CERTIFIED ISO 9001 : 2008 NARESH i TECHNOLOGIES

CoreJava

With SCJP and

JVM Architecture

Volume-1A

COURSE MATERIAL

NARESH i technologies TM

Opp. Satyam Theatre,
Ameerpet, Hyderabad - 500 016.

E-mail:info@nareshit.com www.nareshit.com

Ph:23746666, 23734842 Cell: 9000994007, 9000994008

An ISO 9001 : 2008 Certified Company

CoreJava
with **SCJP** and
JVM Architecture

By
Mr. Hari Krishna

Shri Gajanan

Chapter 1

Introduction to Java

- In this chapter, You will learn
  - Work doing in Software Industry & Software Engineer Skills
  - Why Java is introduced?
  - Java features
  - Complete Java concepts
  - Java learning road map
  - Types of applications
  - Types of internet applications
  - Gmail Project Architecture
  - How Java achieved platform independency?
  - Java Versions and Editions
  - Java software installation and setting environment variables
  - Basic Java programming elements
  - Essential Statements of Java program
  - Java program compilation and execution procedure
  - Common Interview questions on Java programming
  - Difference between print and println methods
  - SUN Microsystems's Promise
  - Coding standards and naming conventions
  - Java platform Architecture

- By the end of this chapter- you will understand complete Java architecture, Java projects design and development architectures and also aware of basic rules in developing, compiling and executing Java programs with Compiler and JVM activities.

Sagar

Interview Questions

By the end of this chapter you answer all below interview questions

On Software Industry

- What are you doing in Software Industry?
- What is the difference between project and product?
- Types of companies bases on projects and products
- When do we call a project is maintenance project?
- Types of teams in software companies
- What are the words used in project development and maintenance.
- Types of logics developed in projects?
- Software Engineer Skills
- Give project architecture to show each technology role in project development
- Interview process and types of questions asked in interview
- Importance of doing Java certification SCJP/OCJP and learning JVM architecture
- Complete Java as per SUN and as per Software industry
- Java Concepts and technologies to develop projects
- Java and its technologies Learning Roadmap
- A short story on Android, Java SE Tree diagram.

On Java Language, Java software and Java Programming

- Why JAVA language is developed?
- Definition of Java, and who invented Java?
- Java features
- Java history
- Introduction to OAK and why OAK renamed to JAVA.
- What is the abbreviation of Java?
- Common terminology used in programming languages
- What is a platform, PD and PID?
- What is a stand-alone and internet application?
- Types of Internet applications
- What are the technologies invented by SUN to develop internet applications?
- Gmail server architecture to understand types of internet applications
- Common logics developed in projects for automatic business
- What is a design pattern?
- Different types of design patterns used in project design and develop for automating business.
- M-V-C Architectures and their advantages
- Why MVC architecture name is MVC, why not CVM or VCM?
- Importance of CoreJava & Where CoreJava concepts used in above architectures
- What is the main feature of a programming language to develop internet application?
- Why C, C++ do not support platform independency?
- How Java achieved Platform independency?

CERTIFIED ISO 9001 : 2008 NARESH TECHNOLOGIES

CoreJava
With SCJP and
JVM Architecture

Volume-1A

COURSE MATERIAL

NARESH technologies TM
SOFTWARE TRAINING & DEVELOPMENT
Opp. Satyam Theatre,
Ameerpet, Hyderabad - 500 016.
E-mail:info@nareshit.com www.nareshit.com
Ph:23746666, 23734842 Cell: 9000994007, 9000994008
An ISO 9001 : 2008 Certified Company

CoreJava
with **SCJP** and
JVM Architecture

By
Mr. Hari Krishna

Index

Volume -1: Java Language and OOPS

Chapter #01: Introduction to Java & OOPS
Chapter #02: Comments, Identifiers, Keywords
Chapter #03: Working with EditPlus Software
Chapter #04: DataTypes & Literals
Chapter #05: Wrapper Classes with Autoboxing & unboxing
Chapter #06: Exception Handling
Chapter #07: Packages 177
Chapter #08: Accessibility Modifiers
Chapter #09: Methods and Types of methods
Chapter #10: Variables and Types of Variables
Chapter #11: JVM Architecture
Chapter #12: Static Members & their control flow
Chapter #13: Non-Static Members & their control flow
Chapter #14: Final Variables and their rules
Chapter #15: Classes and Types of classes
Chapter #16: Inner classes
Chapter #17: Design Patterns
Chapter #18: OOPS Fundamentals and Principles
Chapter #19: Types of objects & Garbage Collection
Chapter #20: Arrays and *Var-arg* types
Chapter #21: Working with jar
Chapter #22: Operators
Chapter #23: Control Statements

Volume -2: Java API and Project

Chapter #24: API & API Documentation
Chapter #25: Fundamental Classes – Object, Class
Chapter #26: Multithreading with JVM Architecture
Chapter #27: String Handling
Chapter #28: IO Streams (File IO)
Chapter #29: Networking (Socket Programming)
Chapter #30: Collections and Generics
Chapter #31: Regular Expressions
Chapter #32: Reflection API
Chapter #33: Annotations
Chapter #34: AWT, Swings, Applet
Chapter #35: Formatting text and date (java.text package)

Shri Gajanan

Chapter 1

Introduction to Java

- In this chapter, You will learn
  - Work doing in Software Industry & Software Engineer Skills
  - Why Java is introduced?
  - Java features
  - Complete Java concepts
  - Java learning road map
  - Types of applications
  - Types of internet applications
  - Gmail Project Architecture
  - How Java achieved platform independency?
  - Java Versions and Editions
  - Java software installation and setting environment variables
  - Basic Java programming elements
  - Essential Statements of Java program
  - Java program compilation and execution procedure
  - Common Interview questions on Java programming
  - Difference between print and println methods
  - SUN Microsystems's Promise
  - Coding standards and naming conventions
  - Java platform Architecture

- By the end of this chapter- you will understand complete Java architecture, Java projects design and development architectures and also aware of basic rules in developing, compiling and executing Java programs with Compiler and JVM activities.

Sagar

Interview Questions

By the end of this chapter you answer all below interview questions

On Software Industry

- What are you doing in Software Industry?
- What is the difference between project and product?
- Types of companies bases on projects and products
- When do we call a project is maintenance project?
- Types of teams in software companies
- What are the words used in project development and maintenance.
- Types of logics developed in projects?
- Software Engineer Skills
- Give project architecture to show each technology role in project development
- Interview process and types of questions asked in interview
- Importance of doing Java certification SCJP/OCJP and learning JVM architecture
- Complete Java as per SUN and as per Software industry
- Java Concepts and technologies to develop projects
- Java and its technologies Learning Roadmap
- A short story on Android, Java SE Tree diagram.

On Java Language, Java software and Java Programming

- Why JAVA language is developed?
- Definition of Java, and who invented Java?
- Java features
- Java history
- Introduction to OAK and why OAK renamed to JAVA.
- What is the abbreviation of Java?
- Common terminology used in programming languages
- What is a platform, PD and PID?
- What is a stand-alone and internet application?
- Types of Internet applications
- What are the technologies invented by SUN to develop internet applications?
- Gmail server architecture to understand types of internet applications
- Common logics developed in projects for automatic business
- What is a design pattern?
- Different types of design patterns used in project design and develop for automating business.
- M-V-C Architectures and their advantages
- Why MVC architecture name is MVC, why not CVM or VCM?
- Importance of CoreJava & Where CoreJava concepts used in above architectures
- What is the main feature of a programming language to develop internet application?
- Why C, C++ do not support platform independency?
- How Java achieved Platform independency?

- Java's Slogan
- Types of Java Softwares
- What is the difference between JDK, JRE and JVM?
- Different environments existed in real time projects development
- What is Java Plug-in?
- JAVA Versions and Editions
- Java Software (JDK and JRE) installation and its folder Hierarchy
- Which edition features are installed if we install JDK/JRE?
- Definition of JDK and JRE *or* What is JDK and JRE?
- What is environment variable?
- Understanding the need of environment variables
- Setting Java environment using *JAVA_HOME, Path* and *Classpath* environment variables
- Types of applications developed using JAVA SE and EE.
- How can we connect Java and .Net applications in a single project?
- Basic JAVA programming elements for developing above projects architectures
- What is the difference between *class* and *interface*?
- Why *enum*?
- Essential statements of a Java Application
- Simple java program development, Compilation and execution.
- When Java source file name and class name must be same.
- Java Programming interview questions
- Why JVM executes only main method why not user defined methods?
- Can we define main method in all classes of a single Java file?
- Can we call main method, if so what is the syntax to call main method?
- What does happen if we pass –ve number in main method call?
- Difference between print and println methods.
- SUN Microsystems's Promise
- Coding standards and naming conventions
- Java platform architecture
- What Can Java Technology Do?
- How Will Java Technology Change My Life?
- Why pointers are eliminated from Java?
- What is JIT compiler?
- Differences between C++ and Java

Introduction to Java

Talk about Java technology seems to be everywhere, but what exactly is it? The following sections explain how Java technology is both a programming language and a platform, and provide an overview of what this technology can do for you.

Why Java?

Java programming language mainly designed to develop *internet applications* by providing platform independency. C, C++ programming languages supports developing only stand-alone application, it can only be executed in current system, cannot be executed from remote system via network call.

Definition of Java

Java is a very simple, high-level, secured, multithreaded, platform independent, object-oriented programming language. It was developed by James Gosling in SUN Microsystems in 1990's for developing internet applications. Its first version is released in January 23, 1996.

It is descendent of C, C++ programming languages. Its syntax is similar to C and C++, but it omits many of the features that make C and C++ complex, confusing and unsafe.

So, in Java we cannot see the most *horrible topics* such as pointers, structure, union, operator overloading, multiple inheritance and many more.

Java Features

To support internet application development Java programming has below features
The Java programming language can be characterized by all of the following buzzwords:

- Simple
- Secure
- Robust
- Portable
- Architecture neutral
- Object oriented
- Multithreaded
- High performance
- Distributed
- Dynamic

Java Definition based on its features

Java is just a simple, secure, robust, portable, object-oriented, interpreted, byte coded, architectural-neutral (platform Independent), garbage collected, multithreaded programming language with strongly typed exception handling mechanisms for writing distributed, dynamically extensible programs.

What is the abbreviation of Java?

There is **no abbreviation** for Java. The Development Team of Java has just chosen this name. The name Java specifically doesn't have any meaning rather **it refers to the hot, aromatic drink COFFEE**. This is the reason Java programming language icon is coffee cup.

Who developed Java?
James Gosling, PhD (born May 19, 1955 near Calgary, Alberta, Canada) along with other engineer scientists discovered Java at SUN Microsystems. He is a famous software developer, best known as the father of the Java programming language.

Terminology used in programming languages

- **Source code**
  Developer written program; it is written according to the programming language syntax.

- **Compiled code**
  Compiler generated program that is converted from source code
  - **Compiler**
    It is a translation program that converts source code into machine language at once
  - **Interpreter**
    It is also a translation program that converts source code into machine language but line by line.

- **Executable code**
  OS understandable readily executable program(.exe files)

- **Compilation**
  It is a process of translating source code into compiled code.

- **Execution**
  It is a process of running compiled code to get output.

Java technology
Unlike other high level programming languages, Java technology is both platform and programming language. Platform is a hardware or software environment in which programs are executed. Java has its own *software based platform* called JVM – Java Virtual Machine – to execute Java programs. Like C or C++ programs, Java programs are not directly executed by OS.

What is a platform?
A *platform* is a hardware or software environment in which a program runs.
For instance, computer platform is (Operating System + Hardware Devices).

What is meant by platform dependent and platform independent application?
Platform Dependent
An application that is compiled in one operating system, if it is not run in different operating system then that application is called platform dependent application. The programming language that is used to develop this application is called platform dependent programming language. C, C++ programming languages are platform dependent programming languages, because these languages program compiled code does not run in different OS. Detailed explanation on this point is given next pages.

Platform Independent

If the application's compiled code is able to run in different operating system then that application is called platform independent application. The programming language that is used to develop this application is called platform independent programming language. Java is platform independent programming language, because Java program compiled code can run in all Operating Systems. Detailed explanation on how Java achieved platform independency is given in next pages.

Java History

- The Java platform was initially developed to address the problems of building software for networked consumer electronic devices. It was designed to support multiple host architectures and to allow secure delivery of software components.
- To meet these requirements, compiled code had to survive transport across networks, operate on any client OS, and assure the client that it was safe to run.
- Before Java Language, the software for these consumer electronic devices such as washing machines, microwave ovens and micro controllers was developed by C, C++ .
- These languages are platform dependent.
- So we require a platform independent language that could be used to create software to be embedded in various consumer electronic devices.
- In an attempt to find such a solution, in SUN Microsystems a team headed with a scientist James Gosling began work on portable, platform independent language that could be used to produce code that would support any technology at any time.
- Initially they named that language as OAK, it's a tree name.
- It was all happened in the middle of 1991 and the late 1992.
- In the mean time of enhancing this OAK language to make more efficient language
- The World Wide Web was emerging into the market.
- In www so many verities of CPUs under different environments will be connected.
- So here also we require a portable and platform independent language.
- The only solution for this problem is OAK.
- So this OAK language was modified or enhanced to fulfill the requirements of internet programming and was renamed to "JAVA".
- It was happened in 1995.

Why the name OAK renamed to Java?

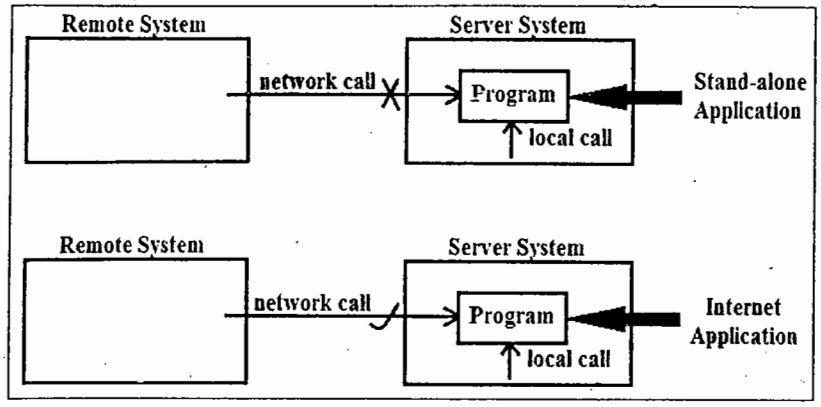
They were unable to register this programming language with name OAK, because already some other product is registered with the same name. So they renamed to Java.

Types of applications

Based the way of execution of programs, all available applications are divided into 2 types

1. **Stand-alone applications**
   An application that can only be executed in local system with local call is called stand-alone application
2. **Internet applications**
   An application that can be executed in local system with local call and also from remote computer via network call (request) is called internet application

Below diagram shows **Stand-alone** and **Internet** applications



Types of internet applications

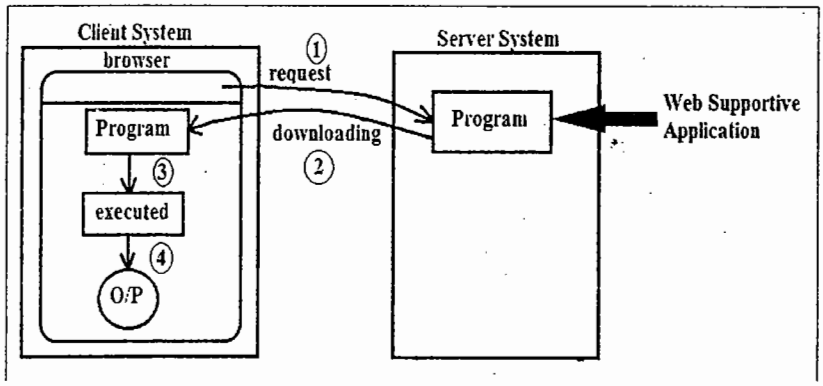
We have two types of internet applications

1. Web Supportive Application - executed in client computer
2. Web Application - executed in server computer

Web Supportive Application

An application that resides in server system and that is downloaded and executed in client computer via network call is called web supportive application.

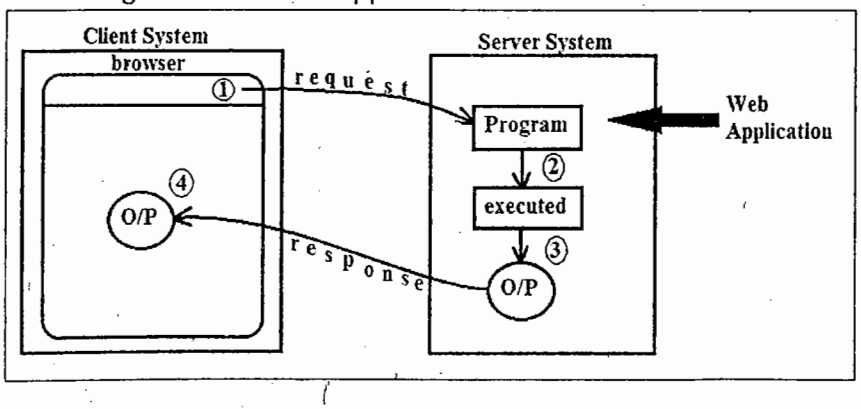
Below diagram shows its execution control flow



Web Application

An application that resides in server system and that is executed directly in server system via network call and sending response (output) back to client is called web application.

Below diagram shows web application execution control flow



What are the technologies invented by SUN to develop above two types of applications?

Applets - to develop web supportive applications
Servlet and **JSP** - to develop web applications

Initially HTML is invented to support developing web supportive applications. We can say in Java, Applet is the replacement of HTML. So using Applet we develop GUI to take input from end user and to generate Reports to show output to end user.

Conclusion:

- Applet is executed in client system browser
- Servlet is executed in server system.

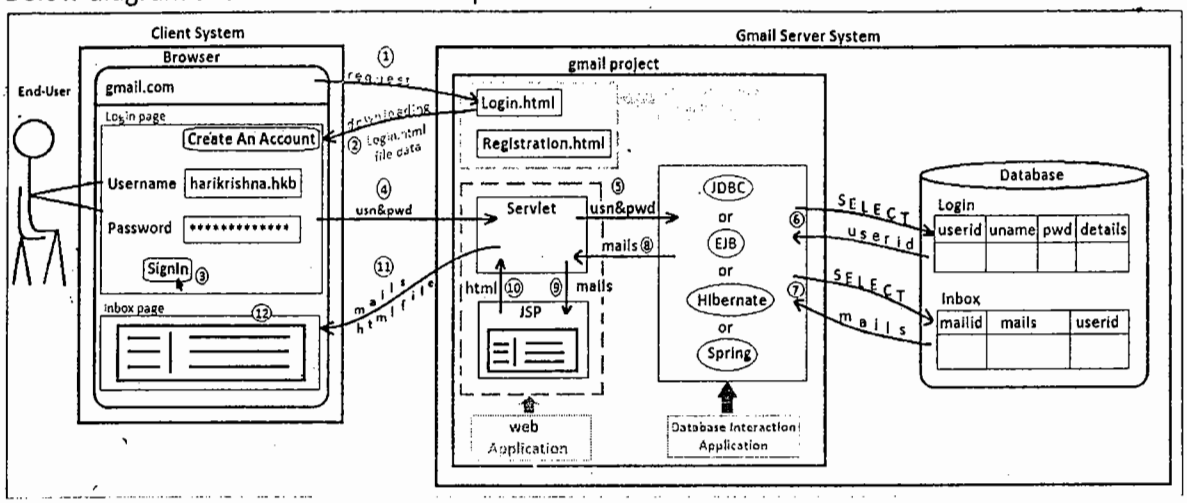
Let us understand above application with real time project Gmail.

Gmail server architecture to understand above two applications

Let us try to understand the operations you are doing to check your mails in gmail account

1. First you will open browser, then you enter gmail server address "gmail.com" in the address bar.
2. Soon you press "Enter" key on keyboard; you will observe gmail login page is appeared. Is that logic page HTML code available in your computer? Or is it downloaded from gmail server? It is downloaded from gmail server then your browser renders that html code and displays it to you. Now tell me HTML technology develops what type of application? Web supportive Application.
3. Now, as per next step you will enter your gmail account username and password, and you will click *SignIn* button. If the given username and password is correct you will get your inbox mails, else you will get an error page saying that username or password is incorrect.
4. Think what happened in background? Browser sends the given username and password to gmail server. Gmail server reads these values and passes those values to an application running at server side. It is nothing but Servlet, means Web Application.
5. Then servlet passes those values to Database Interaction Application, JDBC program.
6. Now, JDBC Program executes SELECT query on DB LOGIN table to check whether user exists with given values or not. If user exists DB returns userid.
   The query would be
   SELECT userid FROM login WHERE uname=givenUsername AND pwd=givenPassowrd;
7. Using this returned userid, JDBC Program executes another SELECT query on INBOX table to get all mails related to this user. DB sends all mails related to this user to JDBC application. The query would be
   SELECT mails FROM inbox WHERE userid=returnedUserid;
8. Then JDBC application sends those mails to Servlet.
9. Now the Servlet sends those mails to JSP to format the mails with help of HTML hyperlinks.
10. Using these mails, JSP generates HTML code dynamically and sends that HTML code to Servlet.
11. Servlet sends those mails which are formatted in HTML code to the browser
12. Finally, the browser displays those mails to user.

Below diagram shows all above 12 steps execution



Conclusions from the above diagram

1. Every real world web project contains below three applications
   1. Web supportive application
   2. Web application
   3. Database interaction application

2. In Java we develop above applications by using
   - **HTML/Applet** - to develop web supportive application.
   - **Servlet and JSP**- to develop web application.
   - **JDBC and EJB** to develop Database Interaction Application.

3. In .NET we develop above applications by using
   - **HTML** - to develop web supportive
   - **ASP.NET** - to web applications
   - **ADO.NET** - to develop Database Interaction Application.

4. Web supportive application (HTML/Applet) is used
   - for collecting input from end-user by providing GUI then to pass it to web application
   - for displaying output to end-user by providing *Report*

5. Web application (Servlet and JSP technologies) is used for
   1. *Processing request* and
   2. *Preparing response.*

   - Servlet role is processing request means
     - Reading input from network that is sent by end-user via HTML forms, then
     - Executing Business Logic by using this input then
     - Finally generating output and sending it to JSP

   - JSP role is preparing response means
     a. Generating dynamic HTML by using the output given by Servlet
     b. Sending the result HTML back to Servlet

   - Then servlet sends this response to browser to display this output to end-user.

