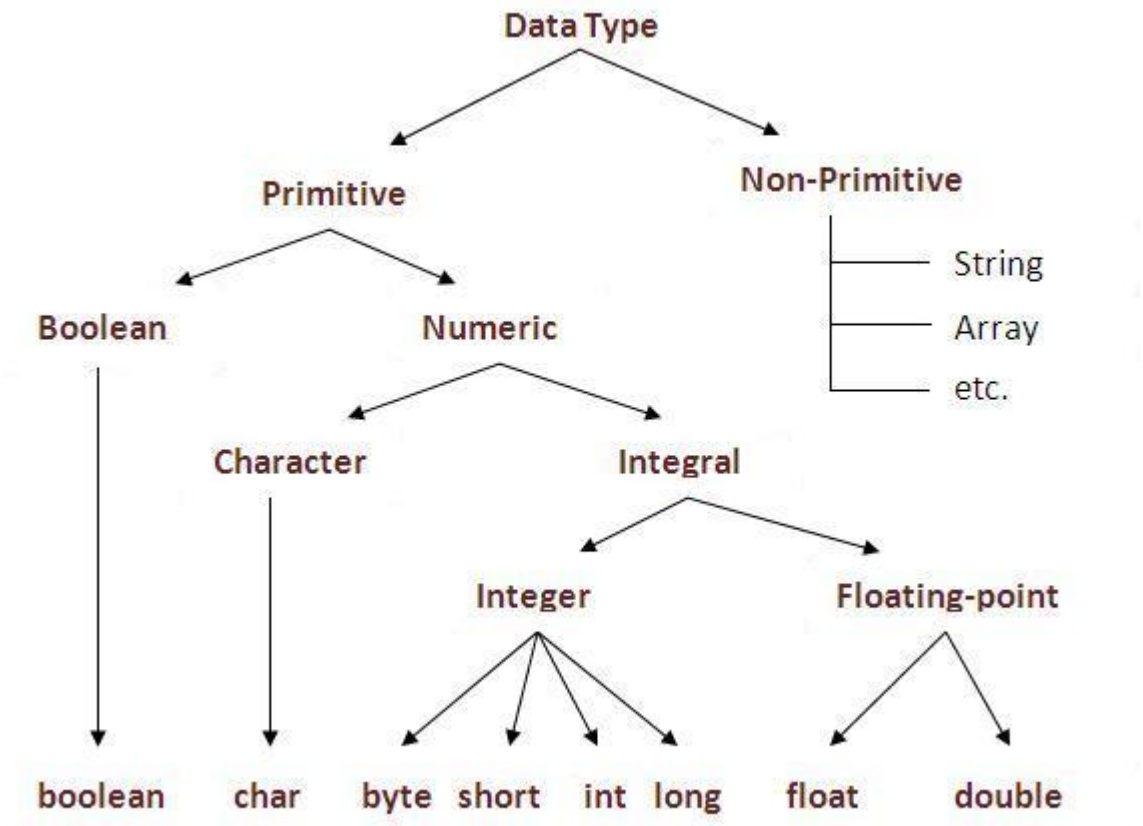
Variable and Datatype in Java

Data Types in Java

Java is a **strongly typed language**. This means that every variable must have a declared type. There are ***eight primitive types*** in Java. Four of them are integer types; two are floating-point number types; one is the character type char, used for code units in the Unicode encoding scheme.

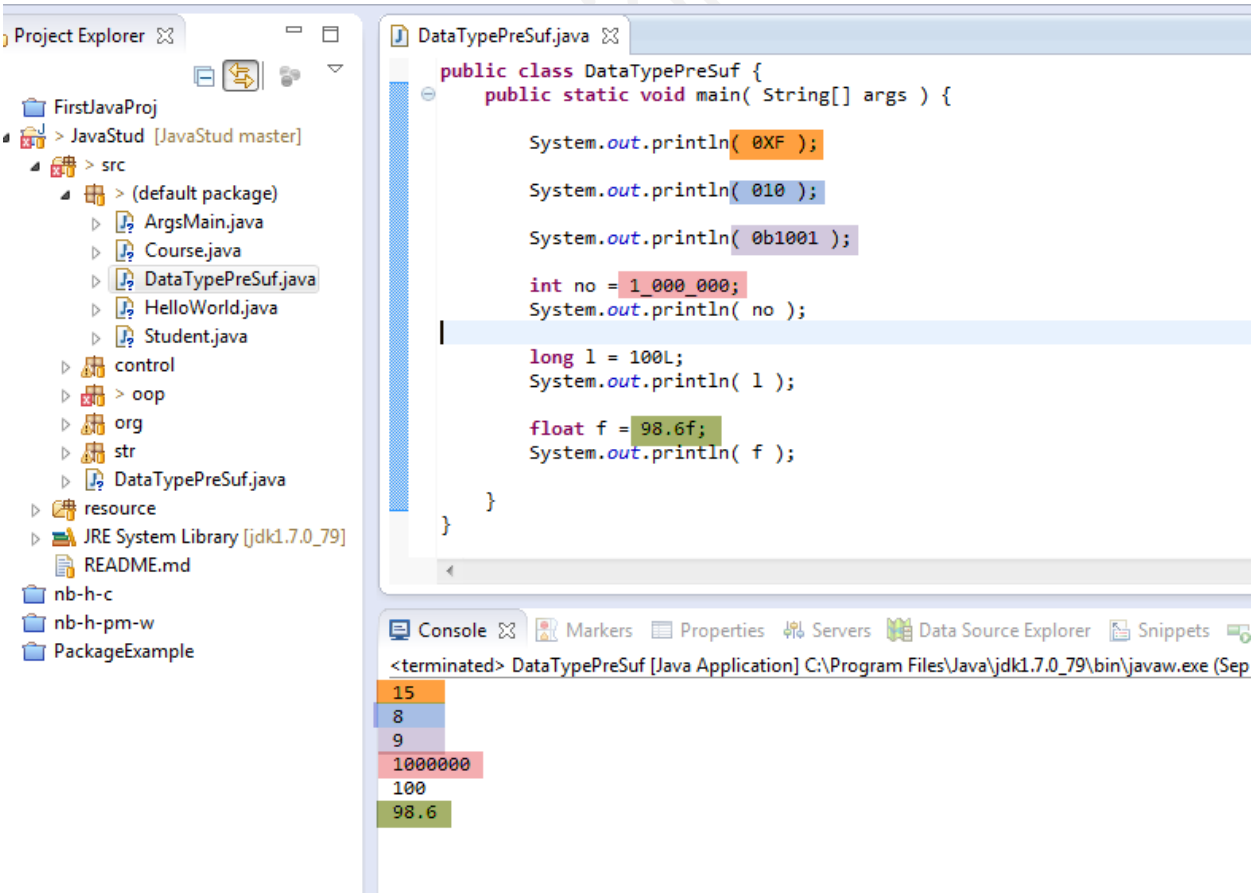
In java, there are two types of data types

- primitive data types
- non-primitive data types



Primitive Types					
Type Name	**Wrapper class**	**Value**	**Range**	**Size**	**Default Value**
`byte`	`java.lang.Byte`	integer	−128 through +127	8-bit (1-byte)	`0`
`short`	`java.lang.Short`	integer	−32,768 through +32,767	16-bit (2-byte)	`0`
`int`	`java.lang.Integer`	integer	−2,147,483,648 through +2,147,483,647	32-bit (4-byte)	`0`
`long`	`java.lang.Long`	integer	−9,223,372,036,854,775,808 through +9,223,372,036,854,775,807	64-bit (8-byte)	`0`
`float`	`java.lang.Float`	floating point number	±1.401298E−45 through ±3.402823E+38	32-bit (4-byte)	`0.0`
`double`	`java.lang.Double`	floating point number	±4.94065645841246E−324 through ±1.79769313486232E+308	64-bit (8-byte)	`0.0`
`boolean`	`java.lang.Boolean`	Boolean	`true` or `false`	8-bit (1-byte)	`false`
`char`	`java.lang.Character`	UTF-16 code unit (BMP character or a part of a surrogate pair)	`'\u0000'` through `'\uFFFF'`	16-bit (2-byte)	`'\u0000'`

Long integer numbers have a suffix L (for example, 4000000000L). Hexadecimal numbers have a prefix 0x (for example, 0xCAFE). Octal numbers have a prefix 0. For example, 010 is 8. Naturally, this can be confusing, so its recommend against the use of octal constants. Starting with Java 7, you can write numbers in binary, with a prefix 0b. For example, 0b1001 is 9. Also starting with Java 7, you can add underscores to number literals, such as 1_000_000 (or 0b1111_0100_0010_0100_0000) to denote one million. The underscores are for human eyes only. The Java compiler simply removes them.



Variable

Variable is name of reserved area allocated in memory. Variable is a name of memory location. There are three types of variables: local, instance and static. There are two types of datatypes in java, primitive and non-primitive.

```
int a, b, c; // declares three ints, a, b, and c.

int d = 3, e, f = 5; // declares three more ints, initializing d and f.

byte z = 22; // initializes z.

double pi = 3.14159; // declares an approximation of pi.

char x = 'x'; // the variable x has the value 'x'.
```

Local Variable

A variable that is declared inside the method is called local variable.

Instance Variable

A variable that is declared inside the class but outside the method is called instance variable . It is not declared as static.

Static variable

A variable that is declared as static is called static variable. It cannot be local.

```
Public class VariableTypeClass{
        int data=50;//instance variable

        static int m=100;//static variable

        void method(){
                int n=90;//local variable
        }
}
```

Exercise:

- Compute the number of miles that light will travel in a specified number of days:

Light.java

```
/**
 * Compute distance light travels using long variables.
 * @author YROJHA
 */
public class Light {
    public static void main(String[] args) {
        int lightspeed;
        long days;
        long seconds;
        long distance;

        // approximate speed of light in miles per second
        lightspeed = 186000;
        days = 1000; // specify number of days here

        seconds = days * 24 * 60 * 60; // convert to seconds
        distance = lightspeed * seconds; // compute distance

        System.out.print("In " + days);
        System.out.print(" days light will travel about ");
        System.out.println(distance + " miles.");
    }
}
```

Console | Javadoc | Problems | Declaration

```
<terminated> Light [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Apr 19, 2015, 1:35:14 PM)
In 1000 days light will travel about 16070400000000 miles.
```

- compute the area of a circle:

Area.java

```
/**
 * Compute the area of a circle.
 *
 * @author YROJHA
 */
public class Area {

    public static void main(String[] args) {

        // Define Variables
        double pi, r, area;

        r = 10.8; // radius of circle
        pi = 3.1416; // pi, approximately
        area = pi * r * r; // compute area

        System.out.println("Area of circle is " + area);
    }

}
```

Console | Javadoc | Problems | Declaration

```
<terminated> Area [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Apr 19, 2015, 1:44:33 PM)
Area of circle is 366.436224
```

Float: Variables of type float are useful when you need a fractional component, but don't require a large degree of precision. For example, float can be useful when representing dollars and cents.

Double: All transcendental math functions, such as sin( ), cos( ), and sqrt( ), return double values. When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, double is the best choice.

- Concatenation of char

CharDemo.java

```
/**
 * Demonstrate char data type.
 * @author YROJHA
 */
public class CharDemo {
    public static void main(String[] args) {

        char ch1, ch2;

        ch1 = 88; // code for X
        ch2 = 'Y';

        System.out.print("ch1 and ch2: ");
        System.out.println(ch1 + " " + ch2);
    }
}
```

Console | Javadoc | Problems | Declaration

```
<terminated> CharDemo [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Apr 19, 2015, 1:51:15 PM)
ch1 and ch2: X Y
```

- Boolean Test

BoolTest.java

```
/**
 * Demonstrate boolean values.
 * @author YROJHA
 */
public class BoolTest {

    public static void main(String[] args) {

        boolean b;

        b = false;
        System.out.println("b is " + b);

        b = true;
        System.out.println("b is " + b);

        // a boolean value can control the if statement
        if (b) {
            System.out.println("This is executed.");
        }

        b = false;
        if (b) {
            System.out.println("This is not executed.");
        }

        // outcome of a relational operator is a boolean value
        System.out.println("10 > 9 is " + (10 > 9));

    }

}
```

Console | Javadoc | Problems | Declaration

```
<terminated> BoolTest [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Apr 19, 2015, 1:55:14 PM)
b is false
b is true
This is executed.
10 > 9 is true
```

The Scope and Lifetime of Variables

Java allows variables to be declared within any block. A block is begun with an opening curly brace and ended by a closing curly brace. A block defines a scope. Thus, each time you start a new block, you are creating a new scope. A scope determines what objects are visible to other parts of your program also determines the lifetime of those objects.

Scope.java

```
// Demonstrate block scope.
public class Scope {
    public static void main(String[] args) {

        int x; // known to all code within main

        x = 10;
        if (x == 10) { // start new scope

            int y = 20; // known only to this block

            // x and y both known here.
            System.out.println("x and y: " + x + " " + y);

            x = y * 2;
        }

        // y = 100; /* Error! y not known here */

        // x is still known here.
        System.out.println("x is " + x);
    }
}
```

Console | Javadoc | Problems | Declaration

```
<terminated> Scope [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (Apr 19, 2015, 2:48:41 PM)
x and y: 10 20
x is 40
```

Here is another important point to remember: variables are created when their scope is entered, and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope. Therefore, variables declared within a method will not hold their values between calls to that method. Also, a variable declared within a block will lose its value when the block is left. Thus, the lifetime of a variable is confined to its scope.

Operators in java

Operator in java is a symbol that is used to perform operations. There are many types of operators in java such as unary operator, arithmetic operator, relational operator, shift operator, bitwise operator, ternary operator and assignment operator. Most of its operators can be divided into the following four groups: arithmetic, bitwise, relational, and logical.

Operators	Precedence
postfix	*expr*++ *expr*--
unary	++*expr* --*expr*
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

int m = 7;
int n = 7;
int a = 2 * ++m; // now a is **16**, m is 8
int b = 2 * n++; // now b is **14**, n is 8

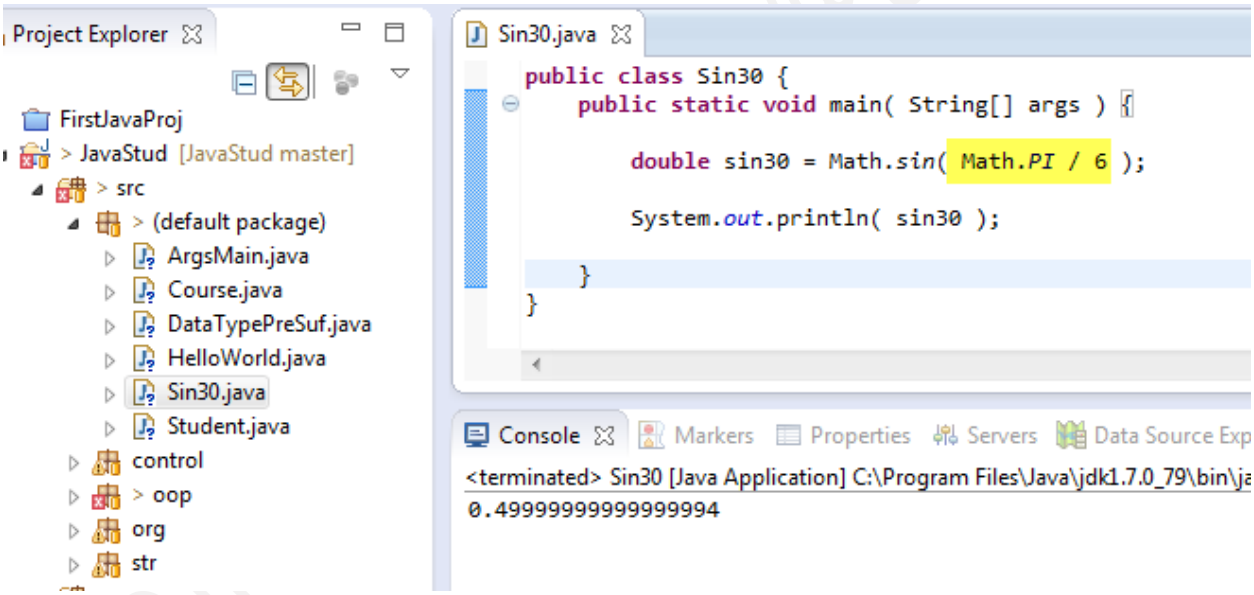
3 == 7 is **false**.
3 != 7 is **true**.
(marks > 40)? “PASS”:”FAIL”
If(null != patientContacts && patientContacts.size() > 0) { }

Mathematical Functions and Constants

The Math class contains an assortment of mathematical functions that you may occasionally need, depending on the kind of programming that you do.
To take the square root of a number, use the sqrt method:

```
double x = 4;
double y = Math.sqrt(x);
System.out.println(y); // prints 2.0
```

The Math class supplies the usual trigonometric functions:
Math.sin
Math.cos
Math.tan
Math.atan
Math.atan2



Type Conversion and Casting

It is often necessary to convert from one numeric type to another. If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an **int** value to a **long** variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no automatic conversion defined from **double** to **byte**. Fortunately, it is still possible to obtain a conversion between incompatible types. To do so, you must use a cast, which performs an explicit conversion between incompatible types.

Java's Automatic Conversions

When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:

- **The two types are compatible.**
- **The destination type is larger than the source type.**

Conversions between Numeric Types

```
int n = 123456789;

float f = n; //f is: 1.23456792E8
```

Casting Incompatible Types

To create a conversion between two incompatible types, you must use a cast. A *cast* is simply an explicit type conversion.
It has this general form: (*target-type*) *value*
For Example:

```
double x = 9.997;
int nx = ( int ) x; // 9
```

double x = 9.997;
int nx = (int) Math.round(x); //10

Automatic Type Promotion in Expressions

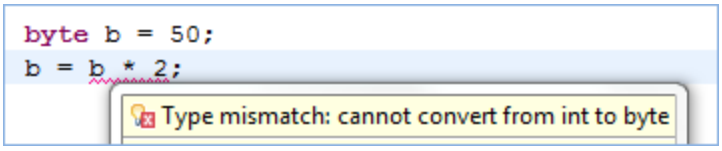
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;

The result of the intermediate term **a * b** easily exceeds the range of either of its **byte** operands. To handle this kind of problem, Java automatically promotes each **byte**, **short**, or **char** operand to **int** when evaluating an expression. This means that the sub expression **a*b** is performed using

integers—not bytes. Thus, 2,000, the result of the intermediate expression, **50 * 40**, is legal even though **a** and **b** are both specified as type **byte**.

As useful as the automatic promotions are, they can cause confusing compile-time errors. For example, this seemingly correct code causes a problem:

```
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
```



The code is attempting to store 50 * 2, a perfectly valid **byte** value, back into a **byte** variable. However, because the operands were automatically promoted to **int** when the expression was evaluated, the result has also been promoted to **int**. Thus, the result of the expression is now of type **int**, which cannot be assigned to a **byte** without the use of a cast. This is true even if, as in this particular case, the value being assigned would still fit in the target type.

In cases where you understand the consequences of overflow, you should use an explicit cast, such as

```
byte b = 50;
b = (byte) (b * 2);
```

which yields the correct value of 100.

The Type Promotion Rules:

First, all **byte**, **short**, and **char** values are **promoted to int**, as just described. Then, if **one operand is a long**, the **whol expression is promoted to long**. If one operand is a float, the entire expression is promoted to float. If any of the operands are double, the result is double.

```
int i = 5000;
float f = 5.67f;

float result = i * f;
```

Java Package and Import

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

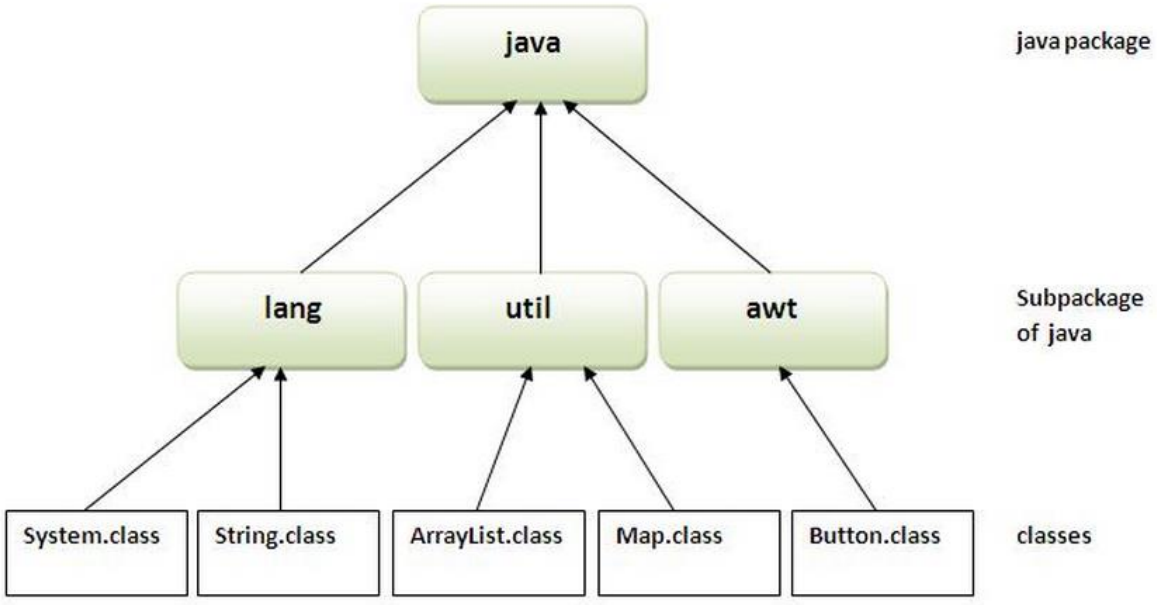
Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

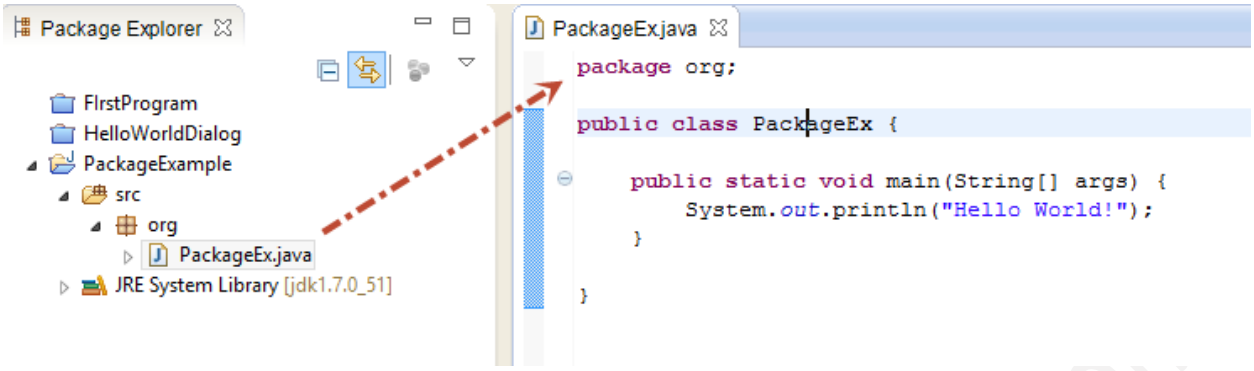
2) Java package provides access protection.

3) Java package removes naming collision.



Simple example of java package

The **package keyword** is used to create a package in java.



How to access package from another package?

There are three ways to access the package from outside the package.

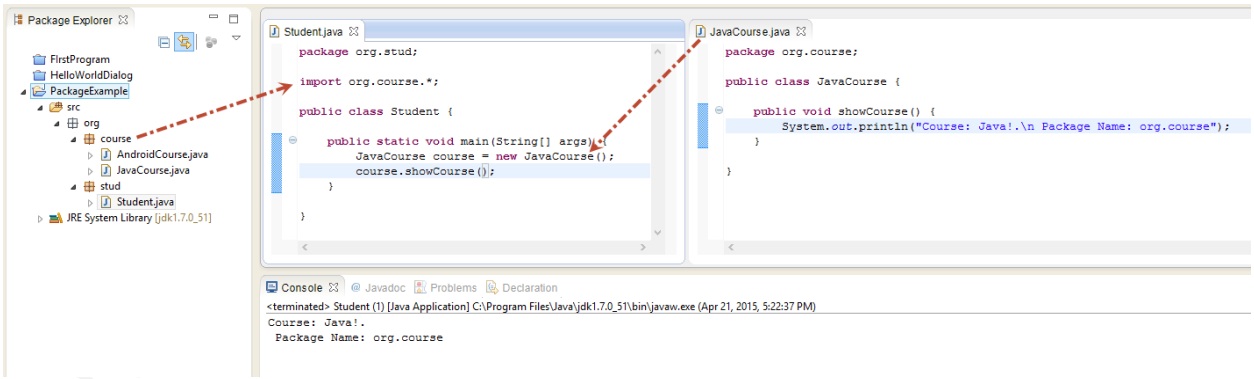
1. import package.*;
2. import package.classname;
3. fully qualified name.

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

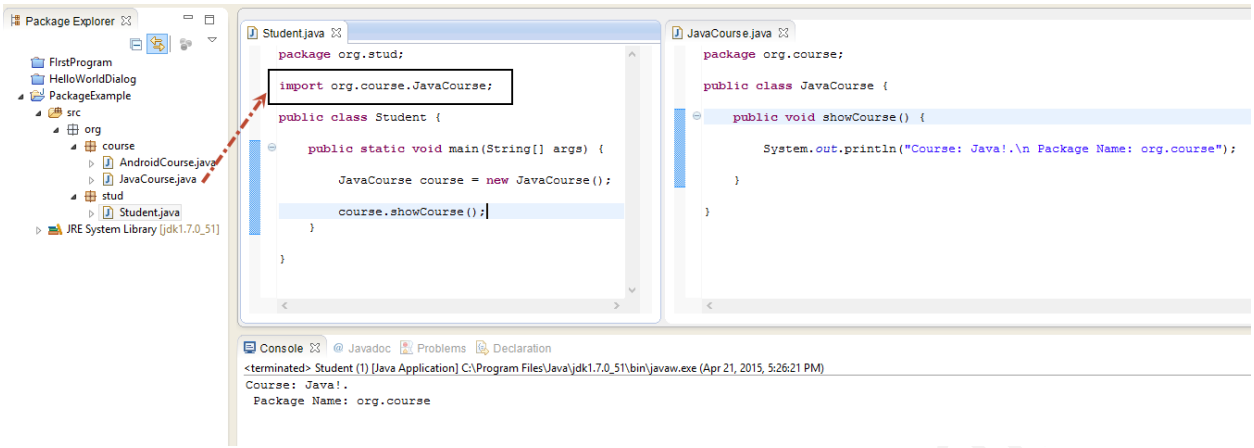
Example of package that import the packagename.*



2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

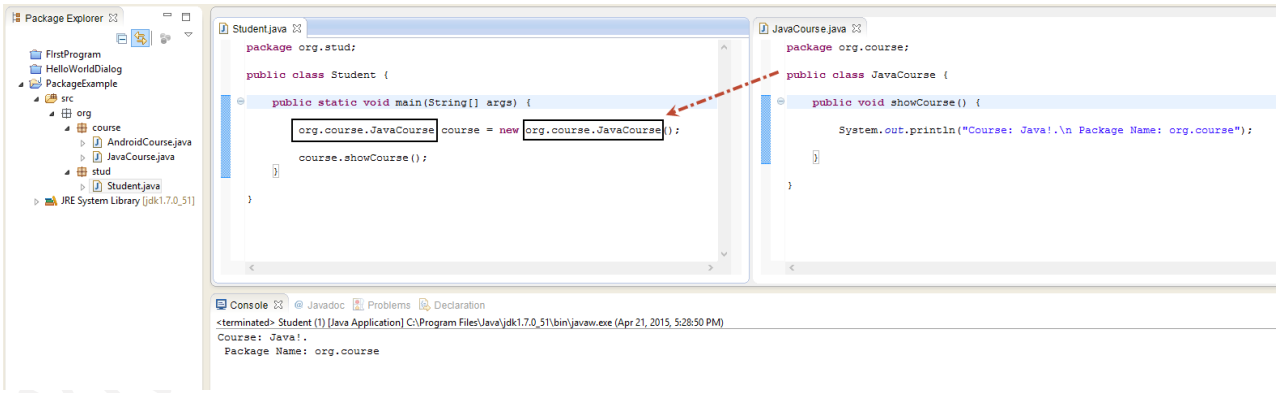


3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

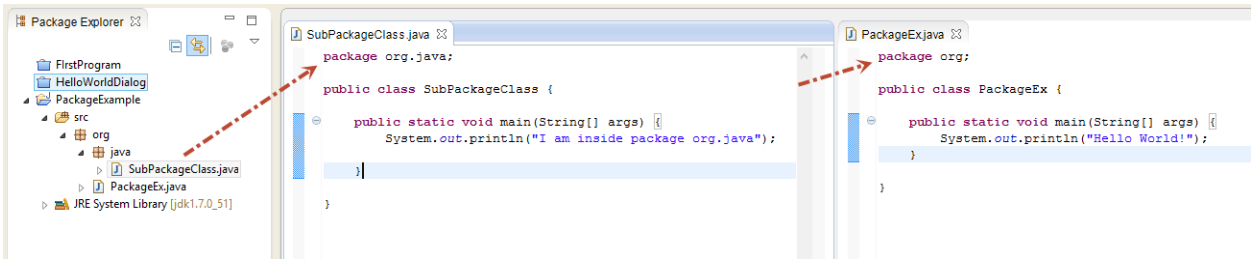
It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name



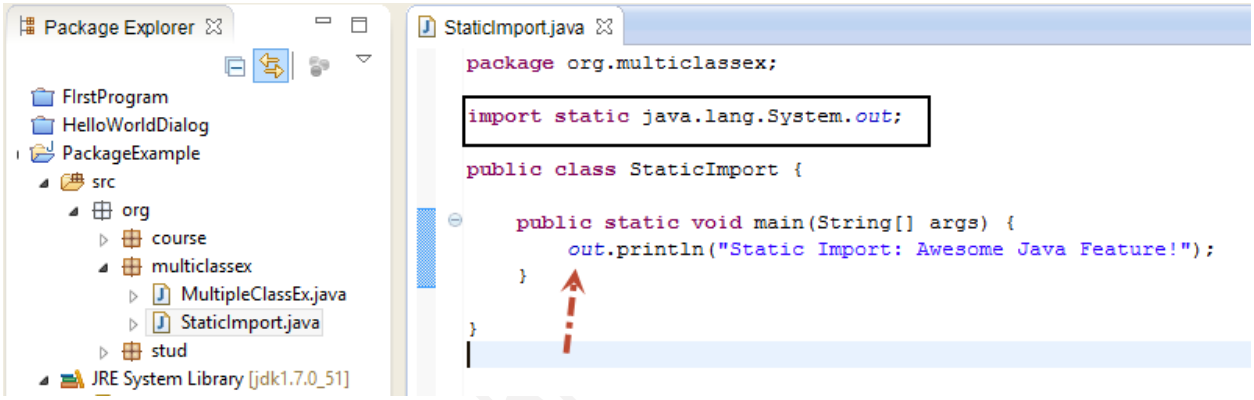
Subpackage in java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.