6. Database Interaction Application (JDBC/EJB technologies) is used for performing CRUD operations on DB.

DB Terminology	
C - Create	*Insert*
R - Read	*Select*
U - Update	*Update*
D - Delete	*Delete*

Software Engineer Skills:

As a being a software engineer you must work in developing all above three types of applications. So you must acquire all below skills

1. Programming Language - ***Java / .NET***
2. GUI Preparation Language - ***HTML, CSS***
3. Scripting Language - ***Java Script, VB Script***
4. Database - ***Oracle, SQL Servler***
5. Server Operating System - ***Unix commands and Shell scripting***
6. Problem Solving Skills - ***CRT*** and ***C with DS***
7. Presentation Skills - ***MS Office***
8. Communication Skills - ***English***

Java concepts and technologies to develop all above three applications in projects

SUN Microsystems divided Java concepts into three categories to support all three types of domains *mobile*, *desktop* and *internet* applications. In Java a category is called as Edition. So we can say Java concepts are divided in to 3 editions

1. Java ME (Micro Edition)
2. Java SE (Standard Edition)
3. Java EE (Enterprise Edition)

Java ME concepts are used for developing ***Mobile Applications***
Java SE concepts are used for developing ***Desktop and Window based applications***
Java EE concepts are used for developing ***Internet and Enterprise applications***

We are here to learn only **Java SE** *and* ***Java EE***

Complete Java:

As a Java learner we must know one important point
i.e.; we have two divisions of Complete Java

1. Complete Java as per SUN
2. Complete Java as per Software industry

SUN Complete Java

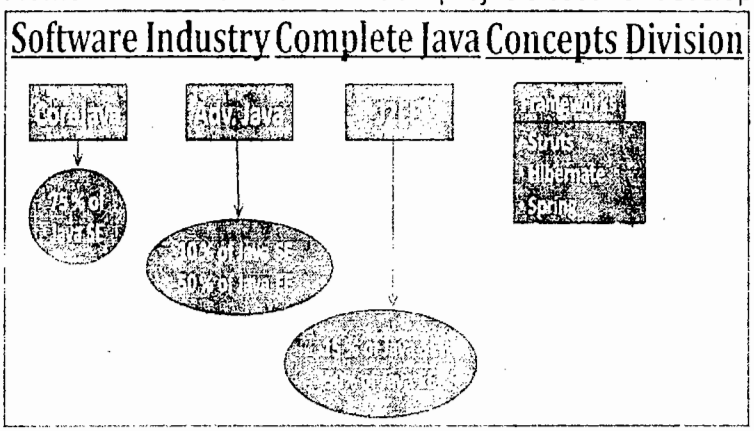
- Java SE
- Java EE

Software Industry Complete Java

To learn Java quickly and easily we divide Java concepts into 4 divisions

1. CoreJava
2. Advanced Java
3. J2EE
4. Frameworks

Frameworks are not part of Java SE or Java EE they are developed by third party companies by using Java EE technologies. Framework softwares are not developed by SUN/Oracle. Framework softwares are reusable projects used for developing new projects.



Naresh *i* Technologies Complete Java

1. CoreJava with *SCJP* and *JVM Architecture*
2. Advanced Java with SCWCD
3. J2EE
4. XML, WebServices
5. JSF
6. Frameworks
   a. Struts
   b. Spring with Hibernate
7. HTML, CSS, Java Script
8. Ajax, jQuery, HTML5, CSS3
9. Java Project Development Supporting tools
10. Academic and Live projects

Batches List

Java SE and Java EE concepts

Java is both language and technology.

- **Java SE has both**
  - *Java language concepts* and
  - *Technologies*
- **Java EE has only**
  - *Technologies*

Java SE Language Concepts

1. Datatypes, Operators, Control Statements
2. OOPS
3. JVM Architecture
4. Garbage Collection
5. String Handling
6. Wrapper Classes
7. Exception Handling
8. Multithreading
9. IOStreams (File IO)
10. Networking (Socket Programming)
11. Collections Framework
12. Regular Expressions
13. Reflection API
14. Annotations
15. Inner Classes
16. AWT, Swings, Applet
17. Working with Jar
18. Date, time and text format

Core Java

Java SE technologies	Java EE technologies
1. JDBC	1. Servlet
2. XML	2. JSP
3. RMI	3. EJB
4. JNDI	4. WebServices
5. Connection Pooling	5. JSF

Adv Java

J2EE

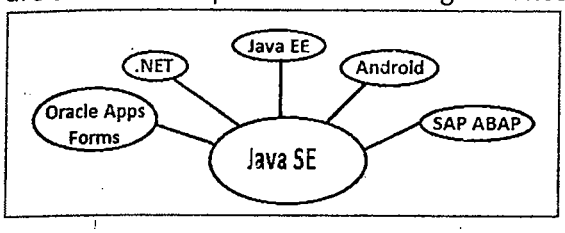
A short story on Android

Android is an open-source software stack for mobile devices which has Operating System, middleware and some key applications. "*Android*" invented by *Open Handset Alliance* (a group of companies), and is bought by *Google* in 2005 to develop their mobile "*Arsenal*"

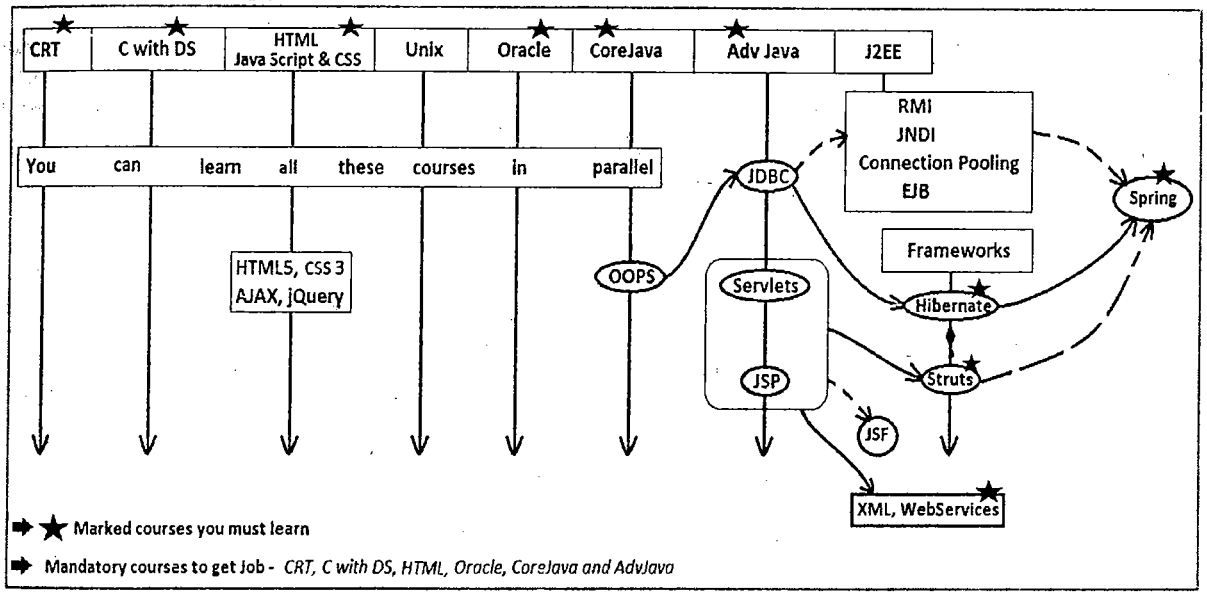
Android is developed by using Java SE concepts, not by using Java ME.

Java SE Tree Diagram

Java SE is the root of all software industry technologies including Testing tool (Selenium), SAP/ABAP, Oracle Application forms. You can easily work on any domain based projects if you are a Java developer as all technologies concepts are similar to Java.



Java course learning path



What is the main feature of a programming language to develop internet application?

Platform Independency because an internet application must run in all Operating Systems irrespective of where it is compiled.

Why C, C++ programming languages cannot support internet application development?

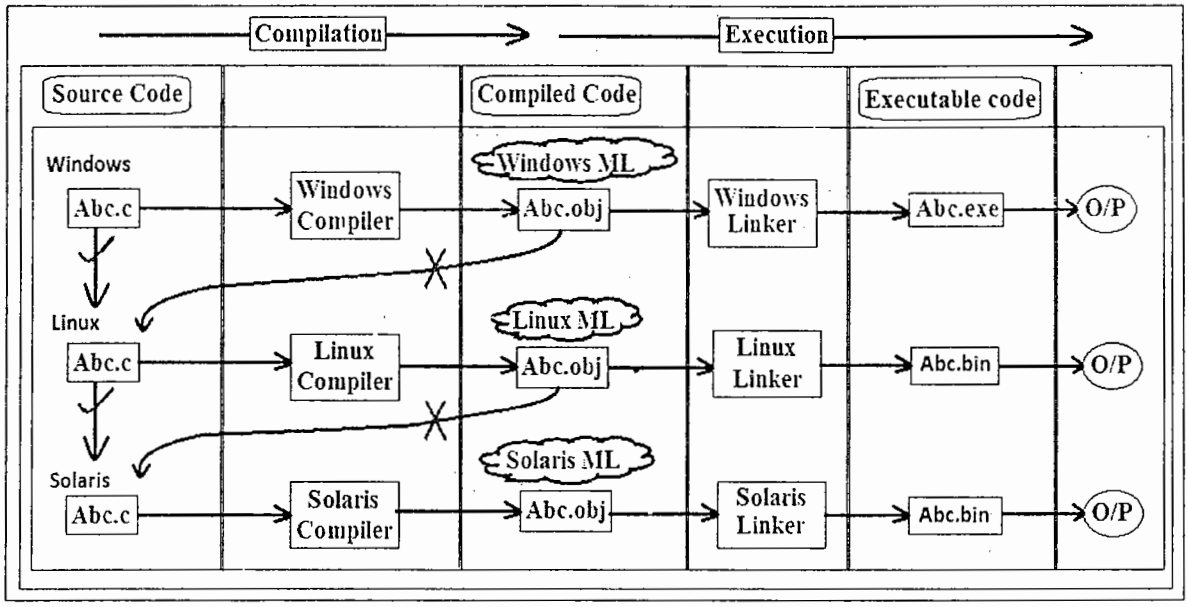
Because they are platform dependent.

Why C, C++ programming languages were developed as platform dependent, why not as platform independent?

By the time C / C++ programming languages were developed there was no internet. Platform independency feature is only required for developing internet applications.

Why C, C++ program compiled code cannot execute in all operating systems?
In these programming languages the compiled code is Machine Language which is understandable only by the current OS. Thus this compiled code cannot be executed in different Operating System. Due to this reason these languages are considered as platform dependent programming languages.

Below diagram shows platform dependency of C and C++.



Concludes

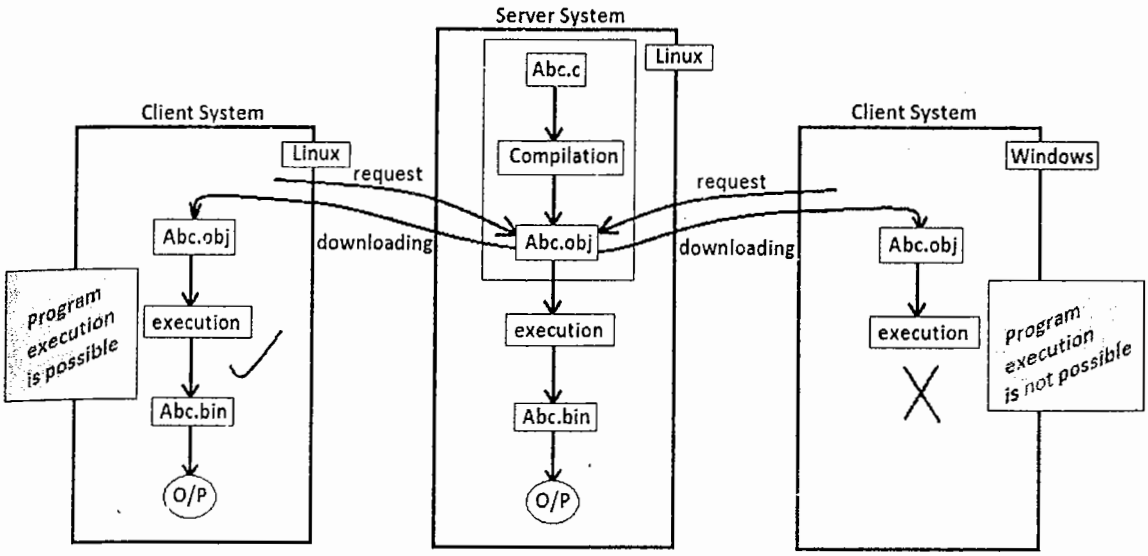
1. C, *C++ compiler is OS dependent*: As you noticed, in the above diagram compiler generates OS specific Machine Language.

2. *C, C++ softwares are also platform dependent*: because compiler is a part of C, C++ software. Hence C, C++ vendors develop compiler specific to OS, and maintains C, C++ softwares separately one per every OS.

3. So to develop C, C++ programs, we must install C or C++ softwares specific to OS.

Is it true that C, C++ cannot support internet application?
Partially YES, they can support internet application development. If client OS is same as Server OS then the C program is downloaded and executed, else it cannot be executed.

But still C, C++ programming languages are not a good choice for developing internet applications, because we cannot always guarantee that the client OS is the same as server OS, also we cannot force the client to have OS same as the server OS. If we force we will lose business. Below diagram shows, C, C++ can support internet applications development partially.



. In above diagram, C program does not run in windows because it was compiled for Linux.

What is the solution for achieving platform independency?
Very simple, *generate ML for client OS*. So to generate ML for client OS, ML generation must be moved from compilation phase to execution phase, because client OS is known only at the time of execution.

Since we moved ML generation from compilation phase to execution phase, what type of code should compiler generate, and after downloading how that code can be translated to Machine language of that client current OS?
In this approach compiler should generate some code that should not be understandable by human being let us say "encrypted code" and should have another translator software to convert this encrypted code into machine language of the current client OS.

Java's implementation to achieve platform independency
In Java, as a solution of above two problems, James Gosling and his team invented a new format code called **bytecode** which is generated by compiler and translator software called **JVM** to translate it into machine language of the current client OS.

- Bytecodes is an intermediate format that can only be understandable by JVM.
- JVM -Java Virtual Machine – is a software that executes Java bytecodes by converting bytecodes into machine language of the current operating system's understandable format.

How did Java Achieve Platform Independency to support internet application development?
Java achieved platform independency by moving Machine language generation from compilation phase to execution phase by introducing bytecodes and JVM – bytecode is a native language of JVM and JVM is translated software that converts bytecodes into current client OS understandable machine language for executing java bytecodes.

What are the Java file's extensions?

In java we have only two extension files

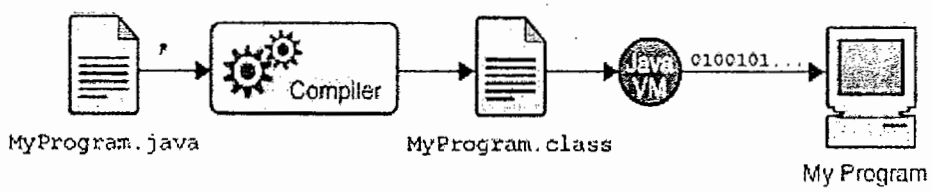
- "*.java*" extension file it called source code which is developed by developer
- "*.class*" extension file it is called compiled code contains bytecode which is generated by compiler from source code.

Like in C or C++, in Java we do not have executable file means ".exe" extension files. More over In Java, the compiled code is not machine language; instead it is an intermediate code, named as "*bytecodes*" stored in a new file with an extension ".class". This code is generated by Java compiler and is only understandable by JVM.

Java program development, compilation and execution procedure

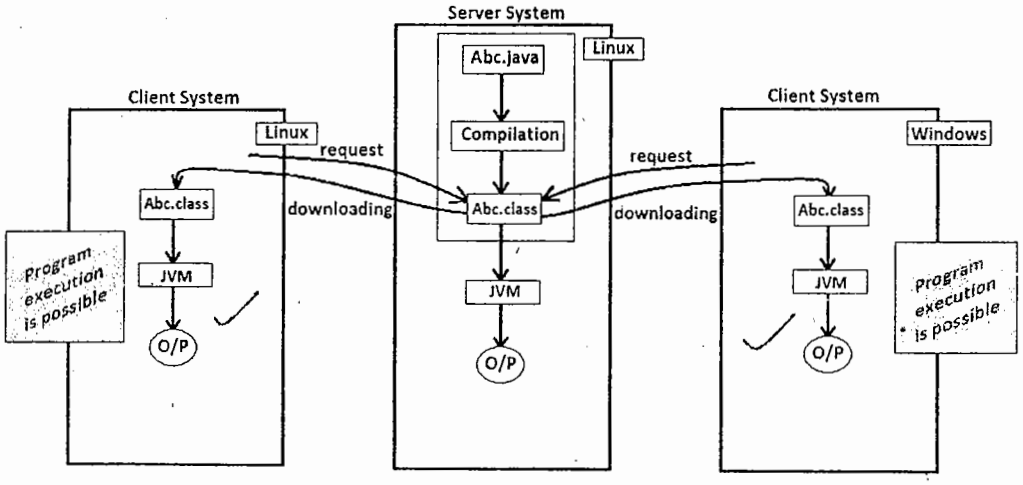
In the Java programming language, all source code is first written in plain text files ending with the "***.java***" extension. Those source files are then compiled into "***.class***" files by the ***javac*** compiler. A ".class" file does not contain code that is native to your processor; it instead contains *bytecodes* — the native language of the Java Virtual Machine. The ***java*** launcher tool then runs your application with an instance of the Java Virtual Machine.

Below diagram shows Java program compilation and execution (*copied from Oracle tutorial*)



Java program execution process as internet application

User downloads ".class" file from server system and executes with the JVM installed in that client computer as shown the below diagram.

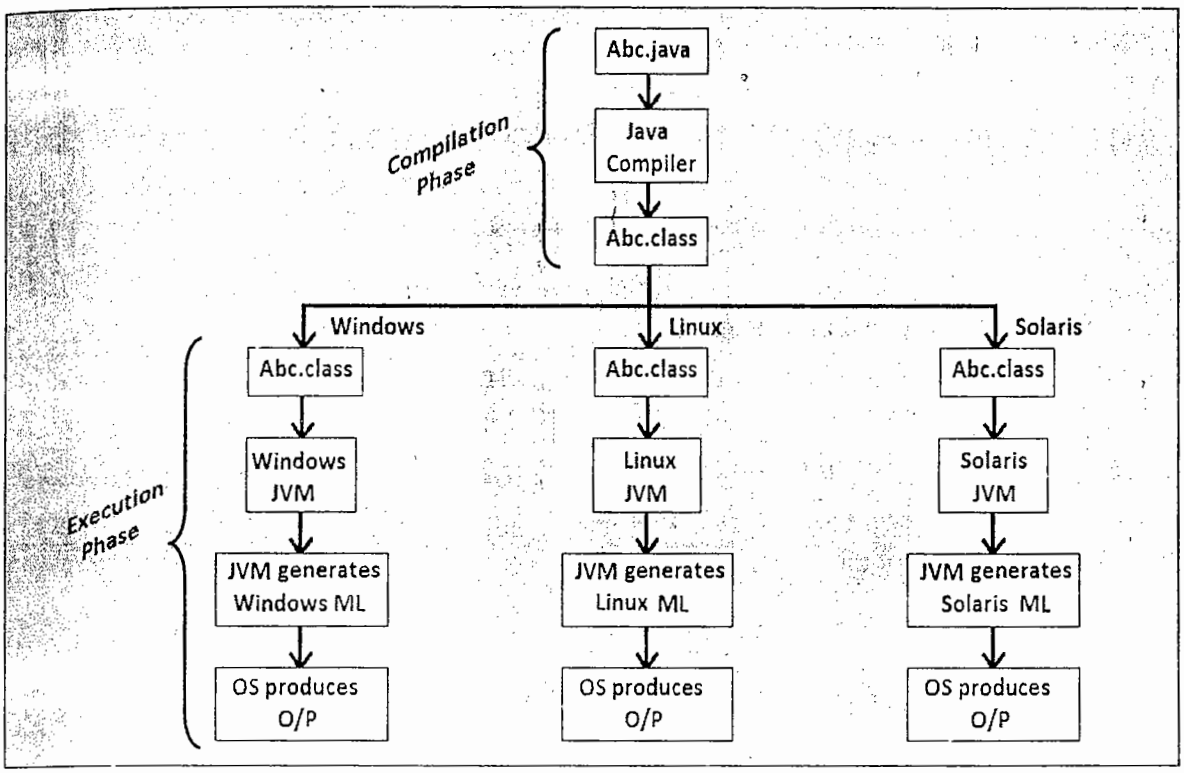


Because the Java VM is available on many different operating systems, the same .class files are capable of running on Microsoft Windows, Linux, Solaris, or Mac OS.

Java's Slogan

"Write Once, Run Anywhere (WORA)"

Below diagram shows Java's Platform Independency block diagram, it also shows implementing above slogan



Conclusions

1. Compiler is responsible to convert Java source code into bytecode and storing these bytecodes in a separate file with extension ".class".
2. JVM is responsible to execute these bytecodes. JVM executes those bytecodes by converting them into the ML of current OS.
3. In Java we do not have a file with machine language; also in Java we do not have executable files (.exe files). We have only 2 files *.java* and *.class*
4. *Java class bytecodes run in all OS,* irrespective of the OS in which the Java program is compiled. Because the Java VM is available on many different operating systems.
5. *Java compiler is OS independent*, because it takes Java source code and generates Java bytecode, both input and output files are Java related files.
6. *JVM is OS dependent,* because it takes Java bytecodes and generates OS dependent ML.
7. *So Java software is platform dependent*, Since JVM platform dependent we must have different Java software for every OS separately. Hence Java software is OS dependent.

Now answer my question, is Java Platform Dependent or Independent?

Java program is platform independent but Java software is platform dependent, because JVM is platform dependent and is available for every OS separately.

Java software becomes platform dependent and Java program becomes platform independent both because of JVM implementation.

Who develop JVM, OS vendor or SUN?

JVM is developed by SUN separately for every OS, not by OS vendor.

Is JVM available for all Operating Systems?
Yes, JVM is available for every OS separately.

Is JVM installed automatically with OS?
No, it is not installed with OS. We must install it in our computer explicitly.

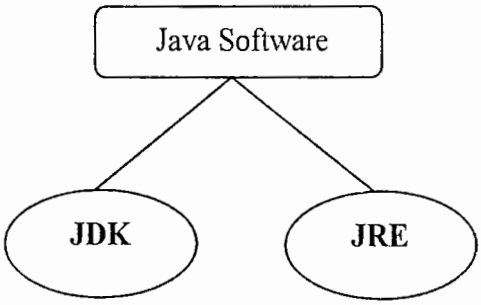
What are the Developer and client responsibilities?

- Developer is responsible of developing, compiling and executing the Java application, so in developer system we must install both Compiler & JVM.
- Whereas client is responsible of only executing that Java application, hence in client computer just installing JVM is sufficient.

Based on the above points we can say that Java software is divided into 2 parts as below

Types of Java Softwares
Java Software is divided into two types

1. JDK – Java Development Kit
2. JRE – Java Runtime Environment

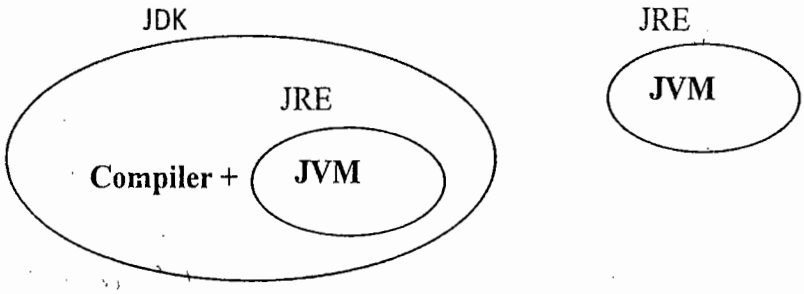


JDK and JRE are called principle products of Java software.

Difference between JDK, JRE and JVM
JVM is a subset of JRE, and JRE is the subset of JDK. When we install JDK, JRE is also installed automatically. JRE software is available as a separate pack, so we can install JRE alone.

- JDK has both compiler and JVM. So using JDK we can develop, compile and execute new java applications and also we can modify, compile and execute already developed applications.
- JRE has only JVM. Hence using JRE we can only execute already developed applications.

Diagram



Different environments existed in real time projects development

1. *Development environment* – Here developers will work to develop new programs (class). Hence we should install JDK in development environment.
2. *Testing environment* – Here testes will work to test the project, means they just execute the project. Hence JRE installation is enough.
3. *Production environment* – Here end-users will work to use the project, means they just execute the project to complete their transactions. Hence JRE installation is enough.