Static Import:

The static import feature of Java 5 facilitate the java programmer to access any static member of a class directly. There is no need to qualify it by the class name.



Access Modifier: public, private, protected

What is the default access modifier in java?

Default access modifier in Java (or) No access modifier in Java

```
public class Account{
  public long amount= 100;
  }
```

If you see the above variable *amount*, you can access anywhere either with in the class, sub classes, even in other packages if you have an instance of `Account` in your hand. You have no clue who is modifying your money and you cannot have much control on it.

```
public class Account{
```

```
  protected long amount= 100;
  }
```

If it is protected, this is a kind of *protected* state (in a *shell*), where only with in the class and sub classes can access (SavingsAccount,CurrentAccount etc ..) and for with in the package which is the class present can access it. With in the package means the package that Account class presented.

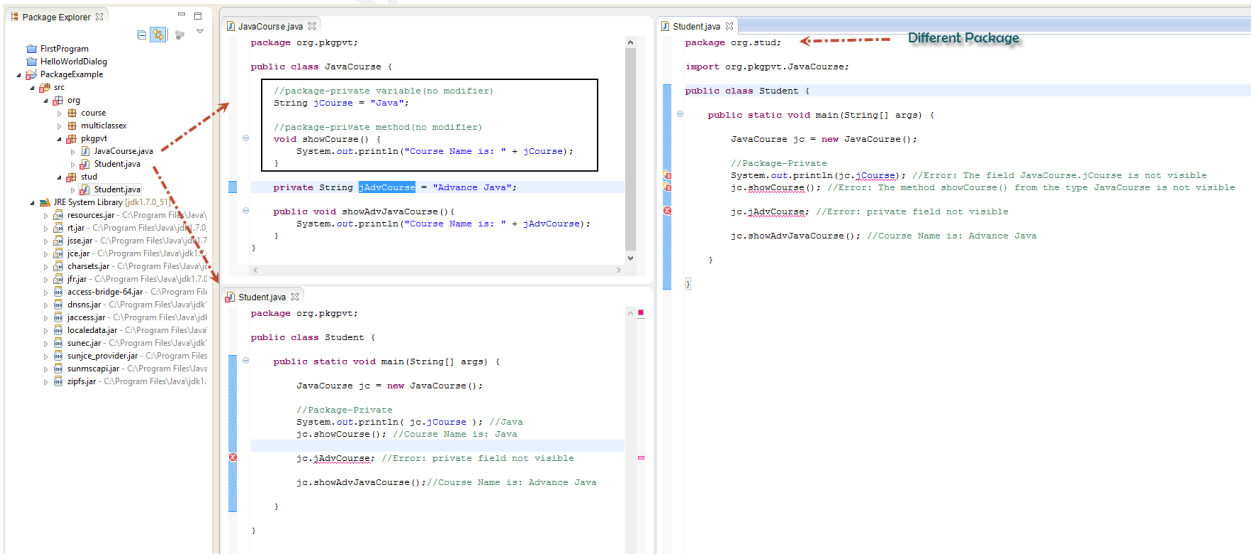
```
public class Account{
  private long amount= 100;
  }
```

Where as in case of *private*, it is strictly accessible only in the current class. No one see the variable out side from class. If anyone still want to expose the private variables value, provide a getter and return the value where there is no way to access the variable outside from the class.

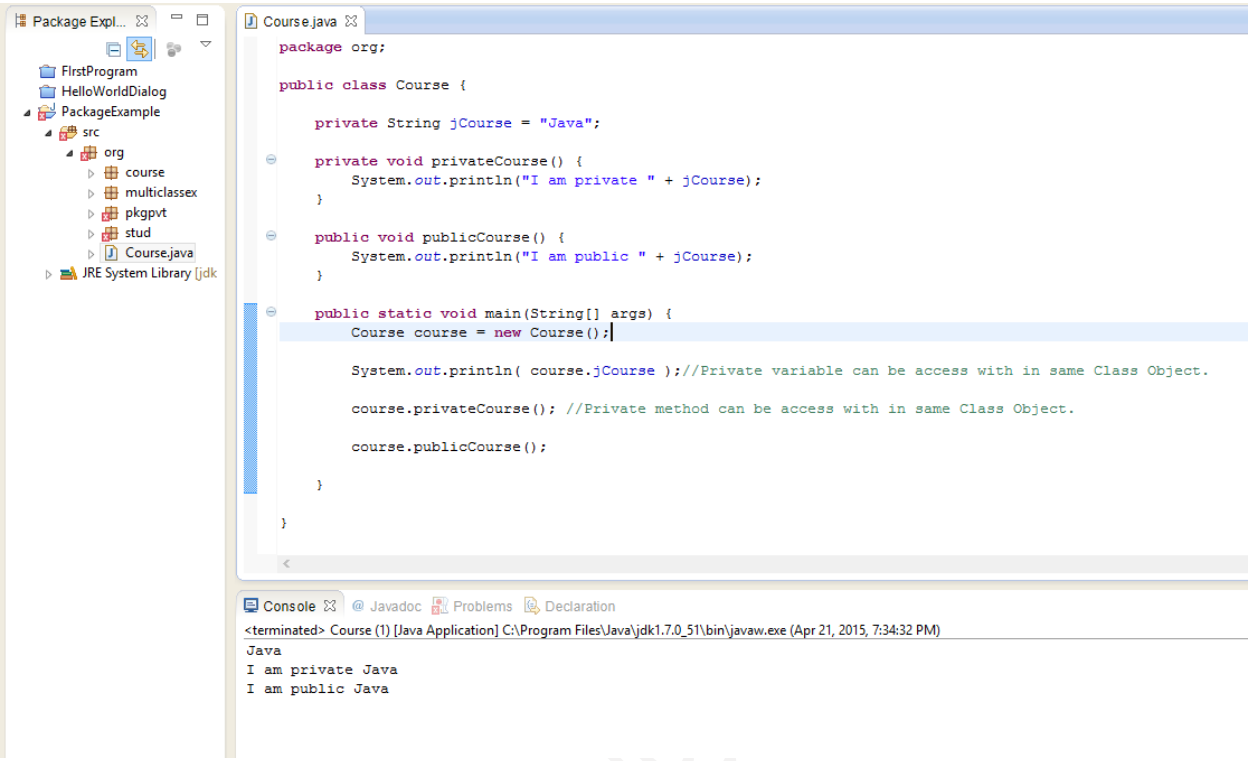
PACKAGE-PRIVATE

```
public class AccessDemo {
  int instanceCount= 1;    //Package-Private
  }
```

So, what if one didn't specify the access modifier ?? Compiler treats that as a default modifier aka no modifier.When there is no modifier specify to any of the members in a class, compiler treats that member as a default accessible type member. A default accessible type member is Package Private member. Only with in the Class and with in the Package only you can access that default member. But It is always recommended to specify the access specifier.



private/public/protected all type of variable/method can be accessed with in same class:



Summary of Access Control:

Controlling Access to Members of a Class

Access level modifiers determine whether other classes can use a particular field or invoke a particular method. There are two levels of access control:

- At the top level—`public`, or *package-private* (no explicit modifier).
- At the member level—`public`, `private`, `protected`, or *package-private* (no explicit modifier).

A class may be declared with the modifier `public`, in which case that class is visible to all classes everywhere. If a class has no modifier (the default, also known as *package-private*), it is visible only within its own package (packages are named groups of related classes — you will learn about them in a later lesson.)

At the member level, you can also use the `public` modifier or no modifier (*package-private*) just as with top-level classes, and with the same meaning. For members, there are two additional access modifiers: `private` and `protected`. The `private` modifier specifies that the member can only be accessed in its own class. The `protected` modifier specifies that the member can only be accessed within its own package (as with *package-private*) and, in addition, by a subclass of its class in another package.

The following table shows the access to members permitted by each modifier.

Access Levels				
Modifier	**Class**	**Package**	**Subclass**	**World**
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

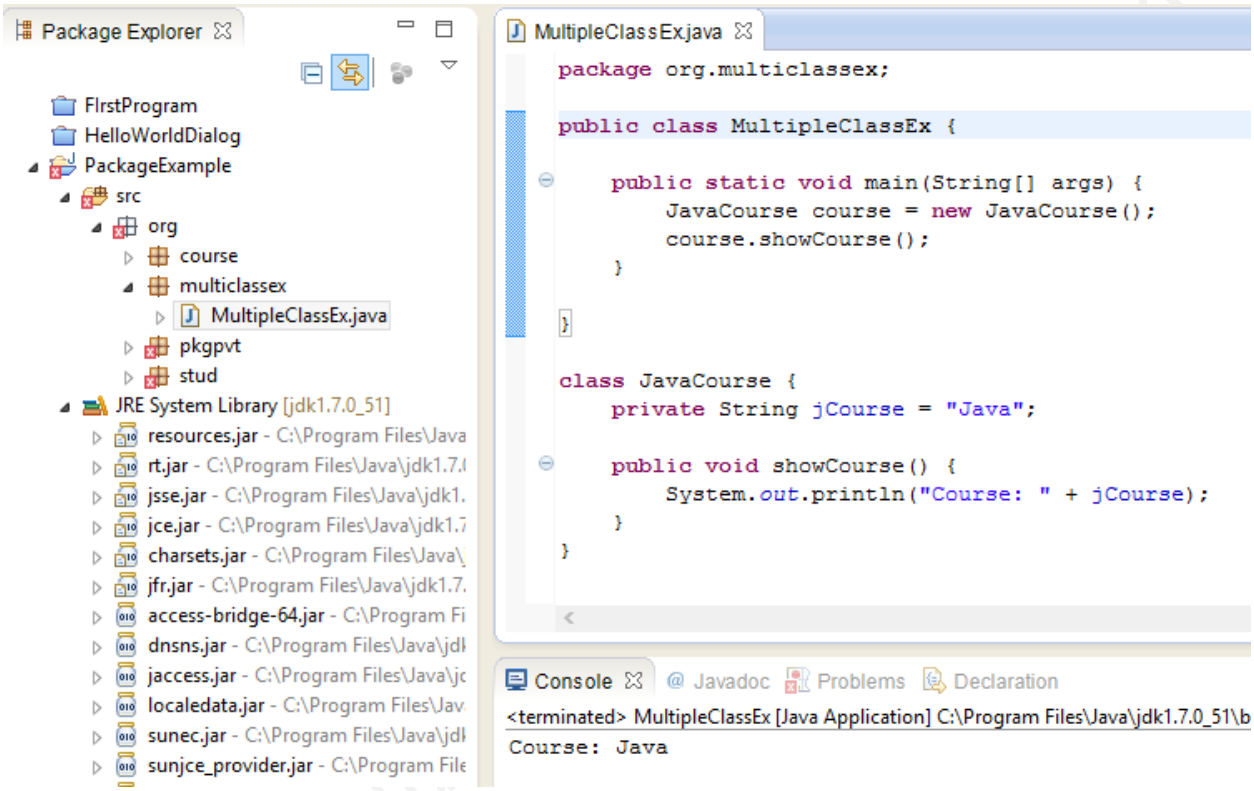
The first data column indicates whether the class itself has access to the member defined by the access level. As you can see, a class always has access to its own members. The second column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member. The third column indicates whether subclasses of the class declared outside this package have access to the member. The fourth column indicates whether all classes have access to the member.

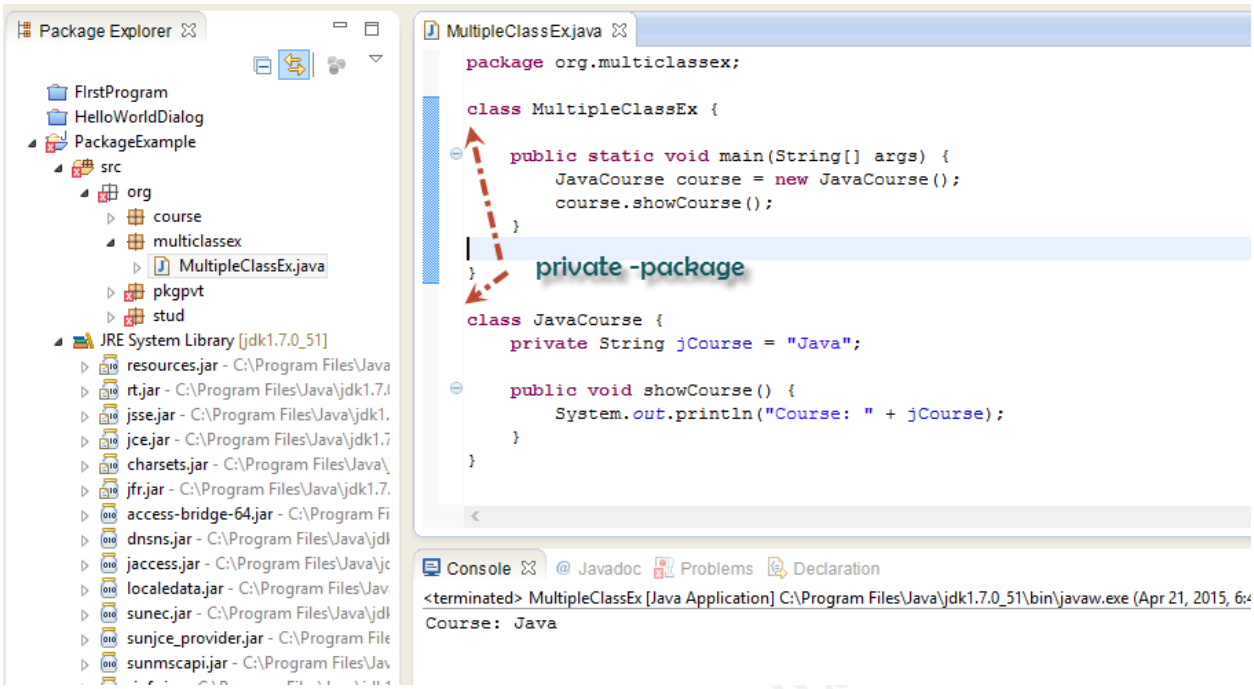
Access levels affect you in two ways. First, when you use classes that come from another source, such as the classes in the Java platform, access levels determine which members of those classes your own classes can use. Second, when you write a class, you need to decide what access level every member variable and every method in your class should have.

Multiple Class in one Java File: (Bad Java Programming Practice) ☹

- There can be only one public class in a java source file and it must be saved by the public class name.
- Generally class cannot be private (***only public, abstract & final are permitted***). If you need so, better use nested Inner Class. One class is defined inside another class is called inner class.

Multiple Class Example:





Nested Class: (we will read in later chapter):

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

Constructor in Java

Constructor in java is a *special type of method* that is used to initialize the object.

Java constructor is *invoked at the time of object creation*. It constructs the values i.e. provides data for the object that is why it is known as constructor.

Rules for creating java constructor

There are basically two rules defined for the constructor.

1. Constructor name must be same as its class name
2. Constructor must have no explicit return type

Types of java constructors

There are two types of constructors:

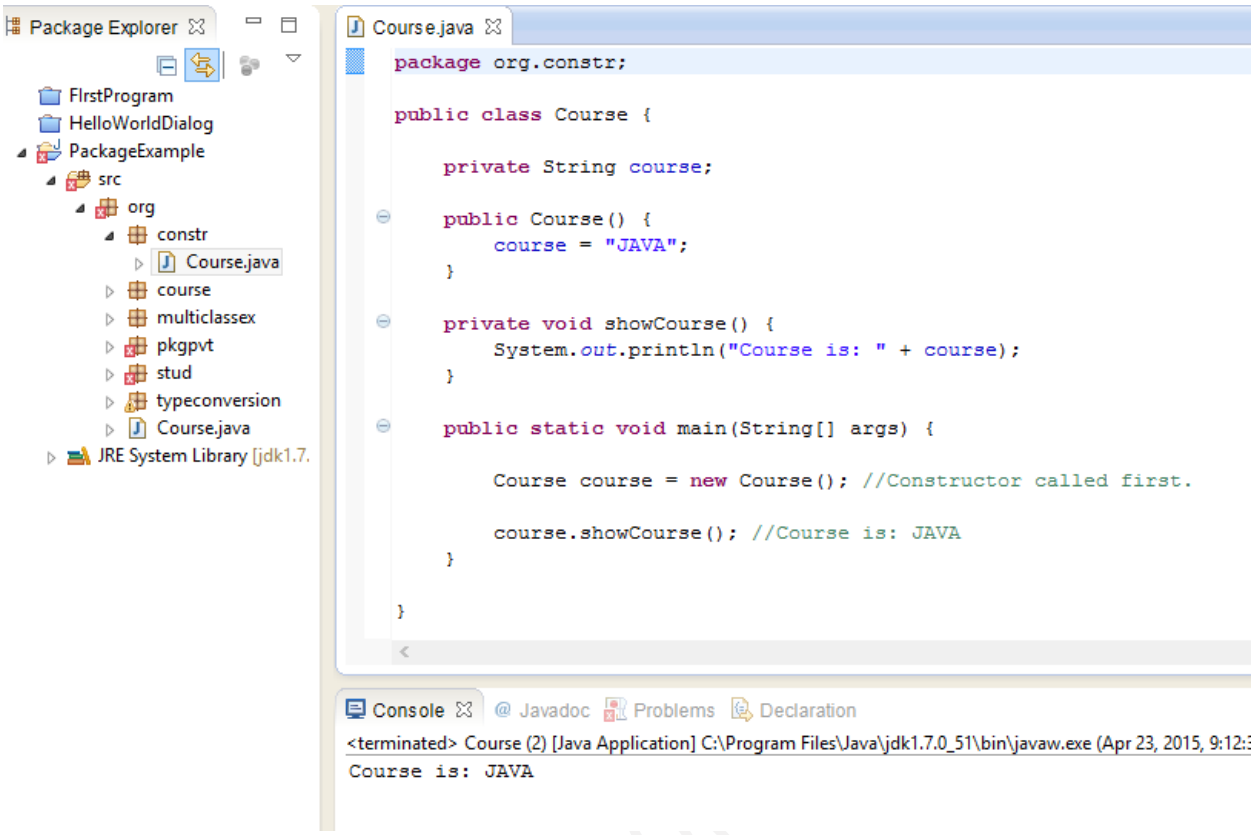
1. Default constructor (no-arg constructor)
2. Parameterized constructor

Java Default Constructor

A constructor that have no parameter is known as default constructor.

Syntax of default constructor:

```
<class_name>(){

}
```



Q) What is the purpose of default constructor?

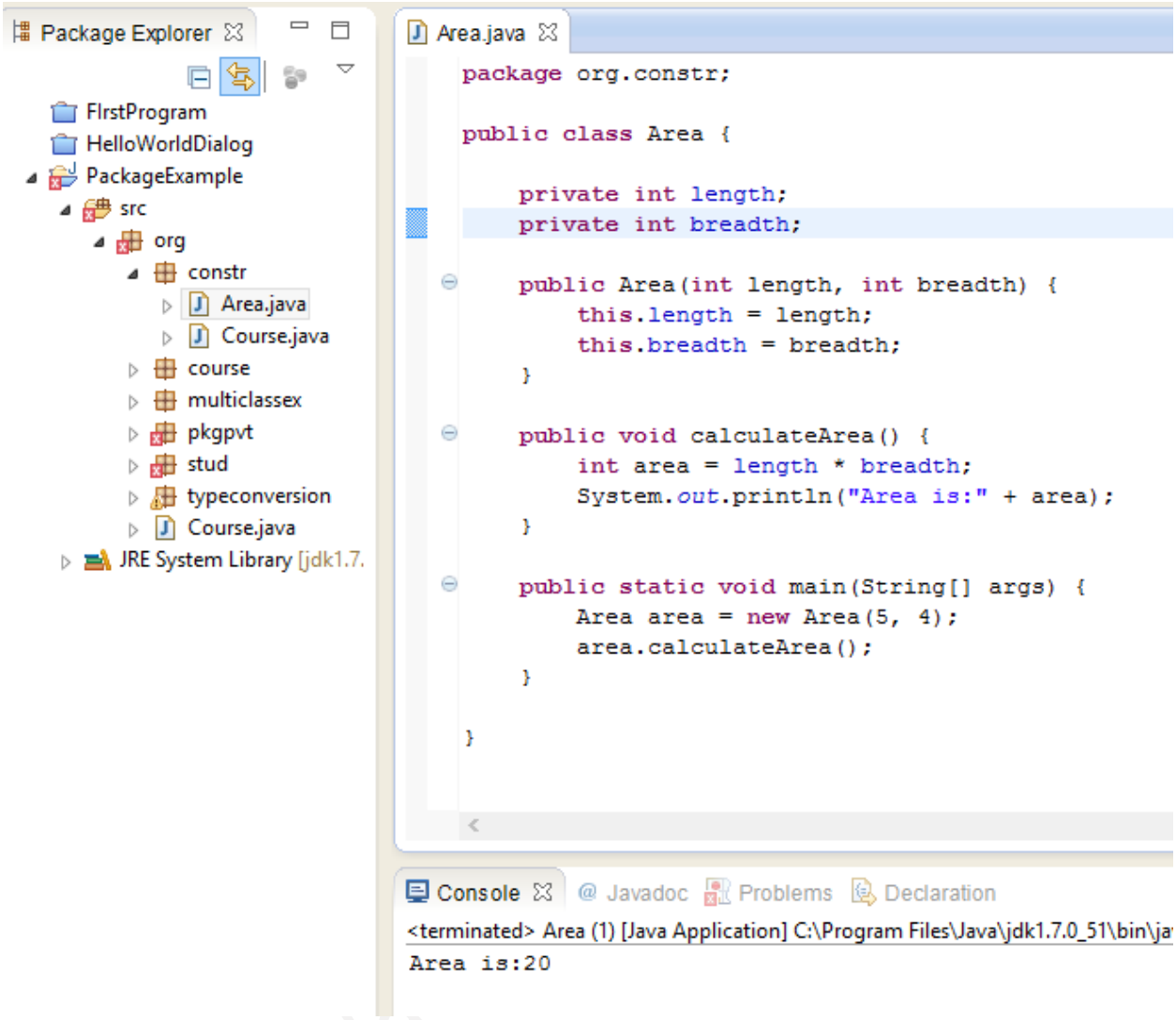
Default constructor provides the default values to the object like 0, null etc. depending on the type.

Java parameterized constructor

A constructor that have parameters is known as parameterized constructor.

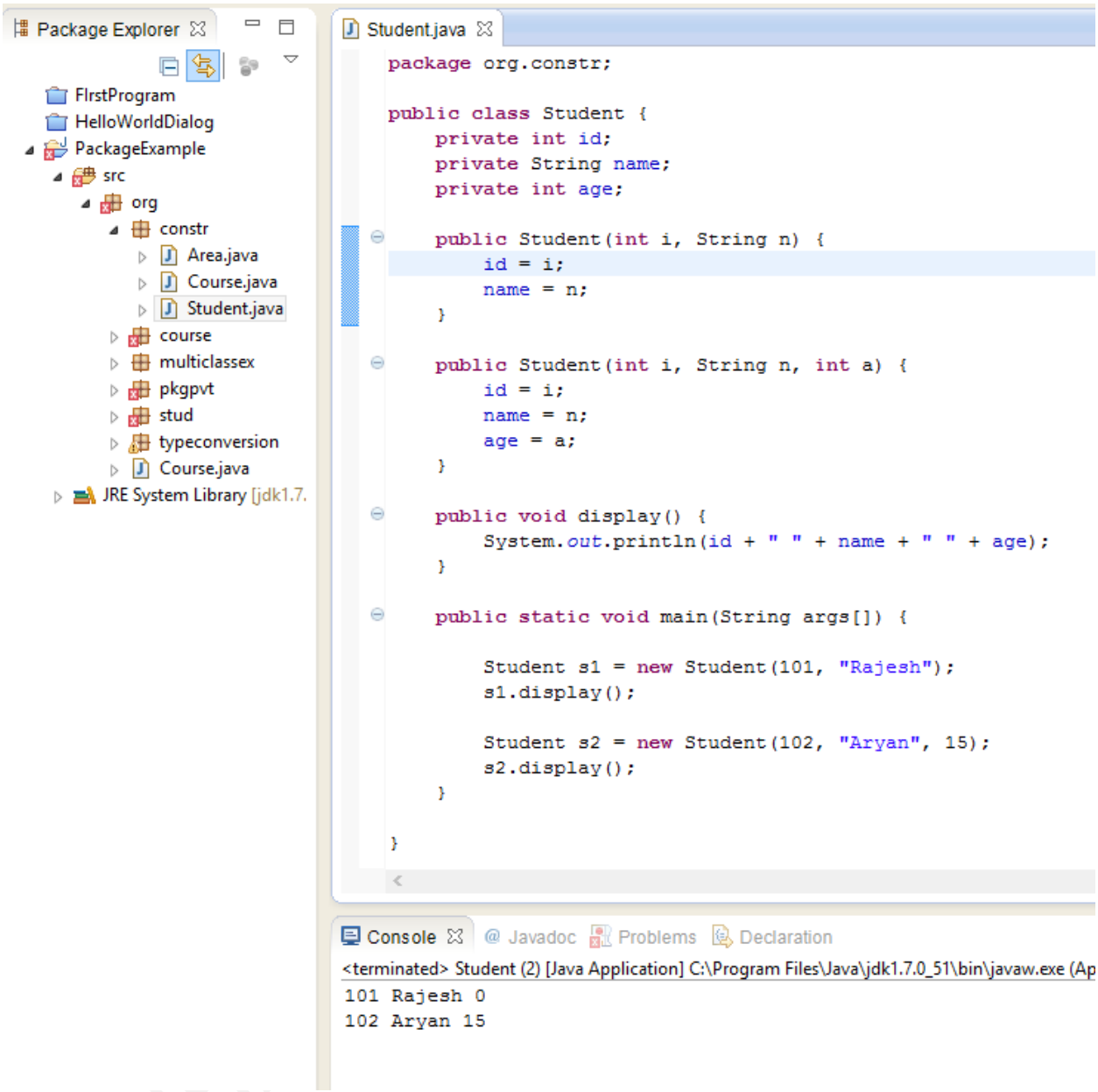
Why use parameterized constructor?

Parameterized constructor is used to provide different values to the distinct objects.



Constructor Overloading in Java

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists. The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.



Difference between constructor and method in java

Java Constructor	Java Method
Constructor is used to initialize the state of an object.	Method is used to expose behaviour of an object.
Constructor must not have return type.	Method must have return type.

Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name may or may not be same as class name.

Q) Does constructor return any value?

Ans:yes, that is current class instance (You cannot use return type yet it returns a value).

Can constructor perform other tasks instead of initialization?

Yes, like object creation, starting a thread, calling method etc. You can perform any operation in the constructor as you perform in the method.

Enumerated Types(enum):

An *enum type* is a special data type that enables for a variable to be a set of predefined constants. In the Java programming language, you define an enum type by using the `enum` keyword. For example, you would specify a days-of-the-week enum type as:

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}

public void tellItLikeItIs(Day day) {
        switch (day) {
            case MONDAY:
                System.out.println("Mondays are bad.");
                break;

            case FRIDAY:
                System.out.println("Fridays are better.");
                break;

            case SATURDAY: case SUNDAY:
                System.out.println("Weekends are best.");
                break;

            default:
                System.out.println("Midweek days are so-so.");
                break;
        }
}
```

How to represent enumerable value without Java enum:

```
public class CurrencyDenom {
   public static final int PENNY = 1;
   public static final int NICKLE = 5;
   public static final int DIME = 10;
   public static final int QUARTER = 25;
}

public class Currency {
   private int currency; //CurrencyDenom.PENNY,CurrencyDenom.NICKLE,
                         // CurrencyDenom.DIME,CurrencyDenom.QUARTER
}
```

Though this can server our purpose it has some serious limitations:

1) No Type-Safety: First of all it's not type-safe; you can assign any valid int value to currency e.g. 99 though there is no coin to represent that value.

2) No Meaningful Printing: printing value of any of these constant will print its numeric value instead of meaningful name of coin e.g. when you print NICKLE it will print "5" instead of "NICKLE"

3) No namespace: to access the `currencyDenom` constant we need to prefix class name e.g. `CurrencyDenom.PENNY` instead of just using PENNY though this can also be achieved by using static import in JDK 1.5
`Java Enum` is answer of all this limitation. Enum in Java is type-safe, provides meaningful String names and has there own namespace. Now let's see same example using Enum in Java:

```
public enum Currency {PENNY, NICKLE, DIME, QUARTER};
```

Here Currency is our **enum** and `PENNY, NICKLE, DIME, QUARTER` are **enum constants**. Notice **curly braces around enum constants** because Enum are type like class and interface in Java. Also we have followed similar naming convention for enum like class and interface (first letter in Caps) and since *Enum constants are implicitly static final* we have used all caps to specify them like Constants in Java.

Java Enum is type like class and interface and can be used to define a set of Enum constants. Enum constants are implicitly static and final and you can not change there value once created. Enum in Java provides type-safety and can be used inside switch statment like int variables.

Important points about Enum in Java

1) **Enums in Java are type-safe** and has there own name-space. It means your enum will have a type for example "Currency" in below example and you can not assign any value other than specified in Enum Constants.