Java Versions and Editions

Java Versions

A version is a number that is used to identify the current features and enhancements developed in that software's current release. Every software is developed with versions.

Also in software companies all projects and products are developed with versions. In most of the cases the version number is starts with "1.0".

- A project is a software that is developed specific to one customer.
- A product is also a software but is developed common for all companies, we can say it is a readymade software.

Below are the sample products with their version numbers.

Java Software	Windows Software	Oracle software
Java 1.0	Windows 98	Oracle 8i
Java 1.1	Windows 2000	Oracle 9i
Java 2 platform	Windows 2003	Oracle 10g
|-> 1.2	Windows XP	Oracle XE
|-> 1.3	Windows Vista	Oracle 11g
|-> 1.4	Windows 7	
Java 5.0	Windows 8	
Java 6.0		
Java 7.0		
Java 8.0		

Major and Minor versions

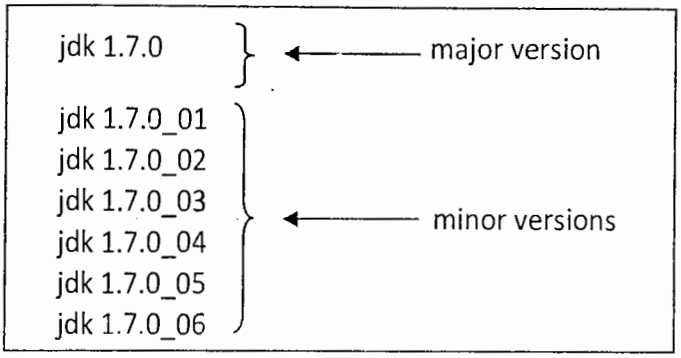
Java has two types of versions

1. Major version – it is the main version of the software contains new features & enhancements of current release of that software
2. Minor version – it is the sub version of the major version contain bug fixes

- Java software is released with major version, if there are any new features or enhancements are added.
- Java software is released with minor version, if there are only bug fixes.

- Java latest Major version is Java 7 (jdk 1.7).
- Java latest Minor version is Java 7 is (jdk 1.7.0_06).

Below diagram gives major and minor versions format



Q) Can we execute Java program using lower version JVM than compiler version?
A) We cannot use lower version JVM, it leads to error *Unsupported Major_Minor* exception.

Rule: JRE version must always >= compiler version.

Java Editions
Edition is a category. All Java features are categorized into three editions
They are

1. Java 2 platform Micro Edition - J2ME
2. Java 2 platform Standard Edition - J2SE
3. Java 2 platform Enterprise Edition - J2EE

Q) What is the significance of the number 2 in above names?
Nothing, it just represents editions were introduced in Java 2 version.

Q) Why Java concepts are divided into editions?
In Java 2 Version SUN introduced new features and concepts to support web and enterprise applications. So SUN thougnt that it is better to divide Java concepts into three categories for easy maintenance and for easy distribution.

Q) What features or concepts we have in above three editions?

- J2ME has concepts to develop software for consumer electronic devices means embedded systems, like mobile. It is popular for developing Mobile gamming applications.
- J2SE has concepts to develop software for Desktop based applications, nothing but the applications previously developed using C, C++.
- J2EE has concepts to develop software for Web and Enterprise applications. These applications are also called as high scale applications they are like banking and insurance based applications.

In Java 5, edition names are changed. SUN removed 2 from edition names, because it is not relevant to current version number Java 5. Below are the new names for all three editions:

1. Java platform Micro Edition - Java ME
2. Java platform Standard Edition - Java SE
3. Java platform Enterprise Edition - Java EE

Below diagram shows all java versions, code name, released date and editions.

Java Name	Principle Products	Code Name	Released Date	Editions
Java 1.0	jdk 1.0, jre 1.0	Oak	January 23, 1996	No editions
Java 1.1	jdk 1.1, jre 1.1	Oak	February 19, 1997	No Editions
Java 2 platform 1.2	j2sdk 1.2, j2sre 1.2	Playground	December 8, 1998	J2ME J2SE J2EE
Java 2 platform 1.3	j2sdk 1.3, j2sre 1.3	Kestrel	May 8, 2000	J2ME J2SE J2EE
Java 2 platform 1.4	j2sdk 1.4, j2sre 1.4	Merlin	February 6, 2002	J2ME J2SE J2EE
Java 5	Jdk 1.5, jre1.5	Tiger	September 30, 2004	Java ME Java SE Java EE
Java 6	Jdk 1.6, jre1.6	Mustang	December 11, 2006	Java ME Java SE Java EE
Java 7	Jdk 1.7, jre1.7	Dolphin	July 28, 2011	Java ME Java SE Java EE

Note: Code name is the Java software internal project name used in SUN Microsystems.

Java Software (JDK) installation and its folder Hierarchy

Follow below steps to install Java software - JDK

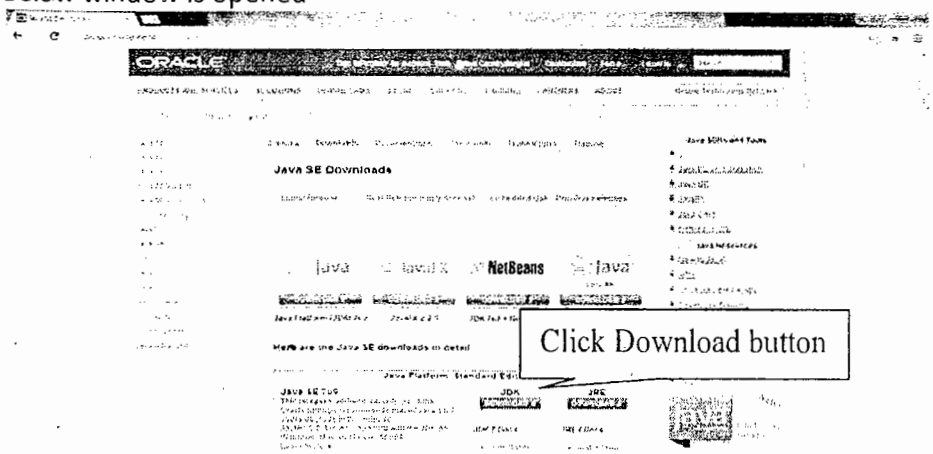
Step #1: Download latest version Java software from oracle.com

Java is an open source software available for free download at Oracle home site (oracle.com).

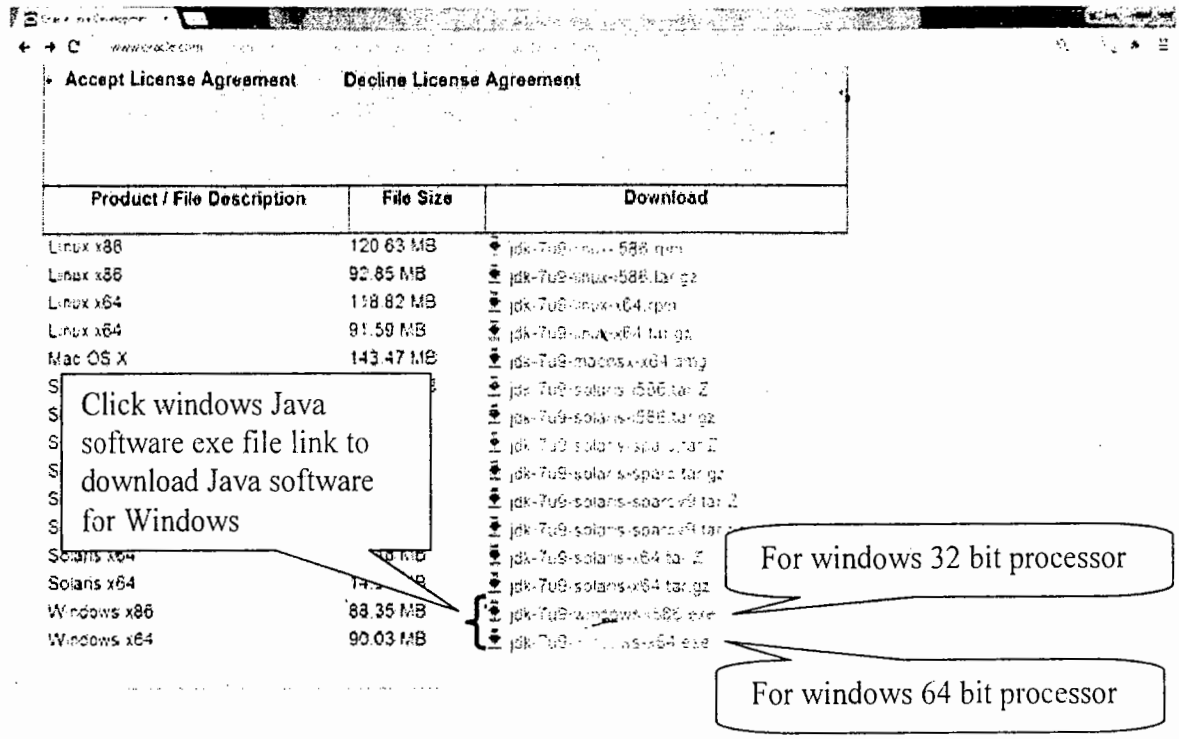
The path for downloading

Oracle.com -> Downloads -> Popular Downloads -> Java for Developers

Below window is opened



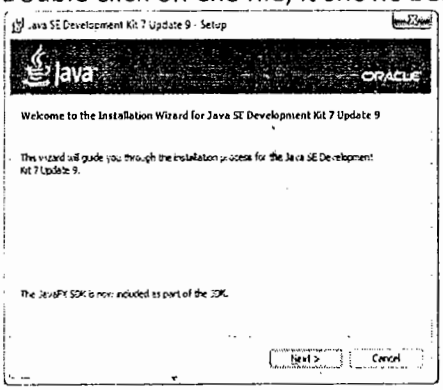
Below window is opened



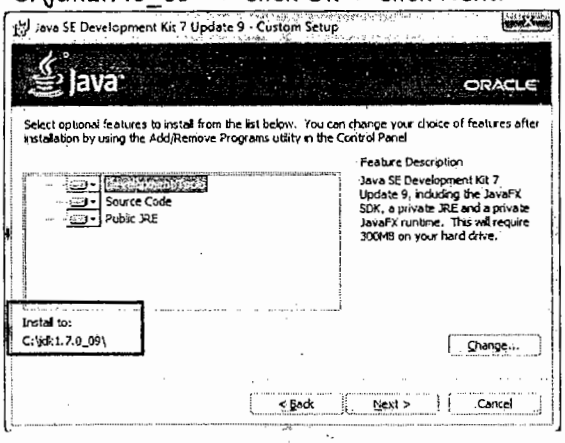
After downloading Java software

Follow below procedure to install Java

- Double click on exe file, it shows below welcome page. Click Next

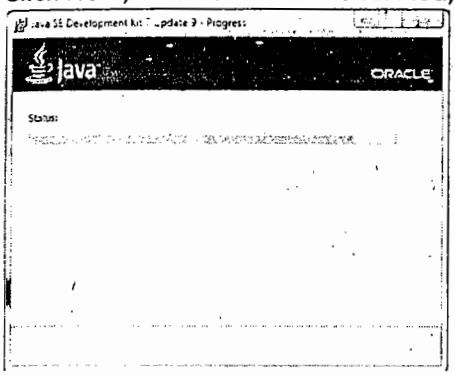


- In the next window it shows jdk installation folder path as shown below "C:\Program Files\Java\jdk1.7.0_09", click on Change button -> change above path to "C:\jdk1.7.0_09" -> click Ok -> click Next.

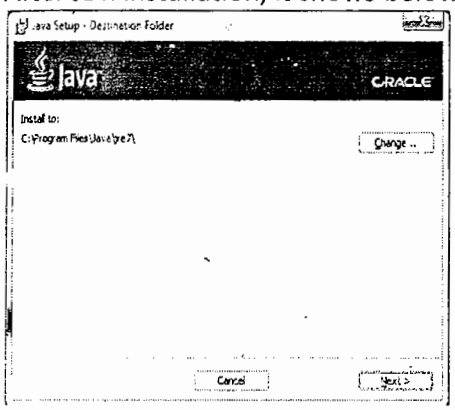


The space in the folder "Program Files" cause problems while compiling and executing java program from Editplus software.

- Click Next, JDK installation is started, and shows below window



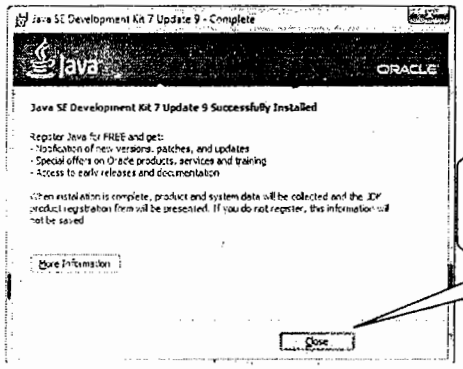
- After JDK installation, it shows below window to install public JRE. Click Next



No need to change folder path for Public JRE installation. It is used for executing Applets by Browser.

- Then it shows JRE installation progress bar as shown below

- After JRE installation it shows below final window, click Finish.



Click Close button to finish installation.

Java software installation is completed. It is installed at "C:\jdk1.7.0_09".
This folder path is called "Java software installed directory" or "JAVA_HOME".

What is JAVA_HOME means?

The Java software installed directory is called JAVA_HOME. We also follow this naming convention for other softwares. For instance Oracle installed directory is called ORACLE_HOME, Tomcat installed directory is called CATALINA_HOME, etc...

Why this naming convention is followed?

This naming conversion is followed in software industry for finding software installed folder path easily. Actually in companies, in server system softwares are installed by IT team and those are used by Development team. These naming conventions fill the gap between these two team members.

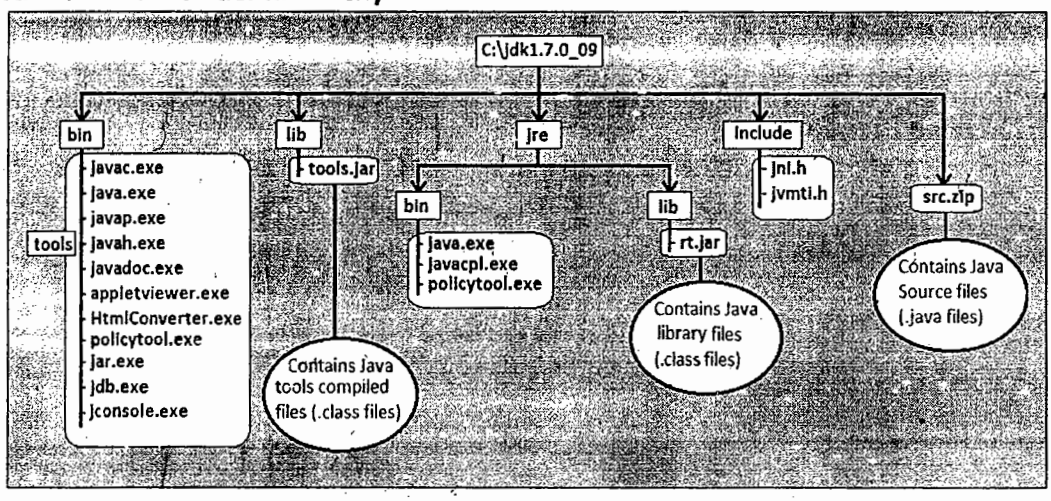
How?

After installing software, IT team stores that software home directory path using an environment variable as shown below, so that development team no need to contact every time to know where the Java, Oracle and Tomcat softwares are installed.

- JAVA_HOME for Java software,
- ORACLE_HOME for Oracle Software,
- CATALINA_HOME for Tomcat software.

Then development team uses the same environment variable to find software installed path. Check "Environment variables" topic for more details.

Java software folder hierarchy



Types of files available in software

Every Software has below two types of files

1. Binary files – are command files by default stored in a folder called "bin"
2. Library files – are program files by default stored in a folder called "lib"

- Binary files are command files, means they have commands to be executed in sequence. Java binary files have commands to compile and execute Java program.
- Library files are program files, means they have logic that is used to develop another applications. Java library files are used in developing new Java applications and applets.

Below diagram shows binary and library files extensions

binary files		Library files	
.exe .bat .cmd .sys	Windows binary files	.obj .h	C library files
		.class .jar	Java library files
.bin .sh	Linux / Solaris binary files	.msil .dll	.NET library files

Now let us understand how can we use Java software to develop new Java application?

How can we compile and execute Java program and from where?
Java programs are compiled and executed using Java binary files **javac** and **java.** We call these two binary files as "tools" in short form. These two tools are used from command prompt.

Where Java files must be stored, in Java software installed folder or in some other directory?
It is not recommended to store Java source files in Java software installed directory, because if we uninstall current Java software for installing next version, all our Java source files are also deleted. So for security reasons it is always recommended to store Java source files in another directory not in Java software installed directory.

What is the meaning of Present Working Directory?
The directory in which the Java source files are stored is called Present Working Directory. To compile and execute Java source files we must use *javac* and *java* tools from the present working directory at command prompt. So the first thing you should do after opening command prompt is changing current directory path to Present Working Directory path, then only your Java source files are identified for compilation and execution by *javac* and *java* tools.

Q) Is *javac* and *java* tools are available from outside Java software installed directory?
No, these files are not available from other folders. To check this point follow below procedure

- Open command prompt (click start-> run -> type *cmd* and then press Enter key)
- By default the folder path shown is current logged in user home directory path, for instance say C:\users\HariKrishna.
- Type *javac* and press Enter

```
Select C:\Windows\system32\cmd.exe

C:\Users\HariKrishna>javac
'javac' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\HariKrishna>
```

If you observe the above error it means javac tool is not available outside of Java software installed directory. To solve this problem and to access all Java tools throughout OS from all folders we must update environment variables.

So, **the point to be remembered** is: Just by installing software its binary and library files are not available from other directories automatically. We must update environment variables with that software binary and library files path.

Understanding environment variables

The variables created in OS to store software's binary and library files are called environment variables. For Java and Java related softwares we should use below two environment variables to store the software binary and library files, they are

1. *Path* - Used by OS to identify binary files
2. *Classpath* - Used by Compiler and JVM to identify Java library files

Path is a mediator between developer and OS to inform softwares binary files path.
Classpath is a mediator between developer and (Compiler, JVM) to inform the library files path those are used in our source code.

- Path environment variable is inbuilt available in Windows OS. So we must just update it with our software Path value in windows. But in Linux and Solaris OS we must create.

- JAVA_HOME and Classpath variables are not by default available they must be created by developer because they are related to only Java and Java related softwares.

Two ways to update environment variables

We have two ways to set environment variables

1. Temporary settings
2. Permanent settings

Temporary settings

Updating environment variables from command prompt is called temporary settings; because those settings are only available for that command prompt till it is closed. Once we close it, all settings are gone.

DOS commands to do temporary settings

- " **SET** " – for creating new environment variable
- " %<*environment variable*>% " – for retrieving existed value
- " ; " – for separating path values
- " **echo** " – for retrieving and print value

Syntax to create/update environment varaibles

```
SET <environment variable>=<new path>;%<environment variable>%
```

Below is the Java *Path* and *Classpath* values

Setting Java binary files path

```
SET Path=C:\jdk1.7.0_09\bin;%Path%
```

Setting Java library files path

```
SET Classpath=.;C:\jdk1.7.0_09\lib\tools.jar;C:\jdk1.7.0_09\jre\lib\rt.jar;%Classpath%
```

Observe that at beginning of Classpath environment variable I have placed "." operator. It is mandatory to place "." operator in Classpath environment variable it represents Present Working Directory. If we do not place "." operator in Classpath JVM cannot execute classes from present working directory and it terminates java program execution by throwing exception *"java.lang.NoClassDefFoundError"*.

Below diagram shows updating Path and Classpath environment variables at command

```
Select C:\Windows\system32\cmd.exe

C:\Users\HariKrishna>javac
'javac' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\HariKrishna>Set Path=C:\jdk1.7.0_09\bin

C:\Users\HariKrishna>javac
Usage: javac <options> <source files>
where possible options include:
  -g                         Generate all debugging info
  -g:none                    Generate no debugging info
  -g:{lines,vars,source}     Generate only some debugging info
  -nowarn                    Generate no warnings
  -verbose                   Output messages about what the compiler
  -deprecation               Output source locations where deprecate
```

As you can observe, now *javac* is identifying from another folder. So, now you can compile and execute Java programs in this command prompt.

The limitation in this approach the current settings are available only for this command prompt. These settings are not work for another command prompt. More over if we close this command prompt all settings are lost.

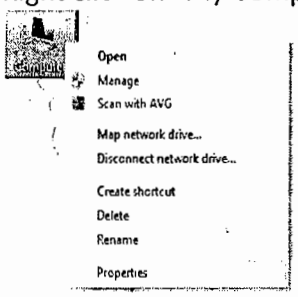
So in this approach we must set all these settings again and again for every new command prompt instance. To solve this problem we must store these settings permanently.

Permanent settings

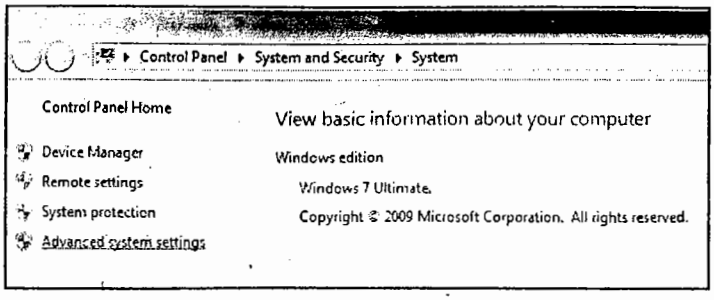
If we set above Java settings permanently they are available from all command prompts even after system restart.

The procedure is

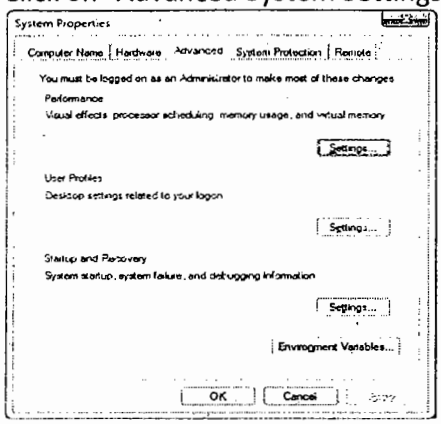
1. Right click on "My Computer" -> Click on "Properties",



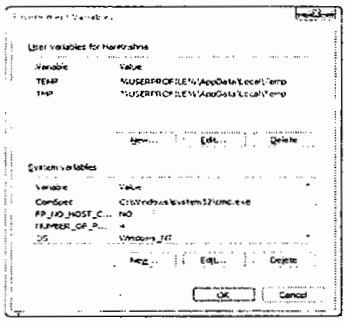
Below window is opened



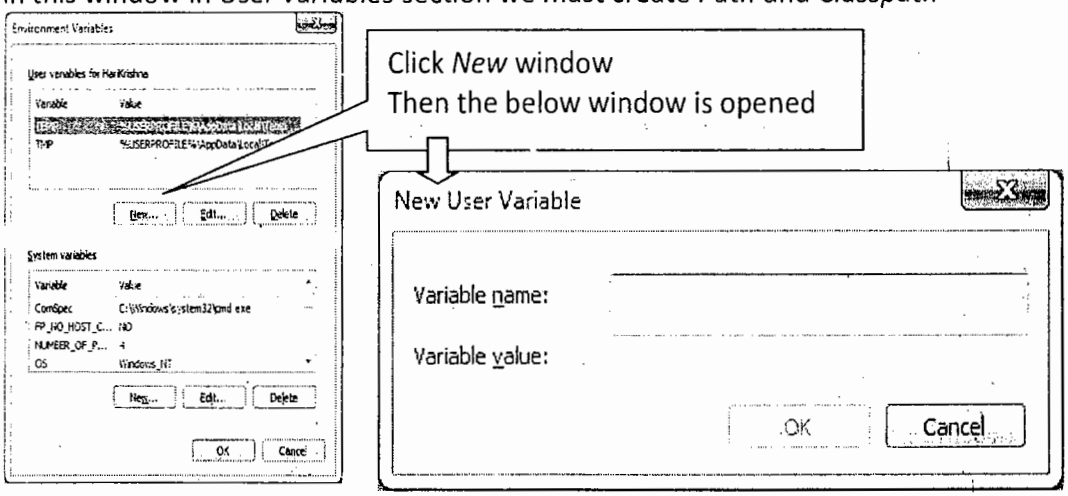
2. Click on "Advanced System Settings" hyper link, below window is opend



3. Click on "Environment Variables" button, below window is opened



4. In this window in *User variables* section we must create *Path* and *Classpath*

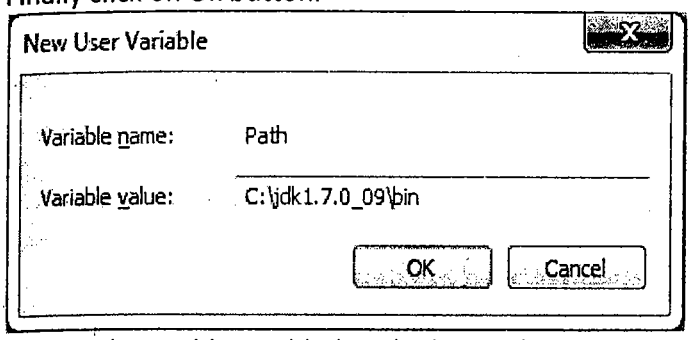


Enter below values as shown below

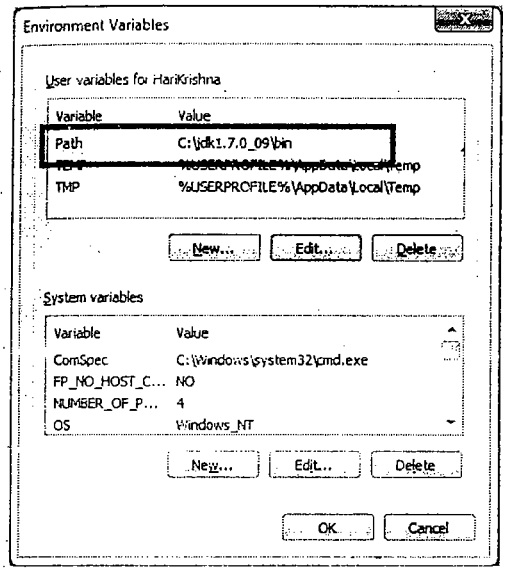
Variable name: Path

Variable value: C:\jdk1.7.0_09\bin

5. Finally click on Ok button.



Then Path variable is added to the list as shown below



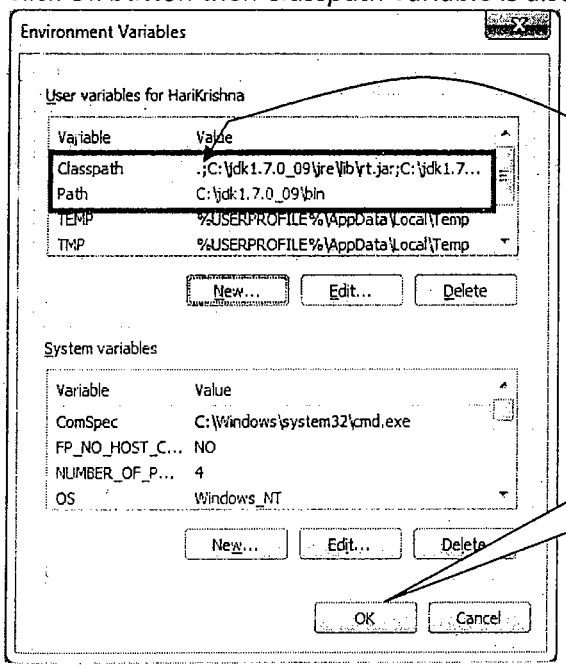
Now create *Classpath* variable

6. Click again *New* button and enter values as below

   Variable name: Classpath

   Variable value: .;C:\jdk1.7.0_09\jre\lib\rt.jar;C:\jdk1.7.0_09\lib\tools.jar

7. Click Ok button then *Classpath* variable is also added to list as shown



Point to be remembered
Classpath environment variable must has "." operator, else Java programs are not executed by JVM. Program execution is terminated with exception "java.lang.NoClassDefFoundError".

Click Ok, and also click Ok on other windows till all windows are closed.

We completed Java software installation and Environment variables setup

Now let us cross check Java path settings

Open new command prompt -> type *javac* command -> press Enter key on keyboard.
You should get the message shown below.

```
Select C:\Windows\system32\cmd.exe

C:\Users\HariKrishna>javac
Usage: javac <options> <source files>
where possible options include:
  -g                         Generate all debugging info
  -g:none                    Generate no debugging info
  -g:{lines,vars,source}     Generate only some debugging info
  -nowarn                    Generate no warnings
  -verbose                   Output messages about what the com
```

If above message does not appear, you have done some wrong setting.
Cross check the *Path* value at command prompt using *echo* command as shown below

```
Select C:\Windows\system32\cmd.exe

C:\Users\HariKrishna>echo %Path%
C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\Wind
owsPowerShell\v1.0\;C:\Program Files\Lenovo\Bluetooth Software\;C:\apache-tomcat
-6.0.26\bin;C:\Program Files\EditPlus 2;C:\jdk1.7.0_09\bin

C:\Users\HariKrishna>
```

Cross check your Path value with one shown in the above screen. It must be placed as commented in the above diagram.

Point to be remembered on Oracle path

In Advanced Java course you will install Oracle software for developing JDBC applications. Oracle software Path is automatically updated in environment variables section. Make sure it is placed after Java software path value. Else you will experience an exception *"java.lang.UnsupportedVersionError"* for java programs execution, because oracle software also contains Java software which is lesser version than your using.

Installing Java software in Linux / Solaris

1. Download "bin" format jdk software from oracle.com that is given specific to Linux or Solaris

2. Open shell prompt
3. Change current directory path to software copied folder path
   a. /home/users/Harikrisna$cd /home/softwares/java
   b. /home/softwares/java$

4. Execute bin file
   a. /home/sotwares/java$ jdk1.7.0_09.bin <-|
5. Java software is installed here in "java" folder
6. So the JAVA_HOME will be "/home/softwares/java/jdk1.7.0_09"

Setting environment variables

To set "Path" and "Classpath" environment variables in Linux and Solaris we must use below commands

Window commands	Linux / Solaris commands
SET	export
%Path%	$Path
;	:
echo %Path%	echo $Path

Temporary settings

At sheli prompt run below commands

$export Path=/home/sotwares/java/jdk1.7.0_09/bin:$Path
$export
Classpath=.;/home/softwares/java/jdk1.7.0_09/lib/tools.jar:/home/softwares/java/jdk1.7.0_09/jre/lib/rt.jar:$Classpath

Permanent settings

In Linux:

- Open user home directory
- Find a file called "*.bash_profile*"
- Open it. [Right click on that file -> click "Edit"]
- Save above two export commands in this file
- Permanent settings are over.

In Solaris:

- Open user home directory
- Find a file called "*.profile*"
- Open it. [Right click on that file -> click "Edit"]
- Save above two export commands in this file
- Permanent settings are over.

Which edition concepts are installed from JDK or JRE software installations?

Only Java SE concepts are installed. Also to install Java EE concepts, in addition to JDK we must also install server softwares like – Tomcat, Weblogic, Glassfish, etc...

Java Reference Books

Theory books

1. **Core Java with OCJP and JVM Architecture** By Hari Krishna
2. **SCJP 1.6** By Kathy Sierra
3. **Effective Java** By Josh Bloch
4. **Inside JVM** By Bill Venners
5. **Java Cookbook Solutions and Examples for Java Developers** By Ian Darwin
6. **Thinking in Java** By Bruce Eckel

Programming books

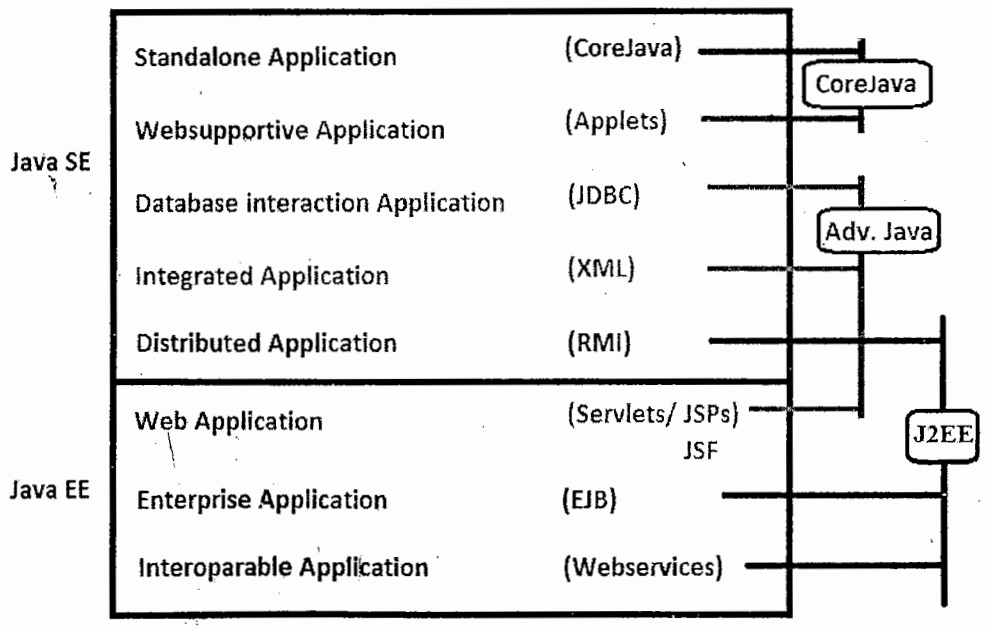
1. **Java Programming** By Hari Krishna
2. **Written test questions in Java programming** by Yashavant Kanetkar
3. **Test Killer** (SCJP Dump)
4. **Develop all C programs in Java**

Online reference

1. oracle.com
2. javastuff.in
3. javapapers.com
4. mindprod.com
5. javaspecialists.eu
6. roseindia.net
7. javaranch.com
8. serverside.com
9. jguru.com
10. java-questions.com
11. stackoverflow.com,
12. tutorialspoint.com,
13. MyJavaHub.com,
14. javamex.com
15. Projecttopics.info

Now we are ready to learn programming in Java. To develop Java program we must learn below important points.

What are the different types of Applications we can develop by using Java SE and EE?
Using Java SE and Java EE we can develop 8 types of applications, they are

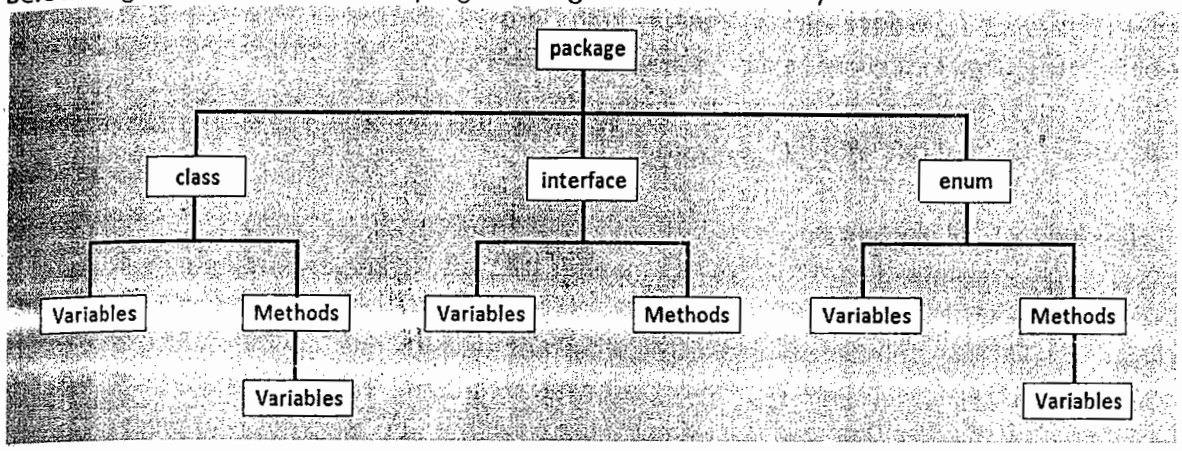


The point to be noticed here is *as per SUN we do not have* Core Java and Advanced Java. These are created by our industry exports for learning Java comfortably.

In this diagram you can observe, the applications list you learn as part of CJ, Adv Java and J2EE courses.

Java Basic Programming Elements

In Java we have 6 basic programming elements to develop all above 8 types of applications. Below diagram shows the 6 basic programming elements hierarchy



Definitions

1. **package**: It is a Java folder used to group related classes, interfaces and enums also used to separate new classes from existed classes if both have same name.

2. **class**
3. **interface**
4. **enum**

   All these three are Java files used to group Java data and logic. enum was introduced in Java 5

5. **variable:** It is a named memory location used to store Java data , such as numbers, characters, strings etc...

6. **method:** It is a sub block of a class used to implement logic of an object operations. **Rule:** logic should be placed only inside a method, cannot be placed at class level directly.

Q) What is the difference between class, interface and enum?

A) interface is a fully unimplemented class, it is used for defining set object operations, where as class is a fully implemented *class,* it is used for implementing an object operations.

So, interface does not allow methods with logic where as class allows methods with logic.

For example: Bank is the object created as interface in Java and HDFCBank, ICICIBank are created as classes with bank operations withdraw, deposit implementations.

Q) Why enum?

It was introduced in Java 5 version to create set of named constants for creating menu kind of items such as Restaurant, Bar menu.

Before Java 5 this menu is created by using class, since it is creating some problems SUN introduced enum with new syntaxes and rules.

Q) What is the mandatory basic programming element?
class is mandatory, because it allows us to define methods with logic.

Q) Why does a Java program start with a class?
Because of below two reasons every Java program starts with a class

1. The real world object can be represented only using class.
2. We can define methods with logic only using class.

Essential statements of Java program
To develop, compile, execute and print output on monitor (console) every Java program must has below 3 statements

1. *class block*: because only class allows us to define method with logic.
2. *main method*: because it is initial point of class logic execution.
3. *System.out.println*: because in Java all kinds of data is printed by using this statement.

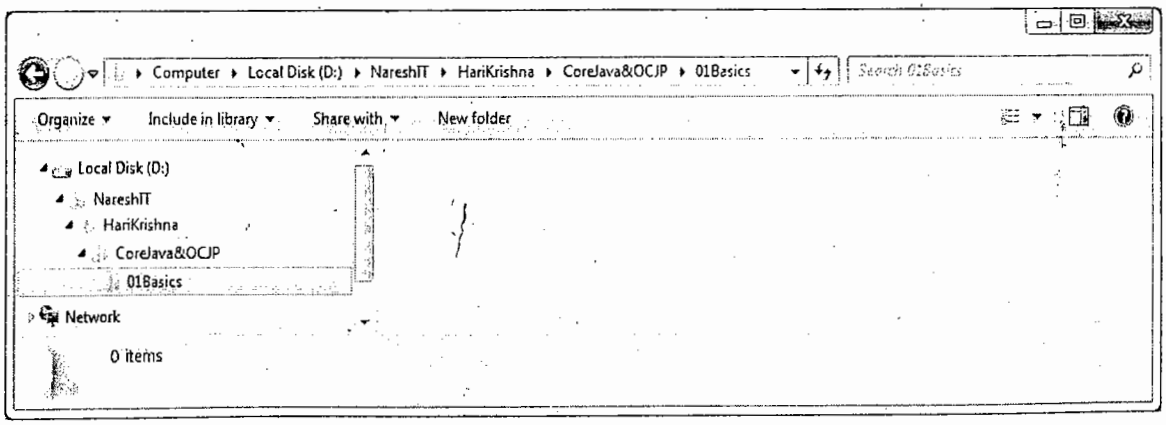
To print **Hi** using above statements the program would be look like:

```
class FirstProgram
{
        public static void main(String[] args)
        {
                System.out:println("Hi");
        }
}
```

Softwares required for developing Java program

1. **JDK** - for compiling and executing
2. **Editor software** - for typing and saving.
   Known editor softwares are notepad, Editplus, Textpad, Notepad++, Eclipse, NetBeans.
3. **Command prompt** - for executing *javac*, and *java* commands

Procedure to develop Java program
Step #1: Create a folder to store all your programs. This folder is called the "PresentWorkingDirectory". Assuming the folder path as (**D:\Naresh IT\HariKrishna\01Basics**)



Step #2: Open notepad (start -> run -> notepad) and type below code

Untitled - Notepad

File Edit Format View Help

```
class FirstProgram
{
        public static void main(String[] args)
        {
                System.out.println("Hi");
        }
}
```

Step #3: Save this file in *01Basics* folder with name *Example.java*.

Note: Java file name can be user defined name. It is not always mandatory to save Java file name with same class name. If class is declared as "public" then only it is mandatory to save the file with class name. Let us try with different java file name.

Example.java - Notepad

File Edit Format View Help

```
class FirstProgram
{
        public static void main(String[] args)
        {
                System.out.println("Hi");
        }
}
```

Folder structure

Computer ▸ Local Disk (D:) ▸ NareshIT ▸ HariKrishna ▸ CoreJava&OCJP ▸ 01Basics

Organize ▾ Include in library ▾ Share with ▾ New folder

Local Disk (D:)
NareshIT
HariKrishna
CoreJava&OCJP
01Basics
Network

Example.java

0 items

Program development is completed, now let us compile and execute

Compilation and execution

Step #4: Compilation

1. Open command prompt
2. Change Drive and then directory to current working directory to *01Basics* folder
3. Then use *javac* tool to compile Example.java file, as shown below

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\HariKrishna>D:

D:\>cd "D:\NareshIT\HariKrishna\CoreJava&OCJP\01Basics"

D:\NareshIT\HariKrishna\CoreJava&OCJP\01Basics>javac Example.java

D:\NareshIT\HariKrishna\CoreJava&OCJP\01Basics>_
```

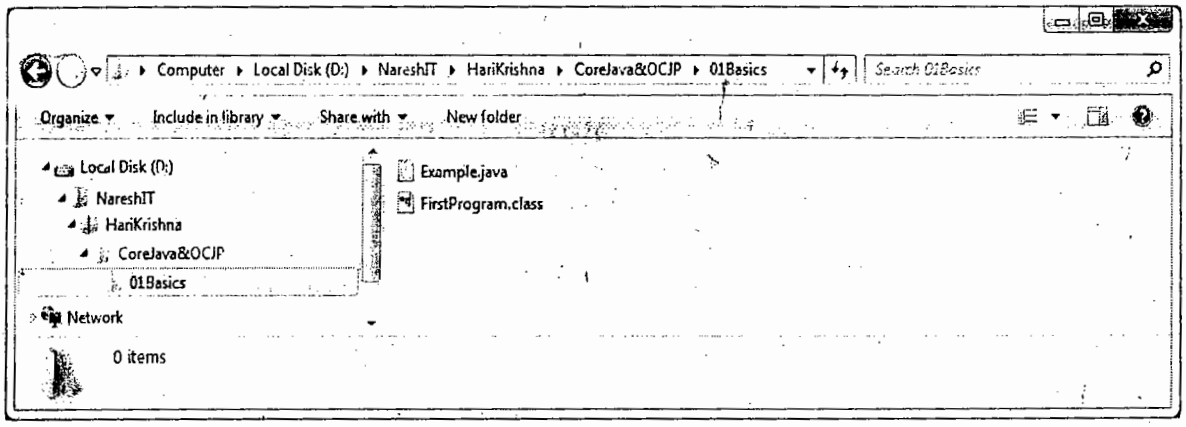
Compiler has generated ".class" file successfully for the class FirstProgram in *01Basics* folder.

Q) Guess what is the ".class" file name?

It is name is *FirstProgram.class*, not Example.class

Note: Compiler generates ".class" file with the *class name* not java file name.

Check 01Basics folder for ".class" file name



Step #5: Execution

Use *java* tool to execute *FirstProgram* class bytecodes file, as shown below

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\HariKrishna>D:

D:\>cd "D:\NareshIT\HariKrishna\CoreJava&OCJP\01Basics"

D:\NareshIT\HariKrishna\CoreJava&OCJP\01Basics>javac Example.java

D:\NareshIT\HariKrishna\CoreJava&OCJP\01Basics>java FirstProgram
Hi

D:\NareshIT\HariKrishna\CoreJava&OCJP\01Basics>_
```

Compiler activity:
It takes Java file name as its input and generates bytecodes for all classes defined in that java file and stores each class bytecodes in a separate ".class" file with name same as class name.

JVM activity:
It takes class name as its input and searches for a .class file with the given class name. If it found it reads and loads that .class file bytecodes into JVM, then starts that class logic execution by calling "main" method.

Q) Is main method mandatory for compilation or execution?
only required for execution not for compilation.

If class does not have main method program compiled fine, but cannot be executed. It leads to runtime error or exception "Exception in thread "main" *java.lang.NoSuchMethodError*: main"
From Java 7 onwards we do not get above exception we get a message. That message is

> Error: Main method not found in class FirstProgram, please define the main method as:
> public static void main(String[] args)

Q) What does the compiler do if the given java file is not found?
It throws CE: *javac: file not found: <filename>*

Q) What does JVM do if the given class's .class file is not found?
It throws RE: java.lang.NoClassDefFoundError: <classname>
From Java 7 onwards in this case we do not get exception we get a message. That message is

> Error: Could not find or load main class A

```
Select C:\Windows\system32\cmd.exe

D:\NareshIT\HariKrishna\CoreJava&OCJP\01Basics>javac A.java
javac: file not found: A.java
Usage: javac <options> <source files>
use -help for a list of possible options

D:\NareshIT\HariKrishna\CoreJava&OCJP\01Basics>java A
Exception in thread "main" java.lang.NoClassDefFoundError: A
Caused by: java.lang.ClassNotFoundException: A
        at java.net.URLClassLoader$1.run(URLClassLoader.java:202)
```

Compilation and Runtime errors:
The errors thrown by compiler at the time of compilation are called compile time errors. These errors are raised due to syntax mistakes, like spelling mistakes, wrong usage of keywords, missing ";" at end of statements, etc...

The errors thrown by JVM at the time of execution are called runtime errors or exceptions. These errors are raised due to logical mistakes, like executing class without main method, dividing integer number with ZERO, accessing array values with wrong index, etc...

Testing:
To get command over compilation and execution process, execute below test cases

In the above program
Case #01: remove ";" at end of Sopln statement, then compile the program compiler throws below compile time error

```
C:\Windows\system32\cmd.exe

D:\NareshIT\HariKrishna\CoreJava&OCJP\01Basics>javac Example.java
Example.java:5: ';' expected
                System.out.println("Hi")
                                       ^
1 error

D:\NareshIT\HariKrishna\CoreJava&OCJP\01Basics>
```

As you can notice, compiler is clearly showing the committed mistake
It displays

- The error we committed
- Line number
- The place where we should keep ";" to resolve this error.

Java is very easy, right?