```
public enum Currency {PENNY, NICKLE, DIME, QUARTER};
Currency coin = Currency.PENNY;
coin = 1; //compilation error
```

2**) Enum in Java are reference type** like class or interface and you can define constructor, methods and variables inside java Enum which makes it more powerful than Enum in C and C++ as shown in next example of Java Enum type.

3) You can **specify values of enum constants at the creation time** as shown in below example:

```
public enum Currency {PENNY(1), NICKLE(5), DIME(10), QUARTER(25)};
```

But for this to work you need to define a member variable and a constructor because PENNY (1) is actually calling a constructor which accepts int value , see below example.

```
public enum Currency {
```

```
        PENNY(1), NICKLE(5), DIME(10), QUARTER(25);
        private int value;

        private Currency(int value) {
                this.value = value;
        }
};
```

Constructor of enum in java must be **private** any other access modifier will result in compilation error. Now to get the value associated with each coin you can define a `public getValue()` method inside java enum like any normal java class. Also semi colon in the first line is optional.

4) Enum constants are implicitly static and final and can not be changed once created. For example below code of java enum will result in compilation error:

```
Currency.PENNY = Currency.DIME;
```

The final field `EnumExamples.Currency.PENNY` cannot be re assigned.

5) **Enum in java can be used as an argument on switch statment** and with "case:" like int or char primitive type. This feature of java enum makes them very useful for switch operations. Let's see an example of how to use java enum inside switch statement:

```
Currency usCoin = Currency.DIME;
   switch (usCoin) {
           case PENNY:
                   System.out.println("Penny coin");
                   break;
           case NICKLE:
                   System.out.println("Nickle coin");
                   break;
           case DIME:
                   System.out.println("Dime coin");
                   break;
           case QUARTER:
                   System.out.println("Quarter coin");
   }
```

from JDK 7 onwards you can also String in Switch case in Java code.

6) Since **constants defined inside Enum in Java are final you can safely compare them using "==" equality operator** as shown in following example of Java Enum:

```
Currency usCoin = Currency.DIME;
if(usCoin == Currency.DIME){
  System.out.println("enum in java can be compared using ==");
}
```

By the way comparing objects using == operator is not recommended, Always use equals() method or compareTo() method to compare Objects.

7) Java compiler automatically generates static `values()` method for every enum in java. `Values()` `method` returns array of Enum constants in the same order they have listed in Enum and you can use `values()` to iterate over values of Enum in Java as shown in below example:

```
for(Currency coin: Currency.values()){
        System.out.println("coin: " + coin);
}
```

And it will print:

```
coin: PENNY
coin: NICKLE
coin: DIME
coin: QUARTER
```

Notice the order its exactly same **with defined order in enums**.

Other Data Types:

Java String

Conceptually, Java strings are sequences of Unicode characters. String is basically an object that represents sequence of char values. java.lang.String class is used to create string object.

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

```
char[] ch={'j','a','v','a'};
String s=new String(ch);
```

is same as:

```
String s="java";
String e = ""; // an empty string
```

There are two ways to create String object:
By string literal: String s="java";
By new keyword : String s=new String(ch);

Java String literal is created by using double quotes.

The `String` class supports several constructors. To create an empty `String`, call the default constructor. For example,

```
String s = new String();

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

String s = new String(chars);
```

Java String class methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

No.	Method	Description
1	char charAt(int index)	returns char value for the particular index
2	int length()	returns string length

3	static String format(String format, Object... args)	returns formatted string
4	static String format(Locale l, String format, Object... args)	returns formatted string with given locale
5	String substring(int beginIndex)	returns substring for given begin index
6	String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index
7	boolean contains(CharSequence s)	returns true or false after matching the sequence of char value
8	static String join(CharSequence delimiter, CharSequence... elements)	returns a joined string
9	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	returns a joined string
10	boolean equals(Object another)	checks the equality of string with object
11	boolean isEmpty()	checks if string is empty
12	String concat(String str)	concatinates specified string
13	String replace(char old, char new)	replaces all occurrences of specified char value
14	String replace(CharSequence old, CharSequence new)	replaces all occurrences of specified CharSequence
15	String trim()	returns trimmed string omitting leading and trailing spaces
16	String split(String regex)	returns splitted string matching regex
17	String split(String regex, int limit)	returns splitted string matching regex and limit

18	String intern()	returns interned string
19	int indexOf(int ch)	returns specified char value index
20	int indexOf(int ch, int fromIndex)	returns specified char value index starting with given index
21	int indexOf(String substring)	returns specified substring index
22	int indexOf(String substring, int fromIndex)	returns specified substring index starting with given index
23	String toLowerCase()	returns string in lowercase.
24	String toLowerCase(Locale l)	returns string in lowercase using specified locale.
25	String toUpperCase()	returns string in uppercase.
26	String toUpperCase(Locale l)	returns string in uppercase using specified locale.

Substrings

You can extract a substring from a larger string with the substring method of the String class.
For example,

```
String greeting = "Hello";
String s = greeting.substring(0, 3);
```

creates a string consisting of the characters "Hel".

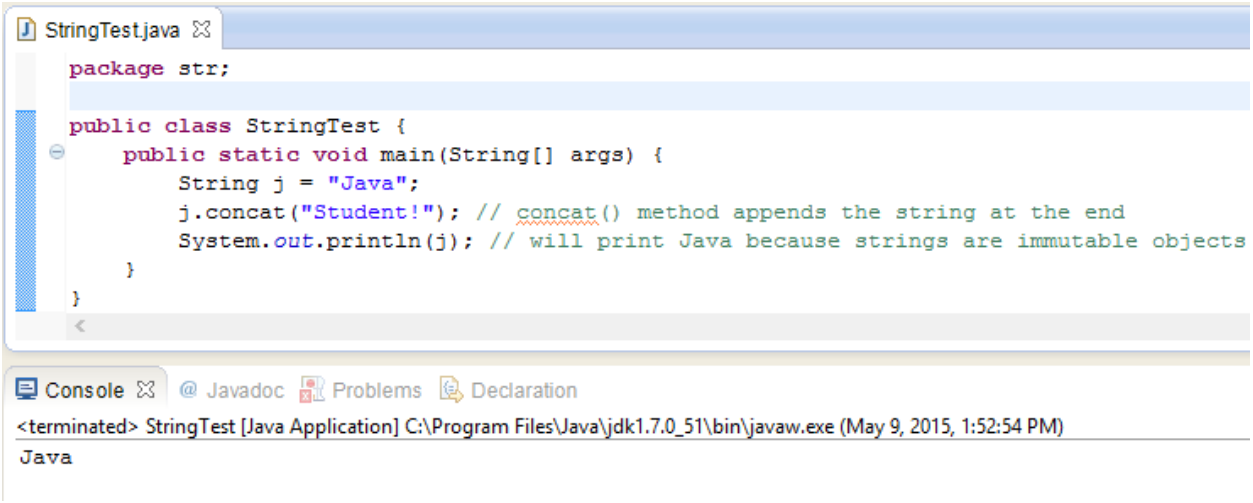
The second parameter of substring is the first position that you *do not* want to copy. In our case, we want to copy positions 0, 1, and 2 (from position 0 to position 2 inclusive). As substring counts it, this means from position **0 inclusive** to position **3 *exclusive***. There is one advantage to the way substring works: Computing the length of the substring is easy.
The string s.substring(a, b) always has length b - a. For example, the substring "Hel" has

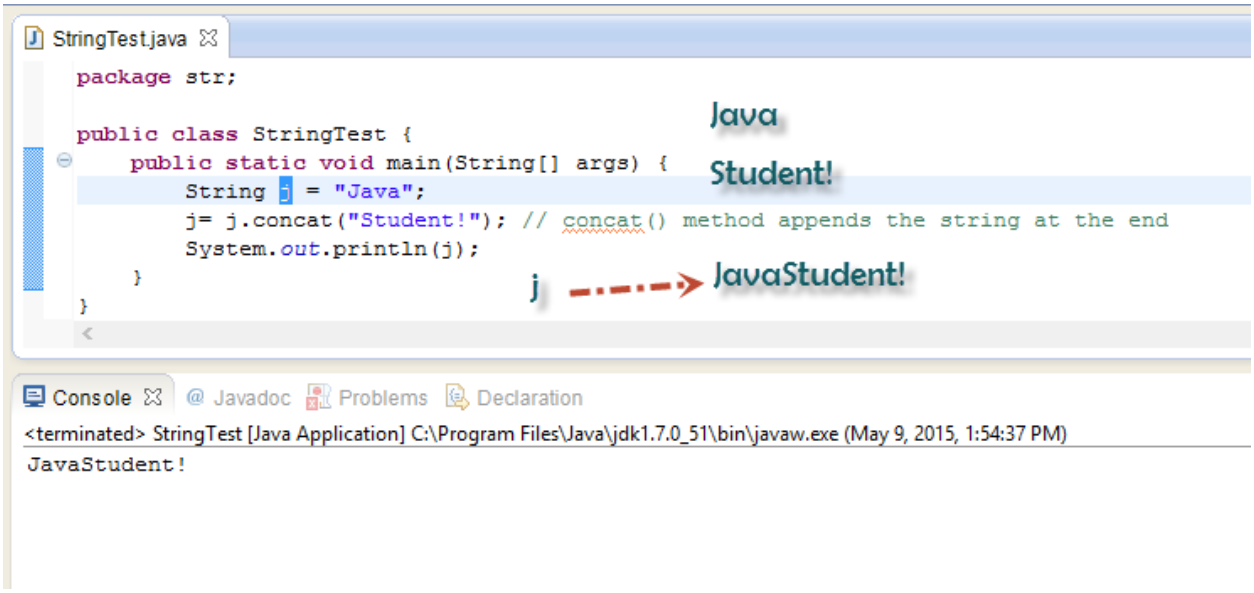
length 3 – 0 = 3.

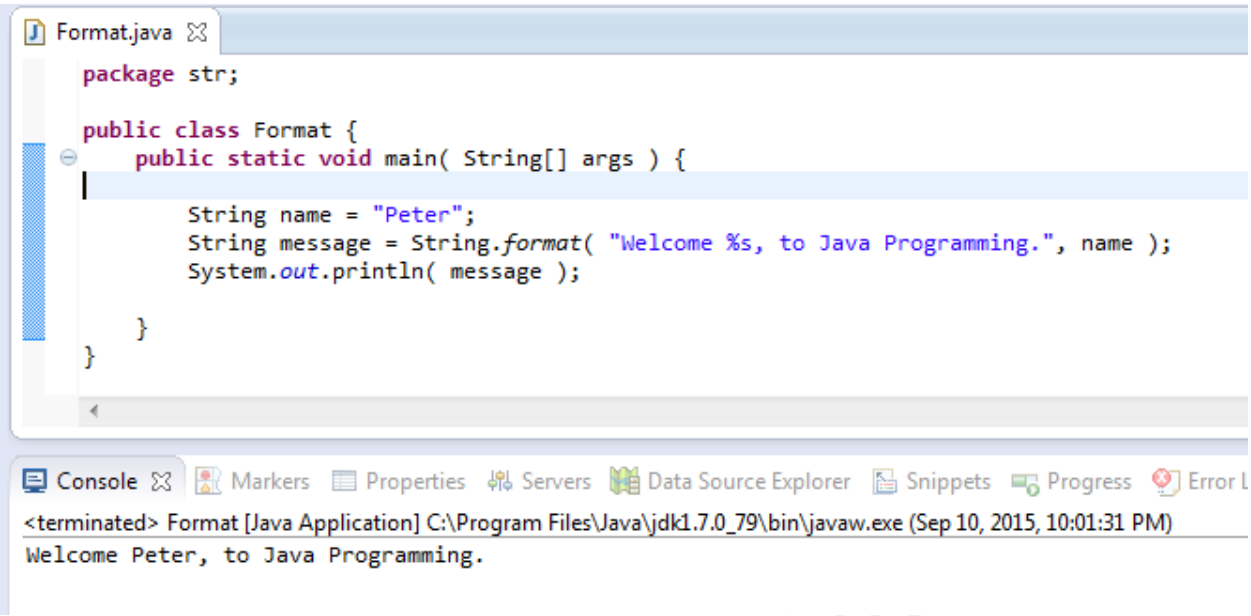
Strings Are Immutable

In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.

Once string object is created its data or state can't be changed but a new string object is created. The String class gives no methods that let you *change* a character in an existing string.







Concatenation

Java, like most programming languages, allows you to use + to join (concatenate) two strings.

String expletive = "Expletive";
String PG13 = "deleted";

String message = expletive + PG13; // "Expletivedeleted"

int age = 13;
String rating = "PG" + age;

sets rating to the string "PG13".

Testing Strings for Equality

To test whether two strings are equal, use the equals method. The expression s.equals(t)
returns true if the strings s and t are equal, false otherwise. Note that s and t can be string variables or string constants.
String h = “Hello”;
String j = “Java”;
h.equals(j) //return false

"Hello".equals(j) //return false
"Hello".equalsIgnoreCase("hello") //return true
"Hello".equalsIgnoreCase(h) //return true

h.equalsIgnoreCase(“Hello”)// return true

Do *not* use the == operator to test whether two strings are equal!
It only determines whether or not the strings are stored in the same location. Sure, if strings are in the same location, they must be equal. But it is entirely possible to store multiple copies of identical strings in different places.

StringTest.java

```
package str;

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

        String hel = "Hello".substring(0, 3); // Hel
        System.out.println("String hel = " + hel);

        boolean eqVal = (hel == "Hel");
        System.out.println("hel == 'Hel' is: " + eqVal);

        boolean eqFun = hel.equals("Hel");
        System.out.println("hel.equals('Hel') is: " + eqFun);
    }
}
```

Console | Javadoc | Problems | Declaration

```
<terminated> StringTest [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (May 9, 2015, 2:22:32 PM)
String hel = Hel
hel == 'Hel' is: false
hel.equals('Hel') is: true
```

If the virtual machine always arranges for equal strings to be shared, then you could use the == operator for testing equality. But only string constants are shared, not strings that are the result of operations like + or substring. Therefore, never use == to compare strings lest you end up with a program with the worst kind of bug—an intermittent one that seems to occur randomly.

String h = “Hello”;

H == “Hello” //may be true

Empty and Null Strings

The empty string "" is a string of length 0. You can test whether a string is empty by calling
if (str.length() == 0)

or

```
if (str.equals(""))
```

However, a String variable can also hold a special value,called null, that indicates that no object is currently associated with the variable.