Case #02: Remove class name, then compile
Case #03: Remove method name, then compile
Case #04: Change S to s in Sopln, then compile
Case #05: Change args to hari, then compile and execute
Case #06: Remove String[] args, then compile and execute
Case #07: Remove void, then compile
Case #08: Remove static, then compile
Case #09: Remove public, then compile
Case #10: Remove main method definition completely, then compile and execute
Case #11: Finally remove class definition completely, then compile and execute
Before executing this case remove "*.class*" from 01Basics folder, then try.

Find out valid main method declarations from the below list

The program should compile and execute without errors when we use below main methods

01. public static void main(String[] args)
02. public static void main(String []args)
03. public static void main(String args[])
04. public static void main([]String args)
05. public static void main(String[] hari)
06. static public void main(String[] args)
07. public static void main(String... args)
08. public static int main(String[] args)
09. public static void mian(String[] args)
10. public static void main(String[5] args)
11. public static void main(int[] args)
12. public static void main(String args)
13. public static void main()
14. static void main(String[] args)
15. public void main(String[] args)
16. void main(String[] args)

Below I have give list interview questions on basic java programming.

Q) Can we create empty Java file, can we compile and execute it?
Yes, but after compilation we do not have .class file. So we cannot execute.

Example. java

```
>javac Example.java
```

Q) Can we create empty class, can we compile and execute it?
Yes we can create and compile empty class. Compiler generates .class file with that class name. We cannot execute this class as it does not have main method. It leads to exception

Example. java

```
class A{

}
```

```
>javac Example.java
        |-> A.class
>java A
Exception ..............
```

Q) Is it mandatory that java file name and class name should be same?
No, not always- file name can be user defined name.

Q) When should the Java file name and class name be the same?
If class is declared as public, file name should be the same as the public class name, else its name can be user defined.

Example. Java

```
class A{

}
```

Example.java

```
public class A{

}
```

CE: class A is public, should be declared in a file named A.java

Q) In a single Java file how many classes can we define?
We can define more than one class. The rule is, class names should be different.

Q) If a java file has multiple classes what is the java file name?
public class name. If there is no public class, java file name can be user defined.

Q) In a java file, how many public classes can we define?
Only one. We can define one public class and multiple non-public classes.

Example. Java

```
class A{
}

class B{
}
```

Example.java

```
public class A{
}

class B{
}
```

A.java

```
public class A{
}

class B{
}
```

A.java

```
public class A{
}

public class B{
}
```

Q) If we compile multiple classes java file, how many .class files are generated by compiler?
Compiler generates .class files as many class definitions as we have in that java file. Compiler generates .class file separately for every class with that class name.

Example.java

```
class A{
        public static void main(String[] args){
                System.out.println("A main")
        }
}

class B{
}
```

```
>javac Example.java
        |-> A.class
        |-> B.class
```

Q) How can we execute all classes?
We should execute each class separately using Java command. If that class has main method JVM prints output, else throws exception. Below diagram shows executing *A* and *B* classes

```
C:\Windows\system32\cmd.exe

D:\NareshIT\HariKrishna\CoreJava&OCJP\01Basics>javac Example.java

D:\NareshIT\HariKrishna\CoreJava&OCJP\01Basics>java A
A main

D:\NareshIT\HariKrishna\CoreJava&OCJP\01Basics>java B
Exception in thread "main" java.lang.NoSuchMethodError: main

D:\NareshIT\HariKrishna\CoreJava&OCJP\01Basics>_
```

Q) What is meant by user defined method and predefined method?
Developer defined method is called user defined or custom method.
Already defined methods are called predefined methods.
These methods may be developed by SUN Microsystem developers, or some other developers.
Ex: println() method is predefined.

Q) What is meant by user defined class and predefined class?
Developer defined class is called user defined or custom class.
Already defined class is called predefined class.
These class may be developed by SUN Microsystem developers, or some other developers.

For example : System, String are predefined classes
FirstProgram, A, B are user defined classes

Q) Does JVM execute user defined methods automatically?
No, it executes only main method. To execute user defined methods We must call them from main method or a method calling from main

What is the output from below program?

Example.java

```
class A{
        void m1(){
                System.out.println("A m1");
        }
        public static void main(String[] args){
                System.out.println("A main");
        }
}
```

```
>javac Example.java
>java A
A main
```

m1() method is user defined method. JVM has not executed it, because it is not called from main method.

Below program shows **calling m1()** method from **main method, *but*** leads to Compile time error: **"non-static method m1() cannot be referenced from static context"**

Example.java

```
class A{
        void m1(){
                System.out.println("A m1");
        }
        public static void main(String[] args){
                System.out.println("A main");

                m1();  ╳ CE: non-static method m1() cannot be referenced from static context
        }
}
```

The **Reason** for the above CE is m1() method does not have permission to access directly, for the time being assume it doesn't have memory.

JVM provides memory for variables and methods only if we use either of the below keywords

1. **static** or
2. **new**

In this chapter I only use static keyword, in next chapters I will teach you the usage of new keyword. Like in main method, for user defined methods also we must use static keyword before return type, then this method can be called directly from main method.

Example.java

```
class A{
        static void m1(){
                System.out.println("A m1");
        }
        public static void main(String[] args){
                System.out.println("A main");
                m1();
        }
}
```

```
>javac Example.java
>java A
A main
A m1
```

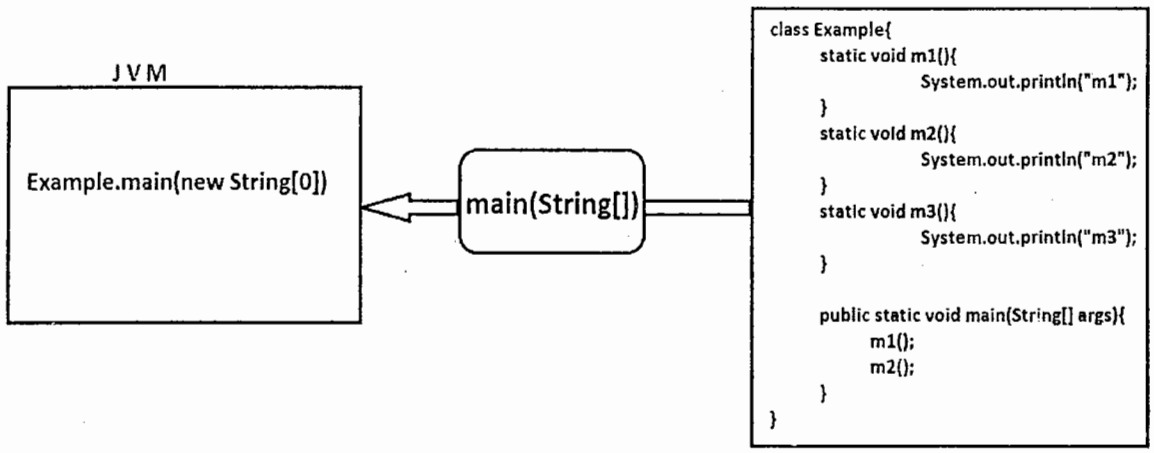
Q) Does JVM execute user defined methods by itself just by declaring them as static?
A) No, static methods also should be called from main method.

Q) Then how main method is executed by JVM automatically? Is it not because of static keyword?
A) Yes, main method is executed not because of static keyword, because it is called from JVM software.

Why JVM executes only man method why it does not execute user defined methods?
JVM executes only main method because it has main method's calling statement. It does not have user defined method's calling statement in its logic. It executes user defined methods only if they are called from main method.

Why JVM software has only main method calling statement why not user defined methods? Or why main method is the initial point of class logic execution? Or Why JVM executes only main method why not user defined methods?
JVM developer doesn't know what methods should be executed, when they should be executed and in which order.

Hence it is the developer responsibility to inform about the methods those must be executed by JVM. For this purpose there should be a common method to call these methods, that method should be known to both JVM and developer, and that method prototype should be given by the JVM S/W designer (vendor) as it must be called from JVM. That method is main method.



Definition of main method
main method is the mediator method between Java developer and JVM to inform what are the methods should be execution when and in which order.

Q) How many user defined methods can we define in a class?
We can define multiple methods, but rule is, method names should be different.

Q) What is the order of execution of all user defined methods?
In the order they are called from main method or from any one of the method calling from main method.

What the output from below program?

Example.java

```
class A{
        static void m1(){
                System.out.println("m1");
        }
        static void m2(){
                System.out.println("m2");
        }
        public static void main(String[] args){
                System.out.println("main");
                m1();
        }
}
```

```
>javac Example.java
>java A
main
m1
```

Example.java

```
class A{
        static void m1(){
                System.out.println("m1");
                m2();
        }
        static void m2(){
                System.out.println("m2");
        }
        public static void main(String[] args){
                System.out.println("main");
                m1();
        }
}
```

```
>javac Example.java
>java A
main
m1
m2
```

Q) If a class does not have main method, how can we execute user defined methods of that class?

We should use another class main method. We should call this method with its class name.

In projects, we do not write main method in every class. Instead we write main method in one class, and we will call all other classes' methods from this class's main method for testing.

Basically main method is given to start class logic execution but not for developing logic directly.

Below program shows calling User defined method from another class MM.

Example.java

```
class A{
        static void m1(){
                System.out.println("A m1");
        }
}

class B{
        static void m2(){
                System.out.println("B m2");
        }

        public static void main(String[] args) {
                System.out.println("B main");
                m2();
                A.m1();
        }
}
```

```
>javac Example.java
>java A
Exception ……..

>java B
B main
B m2
A m1
```

Q) Is it possible to create user defined method in all classes with same name?

Yes, we can define user defined methods with same name in multiple classes. To call those methods we must use that method's class name.

What is the output from the below program?

Example.java

```
class A{
        static void m1(){
                System.out.println("A m1");
        }
}
class B{
        static void m1(){
                System.out.println("B m1");
        }
}
class Test
        public static void main(String[] args) {
                System.out.println("Test main");
                A.m1();
                B.m1();
        }
}
```

```
>javac Example.java
>java Test
Test main
A m1
B m1
```

Q) In single java file, can we define main method in all classes?

Yes we can define. There is no CE, RE. Because compiler stores each class bytecode in separate .class file there is no confusion for JVM in executing main method.

Check below program.

```
//Example.java
class A{
        public static void main(String[] args) {
                System.out.println("A main");
        }
}

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

```
>javac Example.java
>java A
A main

>java B
B main
```

Q) Can we call main method explicitly?

Yes, even the main method is a method, so we can call.

Syntax to call main method

main(*new String[0]*);

In place of ZERO we can pass any +ve integer number

Write a program to call A class main method from B class main method.

Example.java

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

class B{
        public static void main(String[] args) {
                System.out.println("B main");
                A.main(new String[0]);
        }
}
```

```
>javac Example.java
>java A
A main

>java B
B main
A main
```

Q) What does happen when we pass negative number as size to array object?

For example: main (new String[-5]);

A) No CE, but it leads exception "java.lang.NegativeArraySizeException"

Q) What does happen when we call main method in the same class?

```
class A{
        public static void main(String[] args){
                System.out.println("main");
                main( new String[0] );
        }
}
```

A) No CE, but it leads exception "java.lang.StackOverflowError", because it leads to recursive method call

Q) What is the output from the below program?

```
class A{
        public static void main(String[] args){
                System.out.println("A main");
                m1();
        }
        static void m1(){
                System.out.println("A m1");
                main( new String[0] );
        }
}
```

Options:

1. compile time error
2. exception
3. A main
   A m1
   A main
   A m1
   A main
   A m1
   exception

Q) Can we overload main method?

A) Yes

Q) Then which main method is executed by JVM?

A) String[] parameter method

What is the output from the below program?

```
class A{

   public static void main(String args){
      System.out.println("In main(String)");
   }

   public static void main(String[] args){
      System.out.println("In main(String[])");
   }

   public static void main(int[] args){
      System.out.println("In main(int[])");
   }
}
```

Below program shows implementing all above points
//MultipleClasses.java

```
class A{
}

class B{
        static void m1(){
                System.out.println("B m1");
        }
}

class C{
        static void m2(){
                System.out.println("C m2");
        }
        public static void main(String[] args){
                System.out.println("C main");
        }
}

class D{
        static void m3(){
                System.out.println("D m3");
        }
        public static void main(String[] args){
                System.out.println("D main");
                m3();
        }
        static void m4(){
                System.out.println("D m4");
        }
}
```

```
class E{
        static void m5(){
                System.out.pritln("E m5");
        }
        public static void main(String[] args){
                System.out.pritln("E main");

                m5();

                B.m1();
                C.m2();

                D.m3();
                D.m4();

                D.main(new String[0]);

                E.m5();
        }
}
```

Compilation

```
>javac MultipleClasses.java
                |->A.class
                |->B.class
                |->C.class
                |->D.class
                |->E.class
```

Execution:

> java A

Exception in thread "main" java.lang.NoSuchMethodError: main

> java B

Exception in thread "main" java.lang.NoSuchMethodError: main

> java C

C main

> java D

D main

D m3

> java E

E main

E m5

B m1

C m2

D m3

D m4

D main

D m3

E m5

Q) How can we execute all class's methods with single class execution?

A) Call all those methods from the current class main method or from the method that is calling from main method.

"E" class is an example of above point development.

Q) What is the difference between System.out.print() and System.out.println();?

A) "ln' is the difference.

It means

- println() method places the cursor in the next line after printing current output. So that the next coming output will be printed in next line.
- But whereas print() method places the cursor in the same line after printing current output. So that the next coming output will be printed in same line.

For Example

	O/P
System.out.println("A");	A
System.out.println("B");	B
	_

	O/P
System.out.print("A");	AB_
System.out.print("B");	

Write a program to print the given 5 statements in 3 lines

```
class A {
        public static void main(String[] args){
                System.out.print("Welcome to AMAZING WORLD Java, ");
                System.out.println("Welcome to JAVA HEAVEN Naresh Technologies");
                System.out.print("Do Smart work, ");
                System.out.println("Get Best Results");
                System.out.println("All the best for your careers");
        }
}
```

output

```
Welcome to AMAZING WORLD Java, Welcome to JAVA HEAVEN Naresh Technologies
Do Smart work, Get Best Results
All the best for your careers.
```

In the above program is class name "A" is meaningful? Is it provide any information to that class user the operation that we are doing in that class?

No. It is not a meaningful name.

To develop project with more readability we must follow SUN given coding standards.

Coding standards and Naming conventions

Coding Standards (CS) and Naming Conversions (NS) are suggestions given by SUN. CS and NC help developers to develop project with more readability and understandability.

In projects development we should chose names for basic programming elements very carefully. *The name should be meaningful and relevant to present context*.

For example if we are developing a project for the bank,

- class name should be Bank,
- variable names should be accountNumber, balance, username, password, etc...
- method names should be withdraw, deposit, getBalance, transferMoney, etc...

Why coding standards?

A program is written once, but read many times

- During debugging
- When adding to the program
- When updating the program
- When trying to understand the program

Anything that makes a program more readable and understandable saves lots of time, even in the short run. So we must follow coding standards.

If we do not follow *coding standards* and *naming conventions* code is not readable and that code is not allowed in projects.

Naming conventions

1. **Naming a class, interface, enum**
   - Class name should be "*noun*", because it represents *things*.
   - Class name should be in *title case*, means
     "*Every word first letter should be capital letter*".

   For example: HariKrishna, NareshTechnologies, Tiger, StringWordsFinder, PrintStream

2. **Naming variable**
   - Variable name also should be "*noun*", because it represents *values*.
   - In variable name, "*first word first letter should be small* and *after that every word first letter should be capital*".

     For example: customerName, username, password, balance, minimumBalance.

   - In *final variable* all letters should be capital and words must be connected with '_'.
     For example: MIN_BALANCE, PI, RED, BLUE, BLACK, MAX_PRIORITY

3. **Naming methods**
   - Method name should be "*verb*", because it represents *action*.
   - In method name, "*first word first letter should be small* and *after that every word first letter should be capital and should follow ( )*".

     For example: getUsername(), getPassword(), getBalance(), findArea()

4. **Naming package**
   - All letters should be small, and its length should be as short as possible.

     For example: io, lang, util, dao, beans, blogic, gui.

Q) If we do not follow above CS and NS will it leads to CE or RE?
A) No CE, No RE. Your program is not readable, not accepted in the project.

So we can conclude, CS and NC are contracts among all Java developers to develop project with same style of code.

By following above CS and NC, find out each member type in the below program?

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

                System.out.println("Hi");
        }
}
```

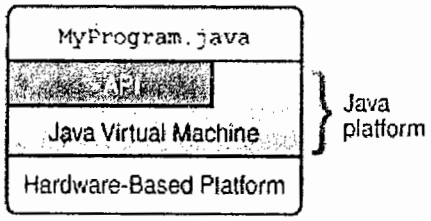
Java platform Architecture

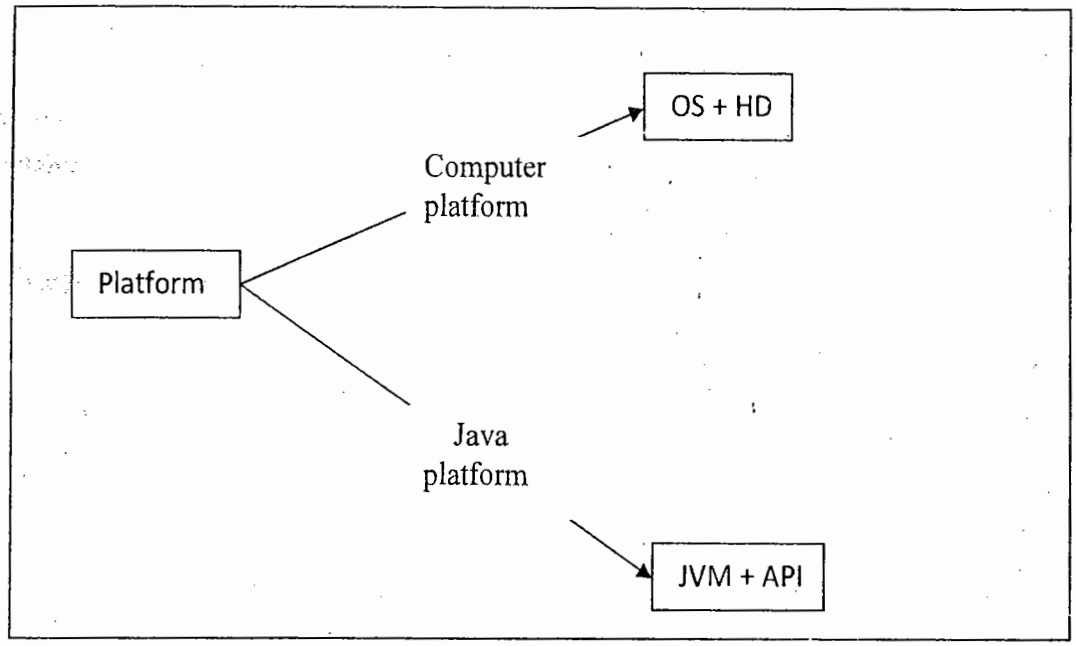
Java architecture has below four components

1. Java source file
2. Java .class file format
3. Java Application Programming Interface (API)
4. Java Virtual Machine (JVM)

- JVM and API both together are called Java Platform. It is a software based platform.

Below diagram shows Java's platform architecture (Copied from Oracle tutorial)





Below content copied from Oracle tutorial

What Can Java Technology Do?

The general-purpose, high-level Java programming language is a powerful software platform. Every full implementation of the Java platform gives you the following features:

Development Tools: The development tools provide everything you'll need for compiling, running, monitoring, debugging, and documenting your applications.

Application Programming Interface (API): The API provides the core functionality of the Java programming language. It offers a wide array of useful classes ready for use in your own applications.

Deployment Technologies: The JDK software provides standard mechanisms such as the Java Web Start software and Java Plug-In software for deploying your applications to end users.

User Interface Toolkits: The Swing and Java 2D toolkits make it possible to create sophisticated Graphical User Interfaces (GUIs).

Integration Libraries: Integration libraries such as the Java IDL API, JDBC API, Java Naming and Directory Interface ("J.N.D.I.") API, Java RMI, and Java Remote Method Invocation over Internet Inter-ORB Protocol Technology (Java RMI-IIOP Technology) enable database access and manipulation of remote objects.

How Will Java Technology Change My Life?

We can't promise you fame, fortune, or even a job if you learn the Java programming language. Still, it is likely to make your programs better and requires less effort than other languages. We believe that Java technology will help you do the following:

Get started quickly: Although the Java programming language is a powerful object-oriented language, it's easy to learn

Write less code: Comparisons of program metrics (class counts, method counts, and so on) suggest that a program written in the Java programming language can be four times smaller than the same program written in C++.

Write better code: The Java programming language encourages good coding practices, and automatic garbage collection helps you avoid memory leaks.

Develop programs more quickly: The Java programming language is simpler than C++, and as such, your development time could be up to twice as fast when writing in it. Your programs will also require fewer lines of code.

Avoid platform dependencies: You can keep your program portable by avoiding the use of libraries written in other languages.

Write once, run anywhere: Because applications written in the Java programming language are compiled into machine-independent bytecodes, they run consistently on any Java platform.

Distribute software more easily: With Java Web Start software, users will be able to launch your applications with a single click of the mouse. An automatic version check at startup ensures that users are always up to date with the latest version of your software. If an update is available, the Java Web Start software will automatically update their installation.

Q. Why pointers are eliminated from Java?

1. pointers lead to confusion for a programmer
2. pointers may crash a program easily, for example, when we add two pointers, the program crashes immediately. The same thing could also happen when we for got to free the memory allotted to a variable and reallot it to some other variable
3. pointers break security. Using pointers, harmful programs like virus and other hacking programs can be developed. Because of the above reasons, pointers have been eliminated from Java.

Q. What is JIT compiler?

JIT compiler stands for Just In-Time compiler, it is the part of JVM which increases the speed of execution of a java program

Differences between C++ and Java

By the way, C++ is also an object- oriented programming language, just like Java. But there are some important feature- wise differences, between C++ and Java.

Let us have a glance at them in Table

C++	Java
C++ is not a purely object-oriented programming language, since it is possible to write C++ programs without using a class or an object	Java is purely an object-oriented programming language, since it is not possible to write a Java program without using atleast one class
Pointers are available in C++	We cannot create and use pointers in Java
de-allocating memory is the responsibility of the programmer	de-allocation of memory will be take care of by JVM
C++ has goto statement	Java does not have goto statement
Automatic casting is available in C++	In some cases, implicit casting is available . But it is advisable that the programmer should use casting wherever required
Multiple inheritance feature is available in C++	Multiple inheritance is not available in Java with classes but possible with interfaces
There are 3 access specifiers in C++ private, public, and protected	Java supports 4 access specfiers: private, public, protecteds, and default
There are constructors and destructors in C++	Only constructors are there in Java. No destructors are available

Summary:

- Java is a secured, platform-independent, multithreaded, object-oriented programming language. It is invented for developing internet applications.
- Java achieved platform independency by moving machine language generation from compilation phase to execution phase by introducing bytecodes and JVM. Bytecode in native language of JVM and JVM is a software that run java bytecodes.
- Java software is of two types JDK and JRE.
- JDK has both Compiler and JVM, so it can be used for both developing and executing new applications. JRE has only JVM, so it can be used for executing already developed applications.
- After installing JDK we must update *Path* environment variable with its *bin* folder
- A Java program can be developed by using any editor software.
- In Java we have only two extension file "*.java*" file which is called Java source code file and "*.class*" file which is called java compiled code, i.e; Java bytecode.
- Java has only 6 programming elements those are:
  package, class, interface, enum, variable and method
- *interface* is a fully unimplemented class used for declaring a set of operations of an object. *class* is a fully implemented class used for implementing object operations. enum is a final class used for defining set named constants of an menu.
- A java source file name and class name can be different.
- They must be same only if class is declared as public.
- A Java source file can have one public class and multiple non-public classes.
- If a Java source file has multiple classes its name should be public class name, else its name can be user defined name.
- Main method is not mandatory for compilation, it is mandatory only for execution.
- JVM executes only main method, it does not execute user defined methods even thought they are declared as static.
- A class member (variable or method) gets memory location only by using either static or new keyword.
- In a single source file we can have main method in every class.
- We can also execute main method from other methods of the class.
- If we call main method from its own block or from a method that is calling from main method it leads to exception "java.lang.StackOverflowError"
- The syntax to call main method is: *classname.main(new String[0]);*
  In place of ZERO we can pass any +ve integer number.
  If we pass negative number it leads to exception
  "java.lang.NegativeArraySizeException"
- Java is both platform and language.
- The Java platform is a software based platform it is both JVM + API.
- Exception we learnt in this chapter
  1. java.lang.NoClassDefFoundError
  2. java.lang.NoSuchMethodError
  3. java.lang.StackOverflowError
  4. java.lang.NegativeArraySizeException

Chapter 2

Comments, Identifiers, Keywords

- In this chapter, You will learn
  - The need of comment, types of comments, syntax rules
  - The identifier, rules in defining identifier.
  - Keywords, types of keywords, its rules.

- By the end of this chapter- you can indentify valid and invalid comments, identifiers, and keywords and also all operations we are performing in a Java application.

Interview Questions

By the end of this chapter you answer all below interview questions

Java Comments

- Definition of comment
- Why comment?
- Types of comments
- CE: unclosed comment
- CE: class or enum or interface expected
- CE: illegal start of type
- CE: illegal start of expression
- SCJP question

Identifiers

- Definition of identifier
- CE: identifier expected
- Rules in defining Identifier?
- Can we use keyword as an identifier?
- Can we use predefined class name as identifier?
- Why can't you use keyword as identifier for class, method, or variable?
- How can we differentiate user defined and predefined classes if both are defined with same name?
- What is the limitation of identifier length?
- SCJP question

Keywords

- Definition of keyword
- Keyword rules?
- What is the basic use of keyword?
- The list of operations we do in a Java file and class?
- List of keywords
- Is null a keyword?
- SCJP question

Java Comments

Definition

A description about a basic programming element is called comment.

Why comment?

Comments are meant for developer to understand the purpose.

Types of comments

Java supports 3 types of comments

1. Single Line comment - //
2. Multiline comment - /* */
3. Document comment - /** */

The statements placed inside these special characters are ignored by compiler while compiling the class. It means, compiler will not generate bytecodes for those comments. So comments are not appeared in .class file.

Below program shows usage of all above 3 comments