```
str = null;
```

To test whether a string is null, use the condition

```
if (str == null)
```

Sometimes, you need to test that a string is neither null nor empty. Then use the condition

```
if (str != null && str.length() != 0)
```

char charAt(int index): returns the code unit at the specified location.

int compareTo(String other): returns a negative value if the string comes before other in dictionary order, a positive value if the string comes after other in dictionary order, or 0 if the strings are equal.

boolean endsWith(String suffix): returns true if the string ends with suffix.

boolean equals(Object other): returns true if the string equals other.

boolean equalsIgnoreCase(String other): returns true if the string equals other, except for upper/lowercase distinction.

int indexOf(String str)

- int indexOf(String str, int fromIndex)
- int indexOf(int cp)
- int indexOf(int cp, int fromIndex)

returns the start of the first substring equal to the string str or the code point cp, starting at index 0 or at fromIndex, or -1 if str does not occur in this string.

- int lastIndexOf(String str)
- int lastIndexOf(String str, int fromIndex)
- int lastindexOf(int cp)
- int lastindexOf(int cp, int fromIndex)

returns the start of the last substring equal to the string str or the code point cp, starting at the end of the string or at fromIndex.

- int length(): returns the length of the string.

String replace(CharSequence oldString, CharSequence newString)
returns a new string that is obtained by replacing all substrings matching oldString in the string with the string newString. You can supply String or StringBuilder objects for the CharSequence parameters.

- boolean startsWith(String prefix)

returns true if the string begins with prefix.

- String substring(int beginIndex)
- String substring(int beginIndex, int endIndex)

returns a new string consisting of all code units from beginIndex until the end of the string or until endIndex - 1.

- String toLowerCase()

returns a new string containing all characters in the original string, with uppercase characters converted to lowercase.

- String toUpperCase()

returns a new string containing all characters in the original string, with lowercase characters converted to uppercase.

- String trim()

returns a new string by eliminating all leading and trailing spaces in the original string.

```
String[] split(String regex):
String s1="java string split method ";
String[] words=s1.split("\\s");
```

NOTE:

```
String s = "four: " + 2 + 2;
System.out.println (s);
```

This fragment displays

four: 22

rather than the

four: 4

```
String s = "four: " + (2 + 2);
```

Now s contains the string “four: 4”.

toString():

If you want to represent any object as a string, **toString() method** comes into existence.

The toString() method returns the string representation of the object.

If you print any object, java compiler internally invokes the toString() method on the object. So overriding the toString() method, returns the desired output, it can be the state of an object etc. depends on your implementation.

Advantage of Java toString() method

By overriding the toString() method of the Object class, we can return values of the object, so we don't need to write much code.

Understanding problem without toString() method

Let's see the simple code that prints reference.

```
1. class Student{
2.  int rollno;
3.  String name;
4.  String city;
5.
6.  Student(int rollno, String name, String city){
7.  this.rollno=rollno;
8.  this.name=name;
9.  this.city=city;
10. }
11.
12. public static void main(String args[]){
13.   Student s1=new Student(101,"Raj","KTM");
14.   Student s2=new Student(102,"Vijay","NPL");
15.
16.   System.out.println(s1);//compiler writes here s1.toString()
17.   System.out.println(s2);//compiler writes here s2.toString()
18. }
19. }
```

```
Output:Student@1fee6fc
       Student@1eed786
```

As you can see in the above example, printing s1 and s2 prints the hashcode values of the objects but I want to print the values of these objects. Since java compiler internally calls toString() method, overriding this method will return the specified values. Let's understand it with the example given below:

Example of Java toString() method

Now let's see the real example of toString() method.

```
1. class Student{
2.  int rollno;
3.  String name;
4.  String city;
5.
6.  Student(int rollno, String name, String city){
7.  this.rollno=rollno;
8.  this.name=name;
9.  this.city=city;
10. }
11.
12. public String toString(){//overriding the toString() method
13.   return rollno+" "+name+" "+city;
14. }
15. public static void main(String args[]){
16.   Student s1=new Student(101,"Raj","KTM");
17.   Student s2=new Student(102,"Vijay","NPL");
18.
19.   System.out.println(s1);//compiler writes here s1.toString()
20.   System.out.println(s2);//compiler writes here s2.toString()
21. }
22. }
```

```
Output:101 Raj KTM
       102 Vijay NPL
```

For those cases in which a modifiable string is desired, Java provides two options: **StringBuffer** and **StringBuilder**. Both hold strings that can be modified after they are created.

The **String**, **StringBuffer**, and **StringBuilder** classes are defined in java.lang. Thus, they are available to all programs automatically. All are declared final, which means that none of these classes may be subclassed. This allows certain optimizations that increase performance to take place on common string operations. All three implement the **CharSequence** interface.

One last point: To say that the strings within objects of type String are unchangeable means that the contents of the **String** instance cannot be changed after it has been created. However, a variable declared as a **String** reference can be changed to point at some other **String** object at any time.

Java StringBuffer class

Java StringBuffer class is used to created mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

Note: Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.

Important Constructors of StringBuffer class

1. **StringBuffer():** creates an empty string buffer with the initial capacity of 16.
2. **StringBuffer(String str):** creates a string buffer with the specified string.
3. **StringBuffer(int capacity):** creates an empty string buffer with the specified capacity as length.

Important methods of StringBuffer class

1. **public synchronized StringBuffer append(String s):** is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
2. **public synchronized StringBuffer insert(int offset, String s):** is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
3. **public synchronized StringBuffer replace(int startIndex, int endIndex, String str):** is used to replace the string from specified startIndex and endIndex.
4. **public synchronized StringBuffer delete(int startIndex, int endIndex):** is used to delete the string from specified startIndex and endIndex.
5. **public synchronized StringBuffer reverse():** is used to reverse the string.
6. **public int capacity():** is used to return the current capacity.
7. **public void ensureCapacity(int minimumCapacity):** is used to ensure the capacity at least equal to the given minimum.
8. **public char charAt(int index):** is used to return the character at the specified position.
9. **public int length():** is used to return the length of the string i.e. total number of characters.
10. **public String substring(int beginIndex):** is used to return the substring from the specified beginIndex.
11. **public String substring(int beginIndex, int endIndex):** is used to return the substring from the specified beginIndex and endIndex.

What is mutable string

A string that can be modified or changed is known as mutable string. StringBuffer and StringBuilder classes are used for creating mutable string.

1) StringBuffer append() method

The append() method concatenates the given argument with this string.

```
1. class A{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer("Hello ");
4. sb.append("Java");//now original string is changed
5. System.out.println(sb);//prints Hello Java
6. }
7. }
```

2) StringBuffer insert() method

The insert() method inserts the given string with this string at the given position.

```
1. class A{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer("Hello ");
4. sb.insert(1,"Java");//now original string is changed
5. System.out.println(sb);//prints HJavaello
6. }
7. }
```

3) StringBuffer replace() method

The replace() method replaces the given string from the specified beginIndex and endIndex.

```
1. class A{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer("Hello");
4. sb.replace(1,3,"Java");
5. System.out.println(sb);//prints HJavalo
6. }
7. }
```

4) StringBuffer delete() method

The delete() method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

```
1. class A{
```

```
2.  public static void main(String args[]){
3.  StringBuffer sb=new StringBuffer("Hello");
4.  sb.delete(1,3);
5.  System.out.println(sb);//prints Hlo
6.  }
7.  }
```

5) StringBuffer reverse() method

The reverse() method of StringBuilder class reverses the current string.

```
1.  class A{
2.  public static void main(String args[]){
3.  StringBuffer sb=new StringBuffer("Hello");
4.  sb.reverse();
5.  System.out.println(sb);//prints olleH
6.  }
7.  }
```

Java StringBuilder class

Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

Important Constructors of StringBuilder class

1. **StringBuilder():** creates an empty string Builder with the initial capacity of 16.
2. **StringBuilder(String str):** creates a string Builder with the specified string.
3. **StringBuilder(int length):** creates an empty string Builder with the specified capacity as length.

Important methods of StringBuilder class

Method	Description
public StringBuilder append(String s)	is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.

public StringBuilder insert(int offset, String s)	is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public StringBuilder replace(int startIndex, int endIndex, String str)	is used to replace the string from specified startIndex and endIndex.
public StringBuilder delete(int startIndex, int endIndex)	is used to delete the string from specified startIndex and endIndex.
public StringBuilder reverse()	is used to reverse the string.
public int capacity()	is used to return the current capacity.
public void ensureCapacity(int minimumCapacity)	is used to ensure the capacity at least equal to the given minimum.
public char charAt(int index)	is used to return the character at the specified position.
public int length()	is used to return the length of the string i.e. total number of characters.
public String substring(int beginIndex)	is used to return the substring from the specified beginIndex.
public String substring(int beginIndex, int endIndex)	is used to return the substring from the specified beginIndex and endIndex.

Java StringBuilder Examples

Let's see the examples of different methods of StringBuilder class.

1) StringBuilder append() method

The StringBuilder append() method concatenates the given argument with this string.

```
1.  class A{
2.  public static void main(String args[]){
3.  StringBuilder sb=new StringBuilder("Hello ");
4.  sb.append("Java");//now original string is changed
5.  System.out.println(sb);//prints Hello Java
6.  }
```

```
7.    }
```

2) StringBuilder insert() method

The StringBuilder insert() method inserts the given string with this string at the given position.

```
1.  class A{
2.  public static void main(String args[]){
3.  StringBuilder sb=new StringBuilder("Hello ");
4.  sb.insert(1,"Java");//now original string is changed
5.  System.out.println(sb);//prints HJavaello
6.  }
7.  }
```

3) StringBuilder replace() method

The StringBuilder replace() method replaces the given string from the specified beginIndex and endIndex.

```
1.      class A{
2.      public static void main(String args[]){
3.      StringBuilder sb=new StringBuilder("Hello");
4.      sb.replace(1,3,"Java");
5.      System.out.println(sb);//prints HJavalo
6.      }
7.      }
```

4) StringBuilder delete() method

The delete() method of StringBuilder class deletes the string from the specified beginIndex to endIndex.

```
1.  class A{
2.  public static void main(String args[]){
3.  StringBuilder sb=new StringBuilder("Hello");
4.  sb.delete(1,3);
5.  System.out.println(sb);//prints Hlo
6.  }
7.  }
```

5) StringBuilder reverse() method

The reverse() method of StringBuilder class reverses the current string.

```
1.  class A{
2.  public static void main(String args[]){
3.  StringBuilder sb=new StringBuilder("Hello");
```

```
4.  sb.reverse();
5.  System.out.println(sb);//prints olleH
6.  }
7.  }
```

Difference between String and StringBuffer

There are many differences between String and StringBuffer. A list of differences between String and StringBuffer are given below:

No.	String	StringBuffer
1)	String class is immutable.	StringBuffer class is mutable.
2)	String is slow and consumes more memory when you concat too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when you cancat strings.
3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.

Object Wrappers and Autoboxing

Each of Java's eight primitive data types has a class dedicated to it. These are known as wrapper classes, because they "wrap" the primitive data type into an object of that class. The wrapper classes are part of the java.lang package, which is imported by default into all Java programs.
The wrapper classes in java servers two primary purposes.

- To provide mechanism to ‘wrap’ primitive values in an object so that primitives can do activities reserved for the objects like being added to ArrayList, Hashset, HashMap etc. collection.
- To provide an assortment of utility functions for primitives like converting primitive types to and from string objects, converting to various bases like binary, octal or hexadecimal, or comparing various objects.

The following two statements illustrate the difference between a primitive data type and an object of a wrapper class:

```
int  x = 25;
Integer  y = new Integer(33);
```

The first statement declares an int variable named x and initializes it with the value 25. The second statement instantiates an Integer object. The object is initialized with the value 33 and a reference to the object is assigned to the object variable y.

Below table lists wrapper classes in Java API with constructor details.

Primitive	Wrapper Class	Constructor Argument
boolean	Boolean	boolean or String
byte	Byte	byte or String
char	Character	char
int	Integer	int or String
float	Float	float, double or String
double	Double	double or String
long	Long	long or String
short	Short	short or String

Below is wrapper class hierarchy as per Java API

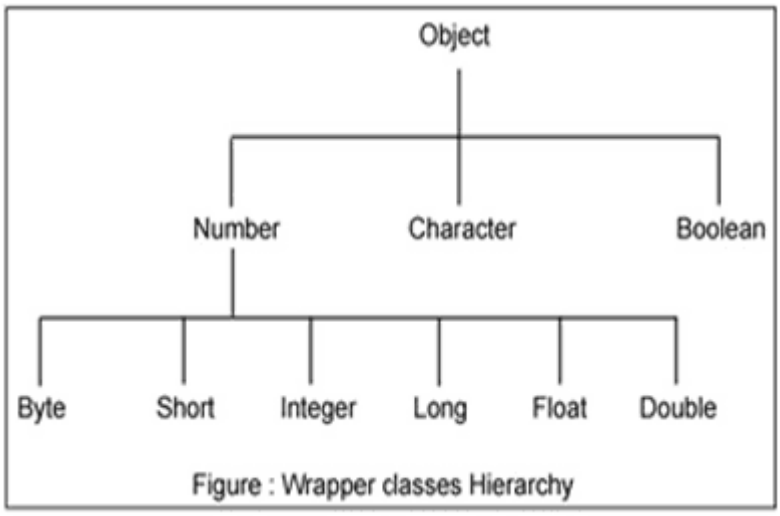


Figure : Wrapper classes Hierarchy

As explain in above table all wrapper classes (except Character) take String as argument constructor. Please note we might get NumberFormatException if we try to assign invalid argument in constructor. For example to create Integer object we can have following syntax.

```
Integer intObj = new Integer (25);

Integer intObj2 = new Integer ("25");
```

Here in we can provide any number as string argument but not the words etc. Below statement will throw run time exception (NumberFormatException)

```
Integer intObj3 = new Integer ("Two");
```

The following discussion focuses on the Integer wrapperclass, but applies in a general sense to all eight wrapper classes.

The most common methods of the Integer wrapper class are summarized in below table. Similar methods for the other wrapper classes are found in the Java API documentation.

Method	**Purpose**
parseInt(s)	returns a signed decimal integer value equivalent to string s
toString(i)	returns a new String object representing the integer i
byteValue()	returns the value of this Integer as a byte
doubleValue()	returns the value of this Integer as an double

floatValue()	returns the value of this Integer as a float
intValue()	returns the value of this Integer as an int
shortValue()	returns the value of this Integer as a short
longValue()	returns the value of this Integer as a long
int compareTo(int i)	Compares the numerical value of the invoking object with that of i. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static int compare(int num1, int num2)	Compares the values of num1 and num2. Returns 0 if the values are equal. Returns a negative value if num1 is less than num2. Returns a positive value if num1 is greater than num2.
boolean equals(Object intObj)	Returns true if the invoking Integer object is equivalent to intObj. Otherwise, it returns false.

WrapperIntro.java

```
public class WrapperIntro {

    public static void main( String[] args ) {
        Integer intObj1 = new Integer( 25 );

        Integer intObj2 = new Integer( "25" );

        Integer intObj3 = new Integer( 35 );

        // compareTo demo
        System.out.println( "Comparing using compareTo Obj1 and Obj2: " + intObj1.compareTo( intObj2 ) );

        System.out.println( "Comparing using compareTo Obj1 and Obj3: " + intObj1.compareTo( intObj3 ) );

        // Equals demo
        System.out.println( "Comparing using equals Obj1 and Obj2: " + intObj1.equals( intObj2 ) );

        System.out.println( "Comparing using equals Obj1 and Obj3: " + intObj1.equals( intObj3 ) );

        Float f1 = new Float( "2.25f" );

        Float f2 = new Float( "20.43f" );

        Float f3 = new Float( 2.25f );

        System.out.println( "Comparing using compare f1 and f2: " + Float.compare( f1, f2 ) );

        System.out.println( "Comparing using compare f1 and f3: " + Float.compare( f1, f3 ) );

        // Addition of Integer with Float
        Float f = intObj1.floatValue( ) + f1;

        System.out.println( "Addition of intObj1 and f1: " + intObj1 + "+" + f1 + "=" + f );
    }
}
```

Console | Markers | Properties | Servers | Data Source Explorer | Snippets | Progress | Error Log

```
<terminated> WrapperIntro (1) [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw.exe (Sep 10, 2015, 5:01:20 PM)
Comparing using compareTo Obj1 and Obj2: 0
Comparing using compareTo Obj1 and Obj3: -1
Comparing using equals Obj1 and Obj2: true
Comparing using equals Obj1 and Obj3: false
Comparing using compare f1 and f2: -1
Comparing using compare f1 and f3: 0
Addition of intObj1 and f1: 25+2.25=27.25
```

Autoboxing

Beginning with JDK 5, Java added two important features: *autoboxing* and *autounboxing*. Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed. There is no need to explicitly construct an object. Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed. There is no need to call a method such as **intValue( )** or **doubleValue( )**.