```
//Addition.java
/**
 * This program shows adding two integer numbers.
 * @author: Hari Krishna
 * @company: Naresh i Technologies
 * @Date: 04/13/2011
 * @Version: 1.0
 */

public class Addition
{
        /*
        This method takes two integer numbers and
                return their addition result.
        */

        public static void add(int a, int b)
        {
                //adding and returning result
                return a + b ;
        }
}
```

Find out valid comments from the below list. Valid comments means compiler should not throw compilation time error when it is appeared in Java file

```
1.      //
2.      ///////////////
3.      // /*
4.      /*
5.      /*  */
6.      /* // */
7.      // /* // */
8.      /* /* */
9.      /* */ */
10.     // /* */ */
11.     /*/* */ */
```

Identifier and its rules

Definition

Identifier is a *name* of the basic programming elements.

Find out identifiers in the below program

```
class FirstProgram
{
        public static void main (String[] args )
        {
                System. out .println ("Hi");
        }
}
```

identifier

Compile time error

If we define any basic programming element without name, compiler throws Compile time error: <identifier> expected

```
class    CE: <identifier> expected
{
        static void m1(){
                System.out.println("m1");
        }
}
```

Rules in defining identifier

While defining identifier we must follow below rules, else it leads to compile time error.

1. Identifier should only contain
   - i. Alphabets [a to z] and [A to Z]
   - ii. Digits [0 to 9]
   - iii. Special characters [_ or $]

2. Identifier should not start with a digit. A digit can be used from second character onwards

   Ex: 1stStudent ×
   No1Student ✓

3. Identifier should not contain special characters, the only '_' or '$' are allowed

   Ex: No#1Student ×
   MIN_BALANCE ✓

4. Identifier should not contain space in the middle of words. If we want to provide gap between words, they must be connected with '_'. Due to this reason '_' is called as connector symbol.

Ex: First Program ×
FirstProgram ✓
First$Program ✓
First_Program ✓

5. Identifier is case sensitive (a != A)

```
class A{
    void b(){} ✓
    void b(){} ×
    void B(){} ✓
}
```

6. Keyword cannot be used as user defined identifier, because they are available throughout JVM directly

```
class static{ ×
}
```

Note:

1. Predefined class names can be used for user defined identifier, because they are defined with separate package.
2. There is no limit in identifier length.

Q) When we cannot use keyword as user defined identifier, how can we use predefined class name as user defined identifier?

Keywords are directly available in JVM so if we use them as our user defined identifier we cannot differentiate them from predefined keywords where as predefined classes are available in packages so we can differentiate our classes from predefined class by using package name.

For example: If we create a class with the name String, we must predefined String name with package name "*java.lang.String*".

Run below test case

Create a java file with the name String.java with the class String as shown below

//String.java

```
class String{}
```

Crete a class Test with main method and compile and execute it.

//Test.java

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

Test class is compiled fine but it leads to exception "java.lang.NoSuchMethodError: main", because Java executes main method with the parameter "java.lang.String".
In this Test class main method parameter is not java.lang.String, it is our user defined class String.

To execute Test class we must use String class name with package name as "java.lang.String" check Test class in the next page.

//Test.java

```
class Test{
   public static void main(java.lang.String[] args){
         System.out.println("Test main");
   }
}
```

Now this Test class is executed and we will see output *Test main* on console

Now tell me what is the complete prototype of main method?

public static void main(java.lang.String[] args)

Q) In first chapter we have not used package name "java.lang" then how did JVM execute all those classes?

Very simple reason, in that chapter we did not create a class with name String. So compiler automatically places package "java.lang" the String parameter.

Find out valid identifiers from the below list. Valid identifier means - in program when we use below identifiers compiler should not throw error. Just apply above rules to get answer.

1. hiheliohru
2. abc1234
3. 4321cba
4. _____$$$$$$$$$
5. static
6. firstprogram
7. Class
8. main
9. String

Java Keywords and its rules

Definition

Keywords are predefined identifiers available directly throughout the JVM. They have a special meaning inside Java source code and outside of comments and Strings.

For Example: public, static, void, class etc

Rules

1. Keywords cannot be used as user defined identifier by the programmer either for variable or method or class names, because keywords are reserved for their intended use.

2. All characters in keyword must be used in lower case, because identifier is case sensitive.

Q) Is Class a keyword?

No, it is not a keyword. Its first character is in uppercase.

Need of keywords

Basically keywords are used to communicate with compiler and JVM about the operations we are performing in Java application.

In a Java file, we perform 10 different operations using keywords

They are:

01. Creating Java file (class, interface, or enum)
02. Storing data temporarily with different size of memory locations
03. Creating memory locations
04. Controlling calculations and modifications
05. Setting accessibility permissions
06. Modifying other default properties
07. Establishing relations between classes
08. Object representation
09. Grouping classes
10. Handling user mistakes

In JAVA, we have 50 keywords to perform all above 10 operations.
Among them 47 were introduced in Java 1.0

- In Java 1.2 new keyword "*strictfp*" was added
- In Java 1.4 new keyword "*assert*" was added
- In Java 5 new keyword "*enum*" was added

Among 50 keywords 2 keywords are reserved words they cannot be used in java program, because keyword is defined but they are not implemented, those are "const, goto".
Below is the list of all 50 keywords. I have divided all 50 keywords into 10 categories based on the operation they perform for easily remembering.

1. Java files (3)
1. class
2. interface
3. enum (1.5)

2. Data Types (8 + 1)
4. byte
5. short
6. int
7. long
8. flaot
9. double
10. char
11. boolean
12. void

These 8 keywords can be used as *data types* and also as *return tye*

This keyword can only be used as *return type*

3. Memory Location (2)
static
14. new

4. Control Statements (11)
1. conditional
15. if
16. else
17. swith
18. case
19. default
2. loop
20. while
21. do
22. for
3. transfer
23. break
24. continue
25. return

5. Accessibility Modifiers (3)
26. private
27. protected
28. public

6. Modifiers (8)
29. static
30. final
31. abstract
32. native
33. transient
34. volatile
35. synchronized
36. strictfp

7. Inheritance relationship
40. extends
41. implements

8. Object representation
37. this
38. super
39. instanceof

9. package
41. package
42. import

10. Exception Handling(5 + 1)
43. try
44. catch
45. finally
46. throw
47. throws
48. assert (1.4)

11. Unused keywords
49. const
50. goto

These are not keywords

Default literals

1. referenced literal
-> null
2. boolean literals
-> true
-> false

Answer below questions

Q1) Is null a keyword?

A) No, it is not a keyword. It is a literal.

Q2) Then can we use null as user defined identifier?

A) No, even though it is not a keyword we cannot use it as identifier.

Q3) How many datatype keywords Java supports? A) 8

Q4) How many types of datatypes java supports?

A) 2 types

1. Primitives types (8)
2. Referenced types (4)

Q5) How many modifiers java supports? A) 11

Q6) How many Accessibility levels java supports?

A) Four, among them 3 are keywords.

For more details check accessibility modifiers chapter.

Find out valid keywords from the below list

01. static	10. instanceOf
02. final	11. strictf
03. syncronize	12. Void
04. package	13. String
05. imported	14. dowhile
06. Public	15. Enumeration
07. main	16. null
08. object	17. finalize
09. void	18. sizeOf

Q) What is the difference between final, finally and finalize?

final and finally are keywords where as finalize is a method name.

final

is a keyword used to create constant variable, method, and class

finally

is a keyword used to define a block of statements to be executed definitely for a try block.

finalize

is a method used to define logic to be executed definitely before an object is destroyed.

The logic we write inside finally block and finalize method is called resource releasing logic or clean-up code.

Chapter 3

Working with *EditPlus* Software

- In this chapter, You will learn
  - Advantages of EditPlus
  - Installing EditPlus software
  - Creating Java file in EditPlus
  - Disabling backup file creation option
  - Configuring Compiler and JVM
  - Rule in developing Java file from EditPlus
  - How does it display compilation, execution errors and output?
  - Enabling Line Numbers
  - Increasing and decreasing fonts
  - How can it display keywords, predefined classes in colors?
  - Short cut for copy&past

- By the end of this chapter- you will feel comfortable developing Java programs using EditPlus software.

Working with *EditPlus* Software

EditPlus is an editor software that supports developing Java, C, C++, HTML, PHP, Perl programs including a pain text file. While it can serve as a good Notepad replacement, it also offers many powerful features for Web page authors and programmers.

Advantages of EditPlus

EditPlus software provides more features then notepad. It provides below features

1. It has colorful screen, It shows
    a. *keywords* in **blue** color
    b. *predefined* classes in **red** color
    c. *Strings* and *literals* in **pink** color
2. It automatically manages *code indentation*.
3. Can add ".java" extension automatically
4. We can *compile* and *execute* Java program directly from Editplus editor
5. It also has auto save option
6. Can display line numbers

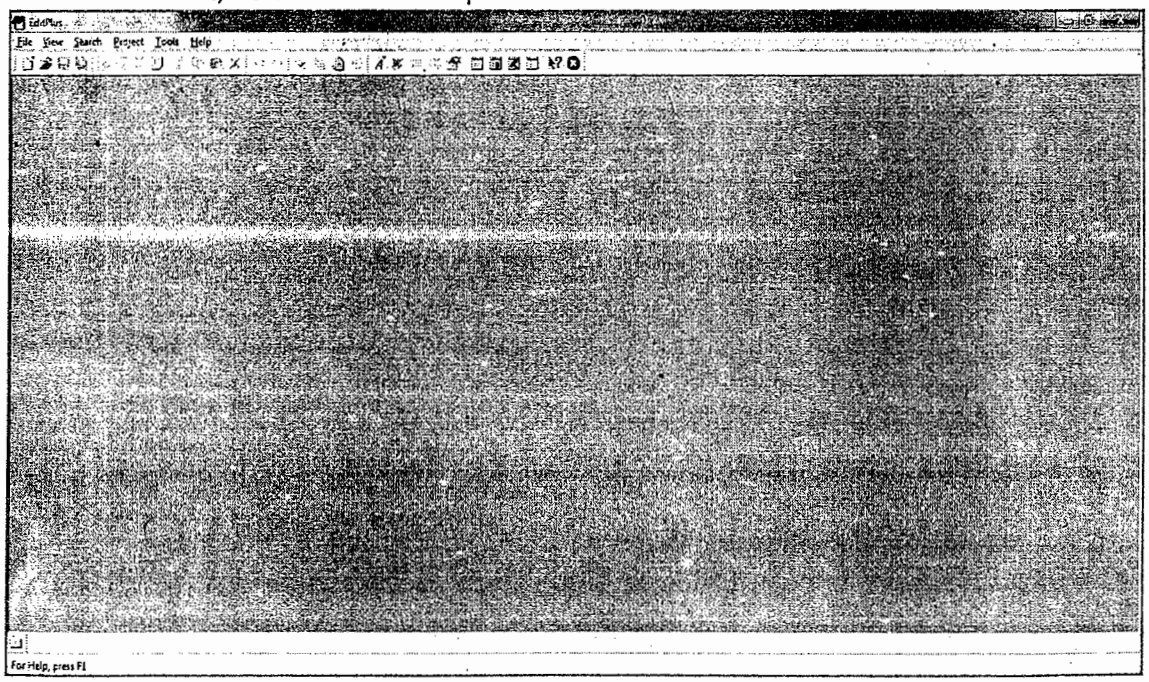
Installing *EditPlus* software

It is a trail version software you can download it from its home page "http://www.editplus.com/"

- You will be downloading "*epp331.exe*" file.
- Double click this exe file, its installation will be started.
- Click on Finish button at last window that shows Editplus installation is completed.

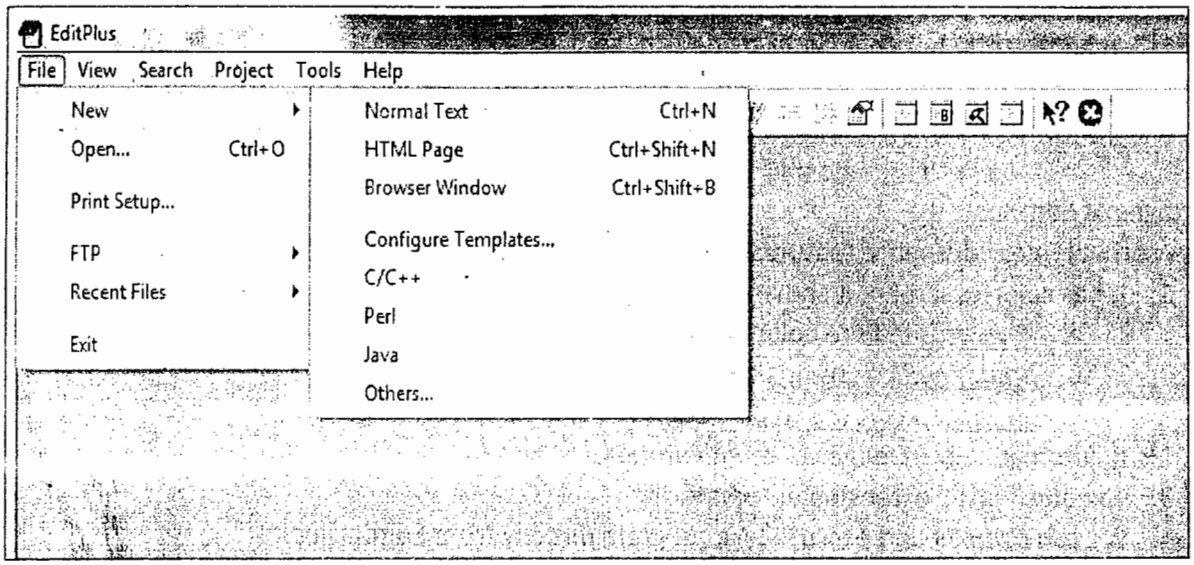
After installation you will find EditPlus shortcut image on desktop.

Double click on it, below window is opened

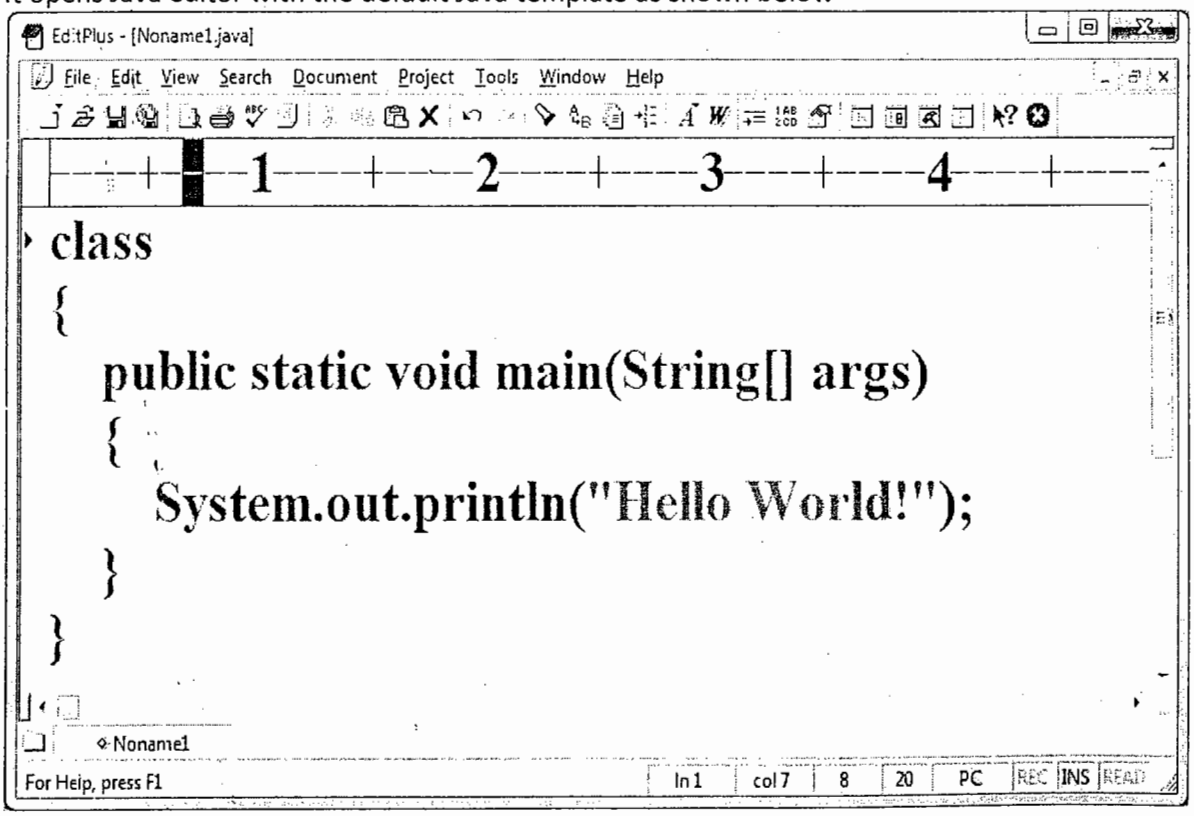


Creating a Java file in Editplus

Click on File -> new -> Java



It opens Java editor with the default Java template as shown below



Enter class name and save this file with .java extension.

Note: It takes java extension automatically.

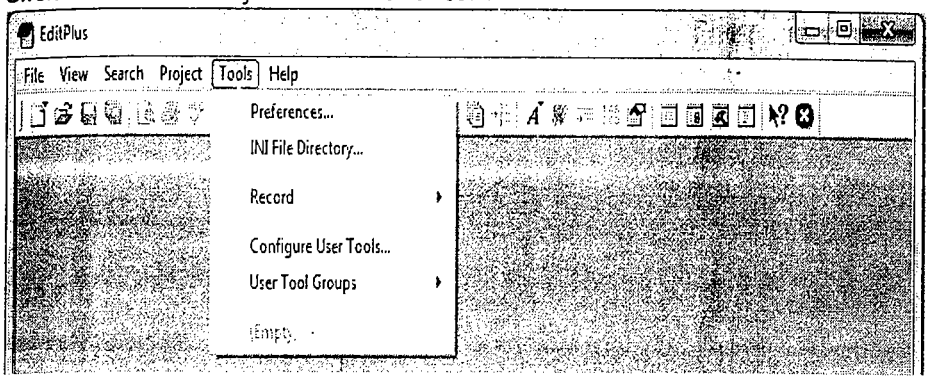
When you save this file you will get two files in PWD

1. *.java* extension file Ex: Test.java
2. *.java.bak* extension file Ex: Test.java.bak

Disabling backup file creation option

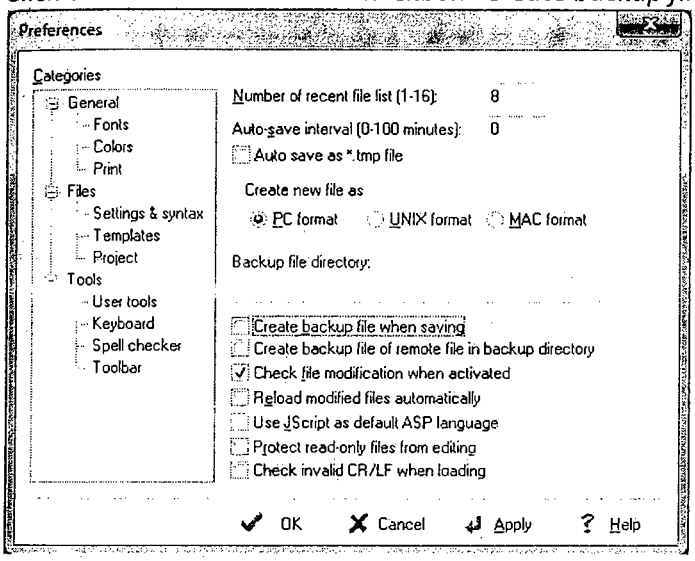
To disable back up file creation follow below steps

Click on *Tools -> Preferences* menu item



Below window is opened,

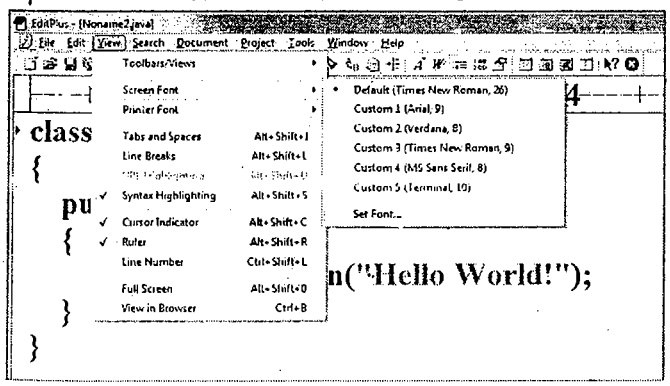
Click on *Files -> uncheck the checkbox "Create backup file when saving"*



Click Ok button to save this change.

Increasing and decreasing fonts

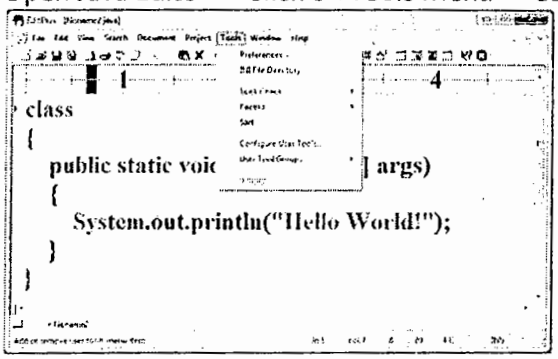
Open Java editor -> Click on *View -> Screen Font -> Set Font* menu item.



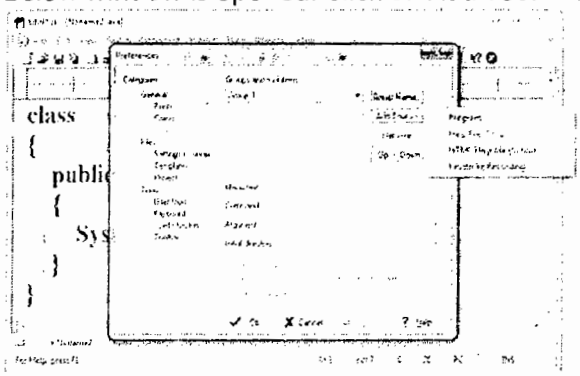
It opens a window, set your desired font -> click Ok button to save changes.

Configuring *javac* and *java* tools to compile and execute Java classes directly from Editplus:

Open Java Editor -> Click on *Tools Menu -> Configure User Tools Menu item*



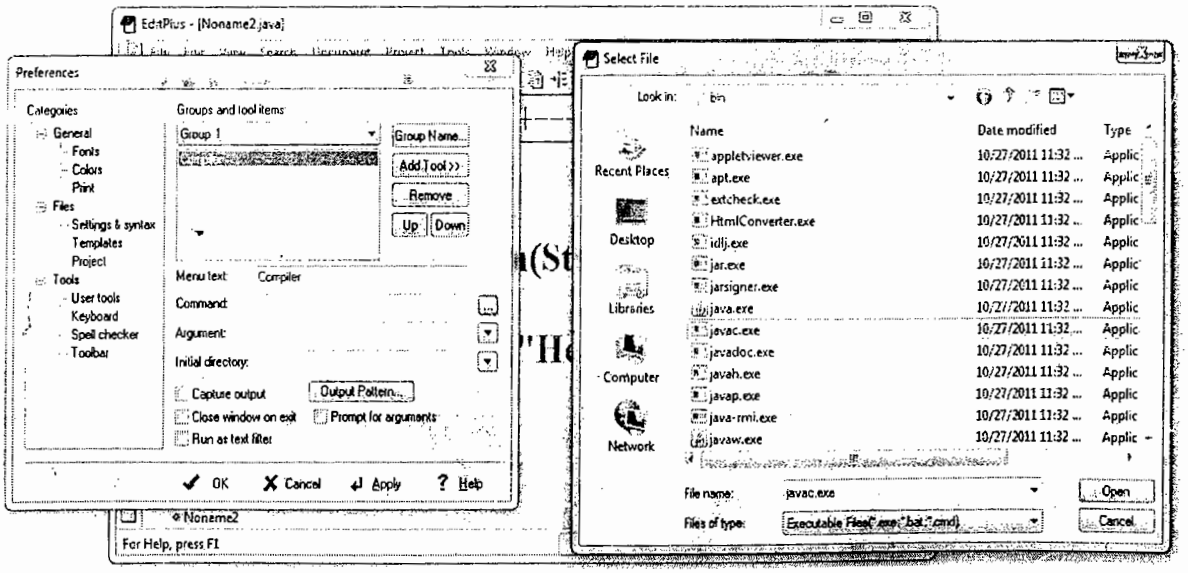
Below window is opened. Click on *Add Tool -> Program*



All above four text fields are enabled, enter text as shown below

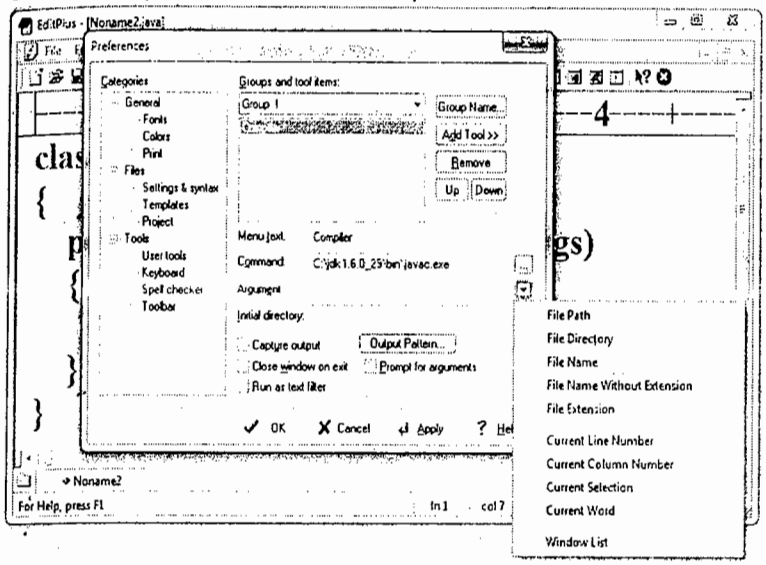
Enter **Menu Text: Compiler** (can be user defined name, it meaningful and relevant)

To select *javac* command path, click on the button placed after its textfield -> browse to C:\jdk1.6.0_29\bin folder -> select *javac.exe* -> click on Open button. Its path is stored in textfield



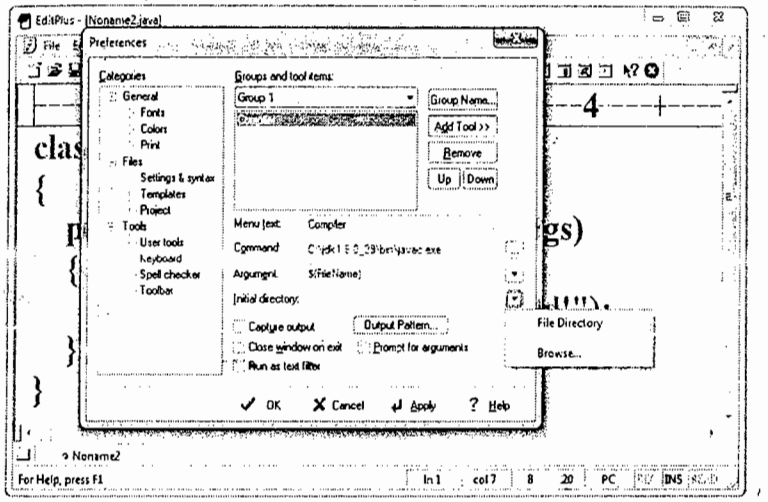
To enter **Argument** value,

Click on *dropdown* button that is placed after its textfield -> Click on *File Name*

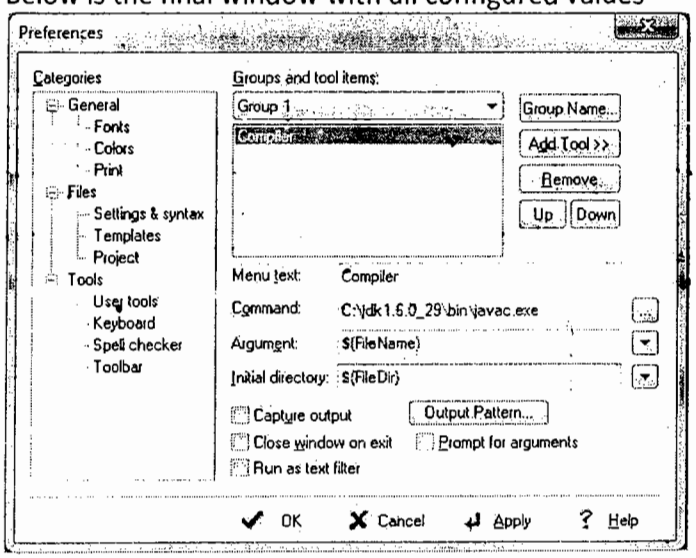


To enter **Initial directory** value

Click on *dropdown* button that is placed after its textfield -> Click on *File Directory*



Below is the final window with all configured values

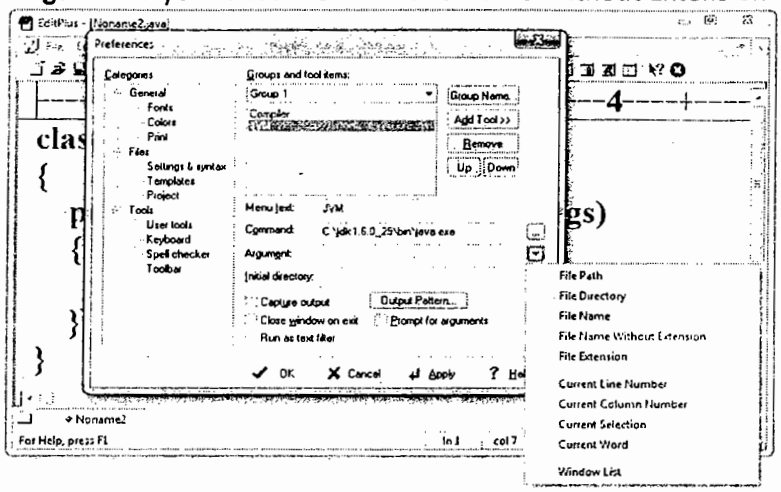


Follow the same above procedure to configure *java* tool

Menu Text: JVM

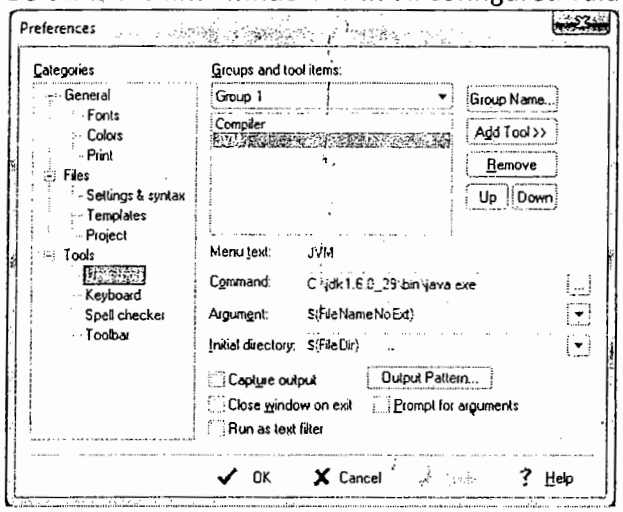
Command: C:\jdk1.6.0.29\bin\java.exe

Argument: your should choose *File Name Without Extension*



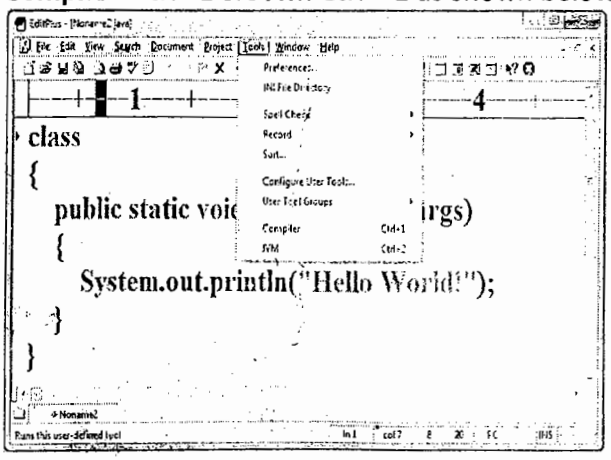
Initial Directory: File Directory

Below is the final window with all configured values of ***javac*** and ***java*** tools



Now you can find these two tools with *Menu text* name in *Tools* menu with short cuts

Compiler: Ctrl +1 & **JVM:** Ctrl + 2 as shown below



Press **Ctrl + 1** for compilation.
Press **Ctrl + 2** for execution.

Rule in developing Java file from EditPlus
To execute Java program from editplus, the Java file name must be same as class name even though class is not public. Else you will get exception *"java.lang.NoClassDefFoudError"*

Why this rule?
Recollect the configuration done for *java* tool,
We set **Argument** value as *"FileName Without Extension"*.

So editplus software executes *java* command by passing filename.

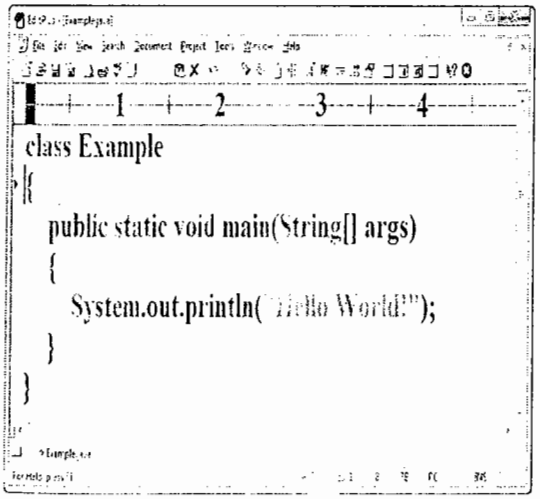
For Example
If we create class with name ***A*** and java file name with name ***Example.java.***

Editplus Software
Compiles this java file as
javac Example.java
Executes the class as
java Example

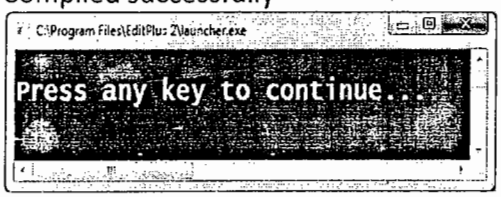
So in this case JVM throws *"java.lang.NoClassDefFoundError*: Example" as there is no ***Example.class***, instead we have ***A.class.***

Note:
This rule is common for all editor softwares those support in-build compilation and execution

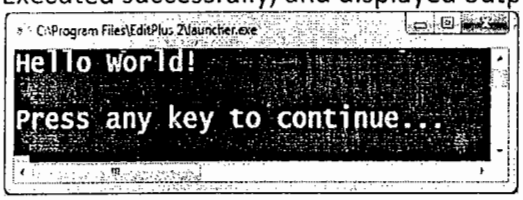
How does it display compilation, execution errors and output?
For compilation and execution it also uses command prompt. So all compilation, execution errors and output is displayed on command prompt windows as shown below.



Compiled successfully



Executed successfully, and displayed output



To enable Line number
Open Java editor -> Click on View -> Line Number menu item.
Short-Cut: Ctrl + shift + L

Q) How can it display keywords, predefined classes in blue and red colors?

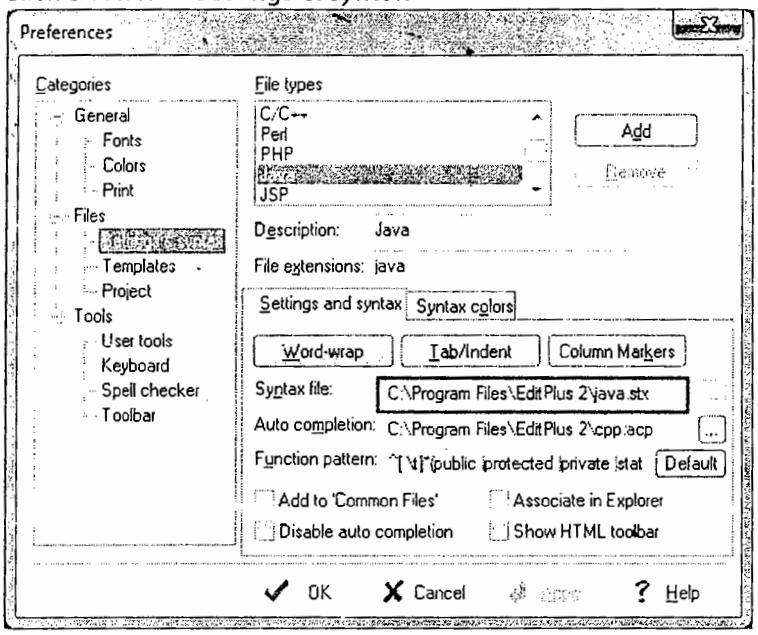
All predefined class names are saved in file, that file path is

C:\Program Files\EditPlus 2\java.stx

In Editplus, this file path is configured in the below window

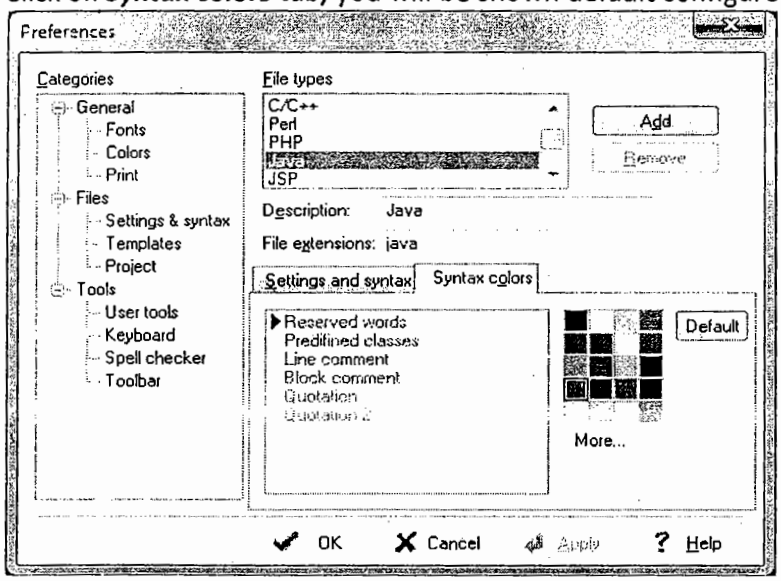
Click on *Tools -> Preferences* menu item ->

Click on *Files -> Settings & syntax*



If you want highlight any other predefined or user defined classes, you must add those class names in this file.

Click on ***Syntax colors*** tab, you will be shown default configured colors



Short- cut for copy&past existed single line:

Place cursor anywhere on the line that you want to ***copy&paste***, and then press ***Ctrl + J.***

Chapter 4

DataTypes & Literals

- In this chapter, You will learn
  - Definition and Need of datatype
  - Different datatypes
  - Storing, modifying and using different types of data
  - Limitation of variable and advantages of array and class
  - Limitation of array
  - Primitive type conversion
  - Reference type conversion
  - java.lang.ClassCastException
  - Type promotion rules
  - Working with System.out.println
  - Arithmetic operators
  - Concatenation operator
  - Equality operators
  - How Compiler and JVM Compiles and executes Java program with variable creation statements and expressions.

- By the end of this chapter- you will be in a position to compile, execute and tell right answer of different expressions output yourself without using computer and Java Software.

Interview Questions

By the end of this chapter you answer all below interview questions

Data types

- Need of Datatype
- Different types of Data
- Types of datatypes
- Definition of datatype
- Why 8 primitive types?
- What is a variable?
- Limitation of primitive types
- Why referenced types?
- Limitation of Array?
- Why class keyword and what can we do using class?
- Why referenced datatypes are called derived?
- Definition object, class, and instance?

- Primitive and reference Type conversions and its rules
  - automatic
  - casting

- Compiler and JVM thinking in type conversions
- SCJP Questions
  - CE: illegal start of expression
  - CE: cannot find symbol
  - CE: unclosed character iiteral
  - CE: empty character literal
  - CE: unexpected type
  - CE: incompatible types
  - CE: possible loss of precision
  - CE: inconvertible types
  - RE: ClassCastException

  - Special cases
    - long can be assigned to float
    - char can be assigned to number variable and int literal can be assigned to char variable
    - ? assignment
    - in char, int, String literals case compiler also checks value in remaining all literals and variables case it checks only type and range

- Type promotions in an expression
- SCJP Questions
  - CE: operator cannot be applied between

Literals

- Definition of Literal
- Types of Literals
  1. Integral Literals
  2. Floating-point literals
  3. Character Literals
  4. Boolean Literals
  5. String Literals
  6. White Space
  7. Comments

- **Write a program to create variables to store data of type**
  - integer
  - floating
  - character
  - String
- **Write a program to create custom datatypes to store**
  - Computer data
  - Student data
  - Employee data

(P.T.O)

- + operator
  - Addition operator
  - concatenation operator
- Working with *System.out.println*
- Types of languages

/ operator

- ArithmeticException
- Infinity, -Infinity
- NaN

Equality operators

- == *and*
- !=

Need of Data types

Data Types are used to store data *temporarily* in computer through a program.

In real world we have different types of data like integer, floating-point, character, boolean, and string, etc. To store all these types of data in program to perform business required calculation and validations we must use data types concept.

Definition of data type

Data type is something which gives information about

- Size of the memory location and range of data that can be accommodated inside that location.
- Possible legal operations those can be performed on that location.
  For Example: on *boolean* data we cannot perform addition operation.
- What type of result comes out from an expression when these types are used in side that expression.

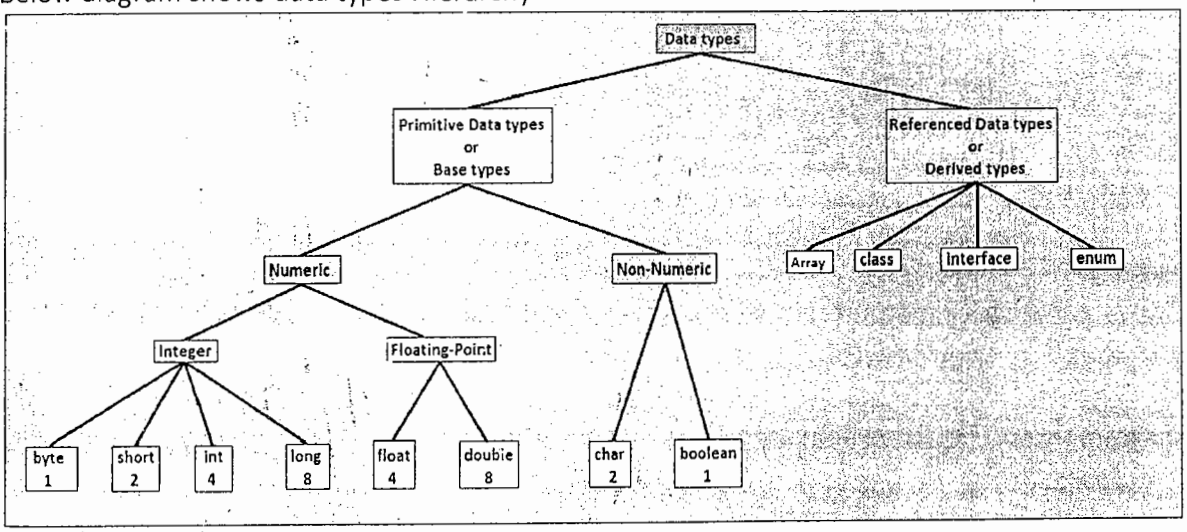
Whichever the keyword gives these semantics is treated as "data type".

Different Java data types

In Java mainly we have two types of data types

- Primitive types (8) - used to store single value at a time
- Referenced types (4) - used to collect multiple of values using primitive types.

Below diagram shows data types Hierarchy



The minimum memory location in Java is 1 byte.

Why do we have 8 primitive types in Java?

Based on *type and range* of data, primitive types are divided into 8 types.

Why do we have 4 referenced types?

To collect *same type* of values Array is given and

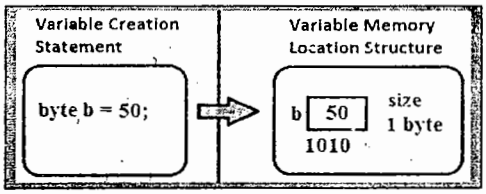
To collect *different type* of values class, interface, enum are given

What JVM does when it encounters data type inside program?

It creates memory location based on the data type size, names that ML with the given name, and stores the assigned value in that memory location as show below. This named memory location is technically called *variable*.

In the below example, we have created a variable of type **byte** with the name **b** to store value **50**, so JVM creates memory location with **size 1 byte** at the address say 1010 and **named it as b** and stores the assigned value 50 in that memory location.

Below diagram shows variable creation and its memory location structure.



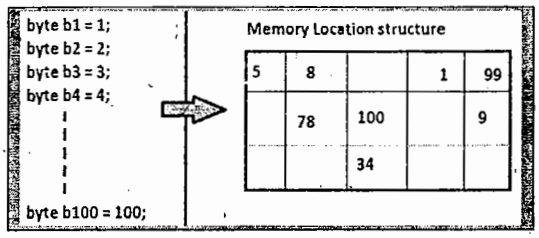
Limitations of primitive Data Types

Using primitive data types we cannot store multiple values in continuous memory locations.

Due to this limitation we face below two problems

Problem #1

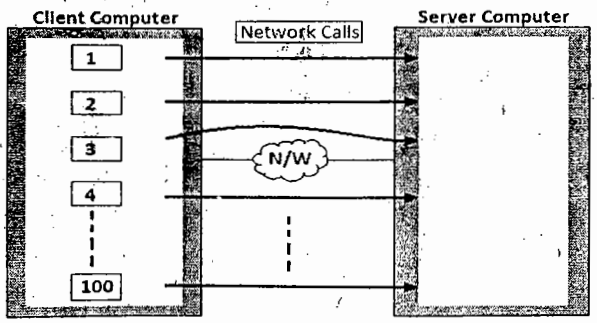
For instance if we want to store multiple values, say 1 to 100, we must create 100 variables. All those 100 variables are created across JVM at different locations as show below. Hence it takes more time to retrieve values.



Problem #2

Also using variables we cannot pass all values to the remote computer with single network call, which increases burden on network and also increase lines of code in program.

Below diagram shows the above limitation.



To pass 100 values it consumes 100 network calls.

Solution

To solve above two problems, we must group all values to send them as a single unit from one application to another application as method argument or return type. To group them as a single unit we must store them in continuous Memory Locations. This can be possible by using referenced datatypes array or class.

Why reference types are given, when we have 8 primitive types?

Referenced types are given to store multiple values in continuous memory locations to retrieve data in quick time and to pass all values with single network call.

Why four referenced types are given?

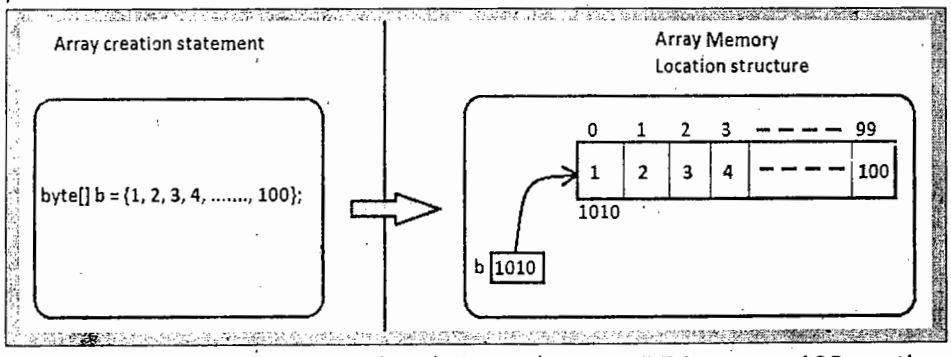
To collect only homogeneous and also heterogeneous type of values.

Understanding Array

In Java Array is a reference data type. It is used to store fixed number of multiple values *of same type* in continuous Memory locations.

Note: Like other data types Array is not a keyword rather it is a concept. It creates continuous memory locations using other primitive and reference types.

Below diagram shows array creation and its memory location structure to store 100 values of type byte.



In the above diagram array is created with 100 values. So JVM creates 100 continuous memory locations with each location of size 1 byte with some starting address, assume, 1010. Each location is created with an index starts with Zero. Finally the base address is stored in the referenced variable b to read and modify those values further.

As you noticed Memory location wise there is no difference in storing multiple values using variables and array, both consumed same size of memory in this case 100 bytes. The only difference is performance. Array, always, gives high performance than all other data types in storing multiple values.

Array Limitation:

Its size is fixed, means it will not allow us to store values more than its size. Also it will not allow different values.

Solution:

Array limitation is solved using **class**.

class

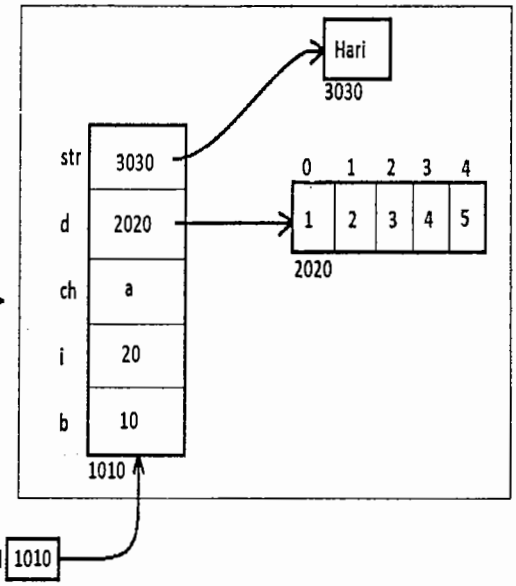
Basically using the keyword *class* SUN provides a way to create new data types to store different types of values in continuous memory location using primitive and referenced types.

Below program shows creating a user defined data type with the name "DataTypesDemo" to group different types of values.

Custom data type creation with the name DataTypesDemo