```
Integer iOb = 100; // autobox an int   ( new Integer(100)   )
int i = iOb; // auto-unbox              ( auto call iOb.intValue()  )

// Autobox/unbox a char.
Character ch = 'x'; // box a char
char ch2 = ch; // unbox a char
```

Converting Numbers to and from Strings

The Byte, Short, Integer, and Long classes provide the parseByte( ), parseShort( ), parseInt( ), and parseLong( ) methods, respectively. These methods return the byte, short, int, or long equivalent of the numeric string with which they are called. (Similar methods also exist for the Float and Double classes.)

```
String str = “5”;
int i = Integer.parseInt(str);
```

The Integer and Long classes also provide the methods **toBinaryString**( ), **toHexString**( ), and **toOctalString**( ), which convert a value into a binary, hexadecimal, or octal string, respectively.

```
int num = 19648;
System.out.println(num + " in binary: " + Integer.toBinaryString(num));
System.out.println(num + " in octal: " + Integer.toOctalString(num));
System.out.println(num + " in hexadecimal: " + Integer.toHexString(num));
```

The output of this program is shown here:

```
19648 in binary: 100110011000000
19648 in octal: 46300
19648 in hexadecimal: 4cc0
```

System

The **System** class holds a collection of static methods and variables. The standard input, output, and error output of the Java run time are stored in the **in**, **out**, and **err** variables.

Using currentTimeMillis( ) to Time Program Execution

Elapsed.java

```
public class Elapsed {
    public static void main( String[] args ) {
        long start, end;

        System.out.println( "Timing a for loop from 0 to 100,000,000" );

        // time a for loop from 0 to 100,000,000
        start = System.currentTimeMillis( ); // get starting time

        for ( long i = 0; i < 100000000L; i++ )
            ;

        end = System.currentTimeMillis( ); // get ending time

        System.out.println( "Elapsed time: " + ( end - start ) );
    }

}
```

Console | Markers | Properties | Servers | Data Source Explorer | Snippets | Progress

```
<terminated> Elapsed (1) [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw.exe (Sep 10, 2015, 5:13:0
Timing a for loop from 0 to 100,000,000
Elapsed time: 49
```

The three streams `System.in`, `System.out`, and `System.err`

System.in

`System.in` is an `InputStream` which is typically connected to keyboard input of console programs.

```
public class ScannerEx {

    public static void main(String[] args) {
        System.out.print("Write Something in console window: ");
        Scanner sc = new Scanner(System.in);
        String inputText = sc.next();
        System.out.println(inputText);
    }

}
```

System.out

```
public static final PrintStream out
```

```
The "standard" output stream. This stream is already open and ready to accept output
data. Typically this stream corresponds to display output or another output
destination specified by the host environment or user.
For simple stand-alone Java applications, a typical way to write a line of output
data is:

     System.out.println(data)
```

System.out.print is a standard output function used in java. where System specifies the package name, out specifies the class name and print is a function in that class

System.err

`System.err` is a **`PrintStream`**. `System.err` works like `System.out` except it is normally only used to output error texts. Some programs (like Eclipse) will show the output to `System.err` in red text, to make it more obvious that it is error text.

Executing Other Programs (java.lang.Runtime. `exec`)

Several forms of the **exec**( ) method allow you to name the program you want to run as well as its input parameters. The **exec**( ) method returns a Process object, which can then be used to control how your Java program interacts with this new running process.
The following example uses exec( ) to launch **notepad/calculator**.

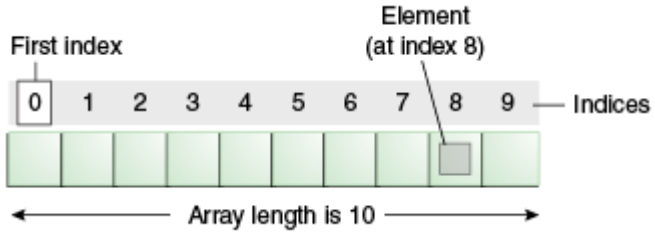
```
public class ExecDemo {

      public static void main(String[] args) {

            try {

                  Runtime r = Runtime.getRuntime( );
                  r.exec( "calc" );

            } catch ( Exception e ) {
                  System.out.println( "Error executing notepad." );
            }

      }
}
```

Java Array

Normally, array is a collection of similar type of elements that have contiguous memory location.

Java array is an object the contains elements of similar data type. It is a data structure where we store similar elements. We can store only fixed set of elements in a java array.

Array in java is index based, first element of the array is stored at 0 index.



Advantage of Java Array

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data easily.
- **Random access:** We can get any data located at any index position.

Disadvantage of Java Array

- **Size Limit:** We can store only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in java.

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in java

Syntax to Declare an Array in java

dataType[] arr; (or)
dataType []arr; (or)

dataType arr[];

Instantiation of an Array in java

arrayRefVar=**new** datatype[size];

Example of single dimensional java array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

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

int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;

//printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);

}}
```

Test it Now

```
Output: 10
        20
        70
        40
        50
```

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

```
int a[]={33,3,4,5};//declaration, instantiation and initialization
```

Let's see the simple example to print this array.

```
class Testarray1{
public static void main(String args[]){

int a[]={33,3,4,5};//declaration, instantiation and initialization
```

```
//printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);

}}
```

Test it Now

```
Output:33
       3
       4
       5
```

Passing Array to method in java

We can pass the java array to method so that we can reuse the same logic on any array.

Let's see the simple example to get minimum number of an array using method.

```
class Testarray2{
static void min(int arr[]){
int min=arr[0];
for(int i=1;i<arr.length;i++)
 if(min>arr[i])
  min=arr[i];

System.out.println(min);
}

public static void main(String args[]){

int a[]={33,3,4,5};
min(a);//passing array to method

}}
```

Test it Now

```
Output:3
```

Multidimensional array in java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in java

```
dataType[][] arrayRefVar; (or)
dataType [][]arrayRefVar; (or)
```

```
dataType arrayRefVar[][]; (or)
dataType []arrayRefVar[];
```

Example to instantiate Multidimensional Array in java

```
int[][] arr=new int[3][3];//3 row and 3 column
```

Example to initialize Multidimensional Array in java

```
arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
arr[1][0]=4;
arr[1][1]=5;
arr[1][2]=6;
arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;
```

Example of Multidimensional java array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

```
class Testarray3{
public static void main(String args[]){

//declaring and initializing 2D array
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};

//printing 2D array
for(int i=0;i<3;i++){
 for(int j=0;j<3;j++){
   System.out.print(arr[i][j]+" ");
 }
 System.out.println();
}

}}
```

Test it Now

```
Output:1 2 3
       2 4 5
       4 4 5
```

What is class name of java array?

In java, array is an object. For array object, an proxy class is created whose name can be obtained by getClass().getName() method on the object.

```
class Testarray4{
public static void main(String args[]){

int arr[]={4,4,5};

Class c=arr.getClass();
String name=c.getName();

System.out.println(name);

}}
```

Test it Now

```
Output:I
```

Copying a java array

We can copy an array to another by the arraycopy method of System class.

Syntax of arraycopy method

```
public static void arraycopy(
Object src, int srcPos,Object dest, int destPos, int length
)
```

Example of arraycopy method

```
class TestArrayCopyDemo {
   public static void main(String[] args) {
      char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
            'i', 'n', 'a', 't', 'e', 'd' };
      char[] copyTo = new char[7];

      System.arraycopy(copyFrom, 2, copyTo, 0, 7);
      System.out.println(new String(copyTo));
   }
}
```

Test it Now

```
Output:caffein
```

Addition of 2 matrices in java

Let's see a simple example that adds two matrices.

```
class Testarray5{
public static void main(String args[]){
//creating two matrices
int a[][]={{1,3,4},{3,4,5}};
int b[][]={{1,3,4},{3,4,5}};
```

```
//creating another matrix to store the sum of two matrices
int c[][]=new int[2][3];

//adding and printing addition of 2 matrices
for(int i=0;i<2;i++){
for(int j=0;j<3;j++){
c[i][j]=a[i][j]+b[i][j];
System.out.print(c[i][j]+" ");
}
System.out.println();//new line
}

}}
```

Test it Now

```
Output:2 6 8
       6 8 10
```

Arrays: (Array Sorting)

int[] a = new int[10000];

Arrays.sort(a);

Big Numbers

If the precision of the basic integer and floating-point types is not sufficient, you can turn to a couple of handy classes in the java.math package: BigInteger and BigDecimal. These are classes for manipulating numbers with an arbitrarily long sequence of digits. The BigInteger class implements arbitrary-precision integer arithmetic, and BigDecimal does the same for floating-point numbers. Use the static valueOf method to turn an ordinary number into a big number:

```
BigInteger a = BigInteger.valueOf(100);
BigInteger b = BigInteger.valueOf(200);
```

Unfortunately, you cannot use the familiar mathematical operators such as + and * to combine big numbers. Instead, you must use methods such as add and multiply in the big number classes.

```
BigInteger c = a.add(b); // c = a + b
BigInteger d = c.multiply(b.add(BigInteger.valueOf(2))); // d = c * (b + 2)
```

java.math.BigDecimal

- BigDecimal add(BigDecimal other)

- BigDecimal subtract(BigDecimal other)
- BigDecimal multiply(BigDecimal other)
- BigDecimal divide(BigDecimal other, RoundingMode mode)

Control Statements

A programming language uses control statements to cause the flow of execution to advance and branch based on changes to the state of a program. Java's program control statements can be put into the following categories: selection, iteration, and jump. Selection statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable. Iteration statements enable program execution to repeat one or more statements (that is, iteration statements form loops). Jump statements allow your program to execute in a nonlinear fashion.

Conditional Statements

The conditional statement in Java has the form

if (*condition*) *statement*

The condition must be surrounded by parentheses.In Java, as in most programming languages, you will often want to execute multiple statements when a single condition is true. In this case, use a *block statement* that takes the form

{

statement1

statement2

. . .

}

For example:

```
if (yourSales >= target)
{
        performance = "Satisfactory";
        bonus = 100;
}
```

if (*condition*) *statement1*;

else *statement2*;

```
int a, b;
//…
if(a < b) a = 0;
else b = 0;
```

Nested ifs

A *nested* `if` is an `if` statement that is the target of another `if` or `else`.

```
if (i == 10) {
      if (j < 20)
            a = b;
      if (k > 100)
            c = d; // this if is
      else
            a = c; // associated with this else
} else {
      a = d; // this else refers to if(i == 10)
}
```

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested `if`s is the *if-else-if* ladder. It looks like this:

if(*condition*)
statement;
else if(*condition*)
statement;
else if(*condition*)
statement;
...
else
statement;

The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

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

            int month = 4; // April
            String season;

            if (month == 12 || month == 1 || month == 2) {
                  season = "Winter";

            } else if (month == 3 || month == 4 || month == 5) {
                  season = "Spring";

            } else if (month == 6 || month == 7 || month == 8) {
                  season = "Summer";
```

```
            } else if (month == 9 || month == 10 || month == 11) {
                  season = "Autumn";

            } else {
                  season = "Bogus Month";
            }

            System.out.println("April is in the " + season + ".");
      }
}
```

Output is:

```
April is in the Spring.
```

Switch

The switch statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of if-else-if statements. Here is the general form of a switch statement:

A `switch` works with the `byte`, `short`, `char`, and `int` primitive data types. It also works with *enumerated types* (discussed in Enum Types), the `String` class, and a few special classes that wrap certain primitive types: `Character`, `Byte`, `Short`, and `Integer`

```
switch (expression) {
  case value1:
     // statement sequence
     break;
  case value2:
     // statement sequence
     break;
  .
  .
  .
  case valueN:
     // statement sequence
     break;
  default:
    // default statement sequence
}
```

```
        int month = 8;
        String monthString;
        switch (month) {
        case 1:
            monthString = "January";
            break;
        case 2:
            monthString = "February";
            break;
        case 3:
            monthString = "March";
            break;
        case 4:
            monthString = "April";
            break;
        case 5:
            monthString = "May";
            break;
        case 6:
            monthString = "June";
            break;
        case 7:
            monthString = "July";
            break;
        case 8:
            monthString = "August";
            break;
        case 9:
            monthString = "September";
            break;
        case 10:
            monthString = "October";
            break;
        case 11:
            monthString = "November";
            break;
        case 12:
            monthString = "December";
            break;
        default:
            monthString = "Invalid month";
            break;
        }
        System.out.println(monthString);
    }
```

Console | Javadoc | Problems | Declaration

```
<terminated> SwitchDemo [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (M
August
```

SwitchDemo.java

```
package switchex;

public class SwitchDemo {

    public static void main(String[] args) {
        int month = 2;
        int year = 2000;
        int numDays = 0;

        switch (month) {
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:
        case 12:
            numDays = 31;
            break;
        case 4:
        case 6:
        case 9:
        case 11:
            numDays = 30;
            break;
        case 2:
            if (((year % 4 == 0) && !(year % 100 == 0)) || (year % 400 == 0))
                numDays = 29;
            else
                numDays = 28;
            break;
        default:
            System.out.println("Invalid month.");
            break;
        }
        System.out.println("Number of Days = " + numDays);

    }

}
```

Console | Javadoc | Problems | Declaration

```
<terminated> SwitchDemo (1) [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (May 9, 2015, 8:15:29 PM)
Number of Days = 29
```

Using Strings in switch Statements

In Java SE 7 and later, you can use a `String` object in the `switch` statement's expression. The following code example, `StringSwitchDemo`, displays the number of the month based on the value of the `String` named `month`:

```
public class StringSwitchDemo {

    public static int getMonthNumber(String month) {

        int monthNumber = 0;

        if (month == null) {
            return monthNumber;
        }

        switch (month.toLowerCase()) {
            case "january":
                monthNumber = 1;
                break;
            case "february":
                monthNumber = 2;
                break;
            case "march":
                monthNumber = 3;
                break;
            case "april":
                monthNumber = 4;
                break;
            case "may":
                monthNumber = 5;
                break;
            case "june":
                monthNumber = 6;
                break;
            case "july":
                monthNumber = 7;
                break;
            case "august":
                monthNumber = 8;
                break;
            case "september":
                monthNumber = 9;
                break;
            case "october":
                monthNumber = 10;
                break;
            case "november":
                monthNumber = 11;
                break;
            case "december":
                monthNumber = 12;
                break;
            default:
```

```
                monthNumber = 0;
                break;
        }

        return monthNumber;
    }

    public static void main(String[] args) {

        String month = "August";

        int returnedMonthNumber =
            StringSwitchDemo.getMonthNumber(month);

        if (returnedMonthNumber == 0) {
            System.out.println("Invalid month");
        } else {
            System.out.println(returnedMonthNumber);
        }
    }
}
```

The output from this code is `8`.

```
Switch statement in Enum:
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

```
      public void tellItLikeItIs(Day day) {
            switch (day) {
            case MONDAY:
                  System.out.println("Mondays are bad.");
                  break;

            case FRIDAY:
                  System.out.println("Fridays are better.");
                  break;

            case SATURDAY:
            case SUNDAY:
                  System.out.println("Weekends are best.");
                  break;

            default:
                  System.out.println("Midweek days are so-so.");
                  break;
            }
      }
```

Iteration Statements

Java's iteration statements are **for**, **while**, and **do-while**. These statements create what we commonly call *loops*. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met.

while

The **while** loop is Java's most fundamental loop statement. It repeats a statement or
block while its controlling expression is true. Here is its general form:
while(*condition*) {
// body of loop
}

The *condition* can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true.

WhileDemo.java

```
package control;

import java.util.Scanner;

public class WhileDemo {

    public static void main(String[] args) {
        System.out.print("Enter any number: ");
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();
        while (n > 0) {
            System.out.println("tick " + n);
            n--;
        }
    }

}
```

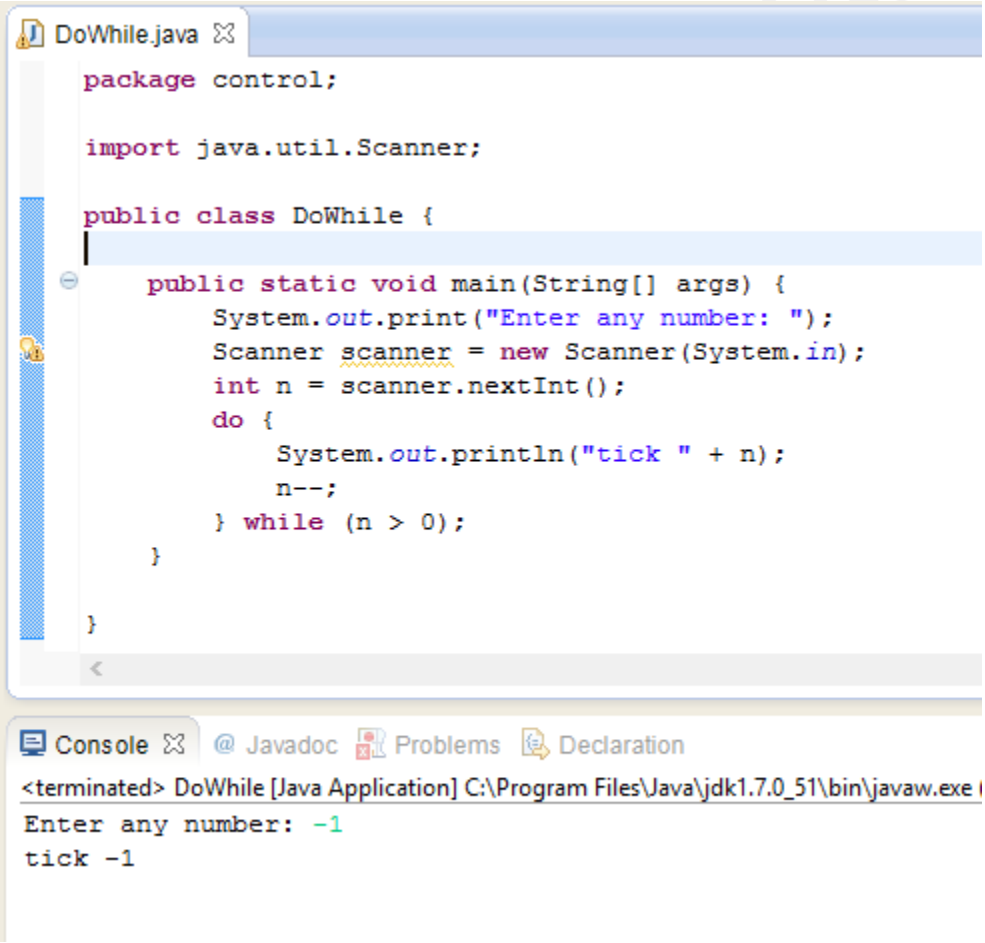
Console | Javadoc | Problems | Declaration

```
<terminated> WhileDemo [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw
Enter any number: 10
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
```

do-while

sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false to begin with. Java supplies a loop that does just that: the **do-while**. The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

do {
// body of loop
} while (*condition*);



for loop:

for(*initialization; condition; iteration*) {

```
// body
}
```
If only one statement is being repeated, there is no need for the curly braces.
```
int n;
for(n=10; n>0; n--){
        System.out.println("tick " + n);
}
```

Declaring Loop Control Variables Inside the for Loop

```
// here, n is declared inside of the for loop
for(int n=10; n>0; n--)
System.out.println("tick " + n);
}
```
When you declare a variable inside a `for` loop, there is one important point to remember: the scope of that variable ends when the `for` statement does.

Using the Comma

There will be times when you will want to include more than one statement in the initialization and iteration portions of the `for` loop.

WhileComma.java

```
package control;

public class WhileComma {
    public static void main(String[] args) {
        int a, b;
        for (a = 1, b = 4; a < b; a++, b--) {
            System.out.println("a = " + a);
            System.out.println("b = " + b);
        }
    }
}
```

Console | Javadoc | Problems | Declaration

```
<terminated> WhileComma [Java Application] C:\Program Files\Java\jdk1.7.0_5
a = 1
b = 4
a = 2
b = 3
```

In this example, the initialization portion sets the values of both `a` and `b`. The two comma-separated statements in the iteration portion are executed each time the loop repeats.

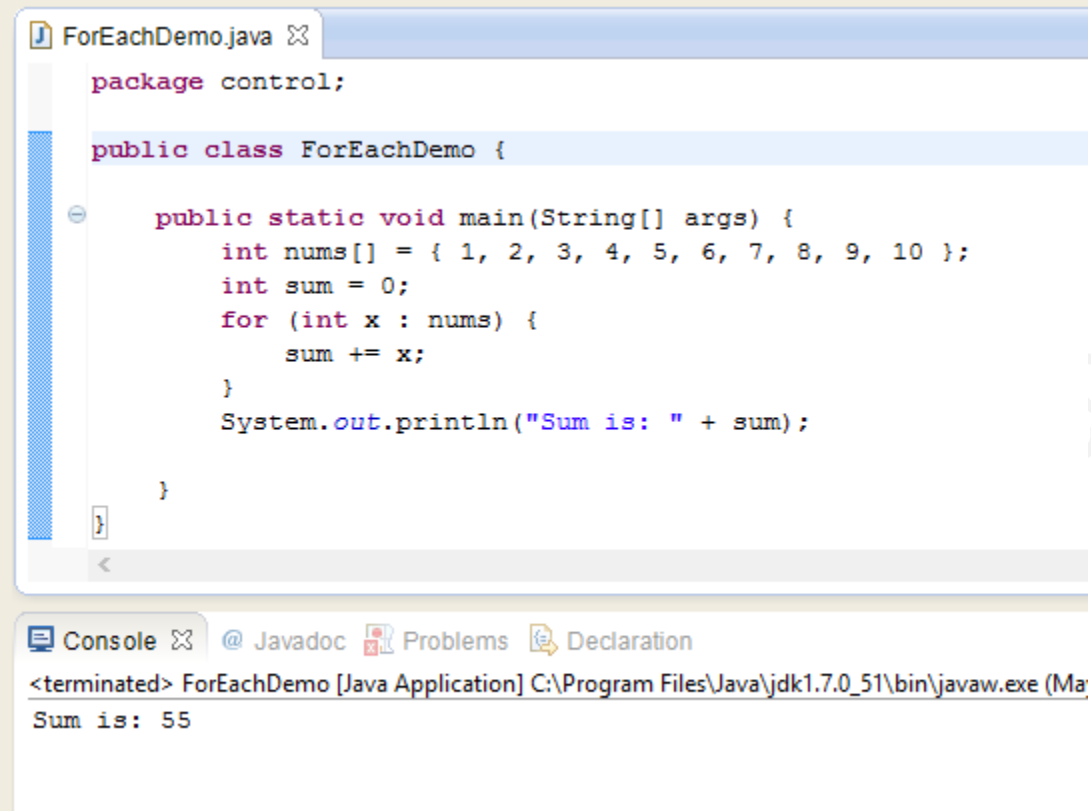
"infinite loops"
You can intentionally create an infinite loop (a loop that never terminates) if you leave all three parts of the `for` empty. For example:

```
for( ; ; ) {
// …
}
```

The For-Each Version of the for Loop

Beginning with JDK 5, a second form of `for` was defined that implements a "for-each" style loop.

for(*type itr-var: collection*) *statement-block*



Statements That Break Control Flow

Jump Statements

Java supports three jump statements: `break`, `continue`, and `return`. These statements transfer control to another part of your program.

Using break

In Java, the **break** statement has three uses. First, as you have seen, it terminates a statement sequence in a **switch** statement. Second, it can be used to exit a loop. Third, it can be used as a "civilized" form of goto.

Using break to Exit a Loop

ForEachDemo.java

```
package control;

public class ForEachDemo {

    public static void main(String[] args) {
        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        int sum = 0;
        for (int x : nums) {
            sum += x;

            if (x == 5) {
                break;
            }
        }
        System.out.println("Sum is: " + sum);

    }
}
```

Console | Javadoc | Problems | Declaration

```
<terminated> ForEachDemo [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (
Sum is: 15
```

Using break as a Form of Goto

Using break as a civilized form of goto.

The general form of the labeled `break` statement is shown here:
break *label*;

Break.java

```
package control;

public class Break {

    public static void main(String[] args) {
        boolean t = true;
        first: {

            second: {
                third: {
                    System.out.println("Before the break.");
                    if (t)
                        break second; // break out of second block
                    System.out.println("This won't execute");
                }
                System.out.println("This won't execute");
            }

            System.out.println("This is after second block.");
        }
    }
}
```

Console | Javadoc | Problems | Declaration

```
<terminated> Break [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (May 9, 2015, 9:00:57 PM)
Before the break.
This is after second block.
```

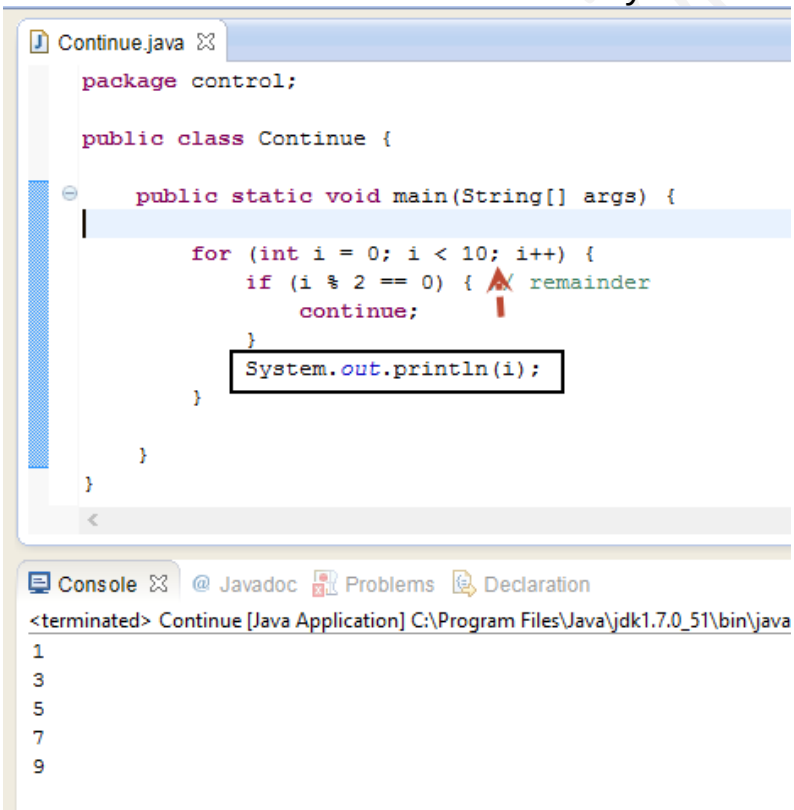
```
// Using break to exit from nested loops
public class BreakLoop4 {
      public static void main(String[] args) {

            outer: for (int i = 0; i < 3; i++) {
                  System.out.print("Pass " + i + ": ");

                  for (int j = 0; j < 100; j++) {

                        if (j == 10)
                              break outer; // exit both loops

                        System.out.print(j + " ");
                  }
                  System.out.println("This will not print");
            }

            System.out.println("Loops complete.");
      }
}
```

This program generates the following output:

```
Pass 0: 0 1 2 3 4 5 6 7 8 9 Loops complete.
```

Using continue:

Sometimes it is useful to force an early iteration of a loop.



return

The last control statement is `return`. The `return` statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.

Return.java

```
package control;

public class Return {

    public static void main(String[] args) {
        boolean t = true;
        System.out.println("Before the return.");
        if (t)
            return; // return to caller
        System.out.println("This won't execute.");
    }

}
```

Console | Javadoc | Problems | Declaration

```
<terminated> Return [Java Application] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (M
Before the return.
```

What is the difference between objects and classes:

1. An object is an instance of a class.
2. A class is the definition of an object. It does not actually become an object until it is instantiated. Since an abstract class can't be instantiated, no object of that type can created. A sub class would need to be defined in order for an object to created.

---

1. Yes, every instance of a class is an object.

2. Classes (whether abstract or not) are not objects. They are types.

```
class House { // blue print for House Objects }
```

```
class Car { // blue print for Instances of Class Car }
```

```
House myHouse = House new();
```

```
House sistersHouse = new House();
```

```
Car myCar = Car new();
String word = new String();
```

the class is the String class, which describes the object (instance) word.

When a class is declared, no memory is allocated so class is just a template.

When the object of the class is declared, memory is allocated.

```
class Cat {} // It is a cat. Just a cat. Class is a general issue. myCat =
new Cat("red", "5kg", "likes a milk", "3 years old"); // It is my cat. It is
a object. It is a really cat. yourCat = new Cat("gary", "3kg", "likes a
meal", "5 years old"); // It is your cat. Another cat. Not my cat. It is a
really cat too. It is an object; abstract class Animal {} // Abstract class
animal = new Animal(); // It is not correct. What is 'animal'? Cat, dog, cow?
I don't know. class Dog : Animal {} // It is a class. It is a dog in general.
someDog = new Dog("brown", "10 kg", "likes a cats"); // It is a really dog.
It is an object.
```