```
class DataTypesDemo
{
    // primitive type variables creation
    byte b      = 10;
    int  i      = 20;
    char ch     = 'a';

    //array type variable creation
    double[] d  = {1,2,3,4,5};

    //predefined class String type variable creation
    String str  = "Hari";

    public static void main(String[] args)
    {
        //user defined class DataTypesDemo variable creation
        DataTypesDemo dd = new DataTypesDemo();
    }
}
```

Below is the Memory Location Structure - this memory is called object



Save the above class with DataTypesDemo.java to compile and execute.

"***DataTypesDemo***" is the new datatype name, ***new*** keyword provides continuous memory locations for all variables created in that class, and that base address is stored in variable ***dd.***

So we can **define class** as - *it is a user defined data type used to store single and multiple values of same and different data type values in continuous memory locations.*

The special feature of class data type is - it also allows us to define methods to provide business logic, i.e. main logic of the application.

Why referenced types are called derived types and what is their size?

Referenced data types are called as derived data types because they are created using primitive types and its size is the addition of all primitive data type's size used in the class. Above class consumes 55 bytes of memory.

Below is the calculation

(byte + int + char + (5 * double) + 4 * char) = (1 + 4 + 2 + (5*8) + (4*2)) = 55 bytes.

Below table shows primitive data type's Size, Range and Default values

Data Type Name	Size [byte(s)]	Range	Default Value
byte	1	-128 to 127	0
short	2	-32,768 to 32,767	0
int	4	-2,147,483,648 to 2,147,483,647	0
long	8	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	0
float	4	1.40129846432481707e-45 to 3.40282346638528860e+38	0.0
double	8	4.94065645841246544e-324 to 1.79769313486231570e+308	0.0
char	2	0 to 65,535	One white space
boolean	1	false or true	false
Reference	Depends on PDT	Depends on primitive data types	null

Default values are only applicable to class level variables. Class level variables are automatically initialized by JVM with default values based on its data type but where as local variables must be initialized by developer explicitly.

Rule: So the rule is we must initialize local variables below accessing them

Q) What is the output from the below program?

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

Literals

Literal is a constant.

Ex: 10, 20.4, 'a', "abc", true

Types of literals and their default datatypes

In Java we have below types of literals

- **Integral Literals:** All integer type literals are called integral literals. By default they are of type int. If we want to represent them as long we must suffix literal with 'l' or 'L'. We do not have byte or short type literals.

- **Floating-point Literals:** All floating point literals are of type double. If we want to represent them as float we must suffix literal with 'f' or 'F'. double literals can also be suffixed with 'd' or 'D'.

- **Character literal**: The single character placed inside single quote is considered as character literal. All character literals are of type char.
  Rule: In single quote we are not allowed to place more than one character.
  In single quote we can place either single space or ONE character.

- **String literal:** Characters placed inside double quote is considered as string literal. All string literals are of type java.lang.String
  Note: In double quote we can place ZERO to 'n' number of characters.

Below table shows Types of literals, their data type and its default types

Type of Literal	Datatype	default value	Sample values
Integral	int	0	10, 20, 30, 40, ...
	long	0	10L, 20L,30L, 40L, ...
Floating-Point	float	0.0	10.0f, 20.0f, 30.34f, 40.3f, ...
	double	0.0	10.0, 20.0, 30.34, 40.3, ...
Character	char	one space	'a', 'b', 'c', '1', '@', '#', '\n', ...
Boolean	boolean	false	true, false
String	java.lang.String	null	"abc", "bbc", "a", "10", "@#$", ...

Identify valid literals from the below list

- 10
- 10.345
- 53.67f
- 2345L
- 3.45d
- 45D
- 20b
- 34s
- 'a'
- a
- '#'
- '1'
- '10'
- ''
- ""
- "abc"
- "10"
- "1"
- "a"
- hi

Primitive Data type conversion

The process of changing one type of value to another type of value is called type conversion.
We develop type conversion by assigning a value of one variable to a variable of another type.
For example:

Below example shows assigning integer value to float variable

```
int a  = 10;          a [ 10 ]

//type conversion
float f      = a;     f [ 10.0 ]
```

Primitive type conversion

We have two types of conversions, they are:

1. Implicit type conversion / Automatic type conversion / widening
2. Explicit type conversion / Casting / narrowing.

Automatic or Implicit Conversion

If STR <= DTR then that conversion is called automatic type conversion. In this conversion there is no loss of data hence compiler compiles the given class.

This conversion is also called *widening, because* if we create destination variable with highest range data type the data size is increased to destination data type size.

Below example shows implicit conversion or widdening

int a = 10;	a [10]	size 4 bytes
long l = a;	l [10]	size 8 bytes - widdening

Casting or **Explicit Conversion** or **Narrowing**

Performing type conversion from highest range datatype variable to lowest range datatype variable by using *cast operator* is called type conversion. It is also called explicit conversion because this conversion performed by developer explicitly by using *cast operator.*

Cast operator is a data type placed in parenthesis after *"= " operator* and before *source variable.*

Syntax: <Destination data type> <variable name> = *(data type)* <source type>;

By using cast operation

- We are convincing compiler that the value stored in source type variable is within the range of cast operator type and
- We are allowing JVM to reduce source value to cast operator type if its range is greater than cast operator type

Sometimes it is necessary to perform type conversion between highest range data type variable to lowest range datatype variable, in this case developer must do conversion explicitly by using cast operator as shown below.

Below example shows casting

```
long  L  = 10;
int   i  = L;
//casting
int   i  = (int)L;
```

✓ L [10] size 8 bytes

✗ CE: possible loss of precession

✓ i [10] size 4 bytes - Narrowing

Here we are telling to compiler that the value is stored in L is with in the range of int, so compiler allows this conversion as it assumes there is no loss of data.

Casting is also called *narrowing* because data size is decreased.

In casting, what will happen if source variable has value greater than destination type range?
No CE, No RE, assignment is performed by reducing its value to the cast operator range by using 2's compliment and stores that result in the destination variable.

We can use below **short-cut formula** to know the reduced value

[minRange + (result - maxRange - 1)]

Below program shows applying casting short-cut formula

```
int    i = 254;          i  [254]
byte   b1 = (byte) i;    b1 [-2]
```

```
minRange + (result - maxRange - 1)
=> -128 + (254 - 127 -1);
=> -128 + (254 - 128)
=> -128 + (126)
=> -2
```

Rules in Primitive Type Conversion

Rule #1: Source and destination data types must be compatible; otherwise it leads to compile time error "**incompatible types**". Except boolean all primitive data types are compatible. It means boolean value or variable cannot be assigned to any other data type variable.

Below example shows **incompatible types** error

```
int   a  = 10;      ✓
float f  = a;       ✓
//boolean b = a;    ✗  CE: incompatible types
                         found  : int
                         required: boolean
```

Rule #2: Source type range must be less than or equals to destination type range, otherwise it leads to CE: "**possible loss of precision**"

Below example shows **possible loss of precision** error

```
float  f1   = 10;   ✓
float  f2   = f1;   ✓
double d1   = f1;   ✓
int    i    = f1;   ✗  CE: possible loss of precision
                           found  : float
                           required: int
```

Rule on cast operator**:**

- *cast operator* data type must be compatible with source type else it leads to CE: "**inconvertible types**" and also
- It should be compatible with destination type else it leads to CE: "**incompatible types**" and also
- its range must be <= destination type range else it leads to CE: "**possible loss of precession**".

Find compilation errors in the below program

```
int  i    = 10 ;
byte b1   = i ;
byte b2   = (byte) i;
byte b3   = (int)  i;
byte b4   = (boolean) i;
```

Self practice bits

Find out compilation errors if any

1. **Automatic / implicit / widdening conversion**
2. **Casting / explicit/ narrowing conversion**

Ex:

```
int a = 10;

float f = a;

boolean bo = a;
```

Ex:

```
int a = 10;

int b = a;

//widdening
long l = a;

//narrowing
byte b = (byte)a;
```

a	10	4 bytes
b	10	4 bytes
l	10	8 bytes
b	10	1 byte

```
int a = 10;
byte b = a;
byte b = (byte)a;
boolean  bo = a;
boolean  bo = (boolean)a;
boolean  bo = (byte)a;
byte b = (short) a;
short s = (byte)a;
byte b = (short)(byte)a;
byte b = (byte)(short)a;
```

```
int a = 254;

byte b1 = (byte) a;
short s1 = (short) a;

short s2 = (short)(byte) a;

System.out.println(a);
System.out.println(b1);
System.out.println(s1);
System.out.println(s2);
```

Special cases

Case #1: Long value can be assigned to float variable, but float variable cannot be assigned to long variable, because below two reasons

- float range is greater that logn range, check ranges table
- floating point value cannot be stored in long variable.

Find out CEs in the below assignments

```
long L = 10;
int i =L;
float f =L;
float f1= 10;
long b = f1;
long L2 = (long)f1;
```

```
float f1 = 254.345f;
byte b1 = (byte) f1;

System.out.println(f1);
System.out.println(b1);
```

Case #2: char and number types are compatible types
So we can assign

- char literal to number variable
- int literal to char variable

In the above assignment JVM performs required conversions
If we assign char literal to number variable, JVM stores that character's ASCII number
If we assign number literal to char variable, JVM stores that number's ASCII character

Below is the list of characters and their ASCII numbers

Character	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
ASCII Number	48	49	50	51	52	53	54	55	56	57

Character	'A'	'B'	'C'	.....	.....	....	....	....	....	'Z'
ASCII Number	65	66	67	.....	.....	....	....	....	....	90

Character	'a'	'b'	'c'	.....	.....	....	....	....	....	'z'
ASCII Number	97	98	99	.....	.....	....	....	....	....	122

Java supports Unicode character set, it is a super set of ASCII character set.

- Unicode character set range is "0 to 65535" and
- ASCII character set range is "0 to 255"

So, the rule is: The number we are assigning to char variable must be in range of 0 to 65535, else it leads to CE: *possible loss of precision*.

What is the output from the below program?

```
char ch1 = 'a';        System.out.println(ch1);
char ch2 = 97;         System.out.println(ch2);

int i1 = 98;           System.out.println(i1);
int i2 = 'b';          System.out.println(i2);

char ch3 = 255;
char ch4 = 65000;
char ch5 = 65535;
char ch6 = 66000;
```

The special character "?"

- If the assigned number's corresponing character is not supporting by your computer JVM internally stores the special character "?"
- By default Windows OS supports only ASCII character set, so if we assign number value beyond 0 – 255 range JVM internally stores the "?" character

What is the out from the below program?

```
char ch1 = 98;
char ch2 = 250;

System.out.println(ch1);
System.out.println(ch2);
```

Q) We know we can assign int literal to char variable, the same way can we assign *int variable* to char variable?

No, because int datatype range is greater than char datatype range

What is the output from the below program?

```
int i1 = 98
char ch2 = i1;

char ch1 = 98;
```

But we can assign int variable to char variable through cast operator

What is the output from the below program?

```
int i1 = 98
char ch2 = i1;
char ch2 = (char)i1;
```

Q) Also think, can we assign char variable to byte, short variables, and byte variables to char variable?

- We can assign char literal to byte, short variables if the assigned char literal ASCII number is within the range of byte or short datatypes.
- But we cannot assign char variable to byte or short variables also we cannot assign byte or short variables to char variable, because their range has negative number.

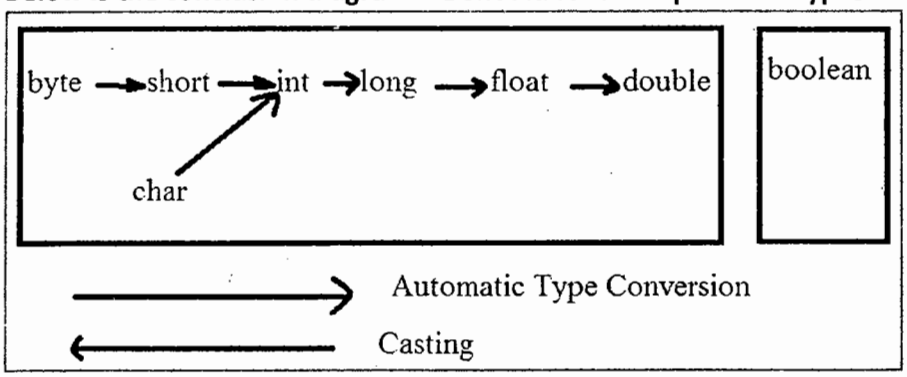
What is the output from the below program?

```
byte b = 97;
char ch = b;  CE: possible loss of precision

char ch = (char)b;
System.out.println(ch);  //a
```

Find out compile time errors in the below program, comment them and print value of remaining variables.

```
char ch1 = 'a';              System.out.println(ch1);
char ch2 = '1';              System.out.println(ch2);
char ch3 = '10';             System.out.println(ch3);
char ch4 = 97;               System.out.println(ch4);
char ch5 = 49;               System.out.println(ch5);
char ch6 = 1;                System.out.println(ch6);
char ch7 = 255;              System.out.println(ch7);
char ch8 = 65000;            System.out.println(ch8);
char ch9 = 65535;            System.out.println(ch9);
char ch10 = 66000;           System.out.println(ch10);
char ch11 = -97;             System.out.println(ch11);

int i1 = 97;                 System.out.println(i1);
int i2 = 'a';                System.out.println(i2);

char ch12 = i1;              System.out.println(ch12);
char ch13 = (char)i1;        System.out.println(ch13);

int i3 = -97;                System.out.println(i3);
char ch14 = i3;              System.out.println(ch14);
char ch15 = (char)i3;        System.out.println(ch15);
float  f = 'a';              System.out.println(f);
```

Below is the conclusion diagram we should follow in primitive types variables assignment



Q) What type of destination variable we should create to store all primitive types of values, except boolean?

A) double type variable.

Q) In how many ways we can initialize a variable?

There are 4 ways to initialize a variable

1. **Using literal** -> int i = 10;
2. **Using another variable** -> int j = i;
3. **Using expression** -> int k = i + j;
4. **Using non-void method** -> int x = m1();

Case #3:

In all above four ways of assignment compiler *checks only source data type and its range but not its value*, but in case of *int and char literals assignment it checks type, range and also its value*. If the value is within the range of destination type variable compiler allows assignment else it throws PLP error.

Find out compilation errors in the below assignments

```
byte b = 10;
short s = 300;
char ch = 108;

int i = 10;
int i = 10L;

float f = 10;
float f = 10L;
float f = 10.0;
float f = 10.0f;

double d = 10.0;
double d = 10.0f;

double d = 10L;
double d = 'a';
double d = 10;
```

Answer below question

```
1. byte b1 = 10;
2. int i = 10;
3. byte b2 = i;

4. Sopln(b1);
5. Sopln(i);
6. Sopln(b2);
```

choose one option

1. CE at line 1
2. CE at line 3
3. 10 10 10
4. CE at line 6
5. none of the above

Reference Types conversion

Like primitive types, we can also perform type conversion operation in between reference types. Assigning one class object to another class referenced variable is called referenced type conversion. To perform referenced type conversion the two classes should be compatible. The classes become compatible only if they are developed with inheritance. Inheritance relation is also called IS-A relation.

So the rule we should check in reference type conversion is:

Source type *IS-A* destination types or not.

How can we develop inheritance relation between two types?

By using either *extends* or *implements* keywords

"extends" is used for developing inheritance between two classes or two interfaces, and
"implements" is used for developing inheritance between interface, class.

Syntax:

```
class Example{}                    interface A{}
class Sample extends Example{}     interface B extends A{}
                                   class C implements A{}
```

The class that is *placed after extends keyword* is called *super class*, here Example is super class, and the class that is *placed before extends keyword* is called *sub class*, here Sample is sub class. The classes created *without inheritance* relationship are called *siblings*

Q) Which are the classes called compatible?

- Subclass is compatible with Superclass
- Super class is not compatible with subclass
- Siblings are not compatible.

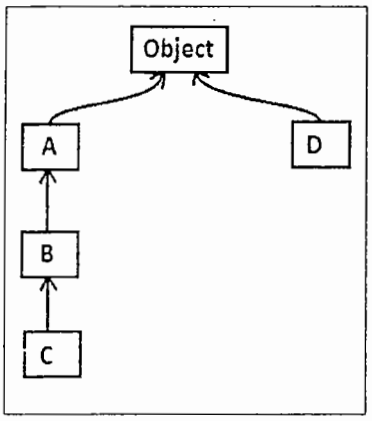
Q) What is the right assignment between referenced types?

The rule we should check in referenced type conversion is *Source Type* "IS-A" *Destination Type* or not. So we can assign subclass object reference to a superclass referenced variable. But super class object reference cannot be assigned to a subclass referenced variable. Also sibling object reference cannot be assigned to another sibling class referenced variable.

Q) What type of referenced variable we must create to store all types of objects?

A) java.lang.Object class type, because it is the super class of all types of classes.

Consider below inheritance hierarchy, and identify relations between these classes



- ➡ Object class is the super class of all user defined and predefined classes
- ➡ class A "IS-A" subclass of Object
- ➡ class B "IS-A" subclass of A, Object
- ➡ class C "IS-A" subclass of B, A, Object
- ➡ class D "IS-A" subclass of Object, and is sibling of A, B, C

So, we can store

- ➡ A class object reference in Object class refrenced variable
- ➡ B class object reference in A or Object class refrenced variable
- ➡ C class object reference in B or A or Object class refrenced variable
- ➡ D class object reference in Object class reference

```
/*
Below program shows implementing
above inheritance hierarchy.
*/

class A{}
class B extends A{}
class C extends B{}
class D{}
```

Identify compile time errors in the below assignments?

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

        Object obj1 = new Object();
        Object obj2 = new A();
        Object obj3 = new B();
        Object obj4 = new C();
        Object obj5 = new D();

        A a1 = new A();          B b1 = new A();
        A a2 = new B();          B b2 = new B();
        A a3 = new C();          B b3 = new C();
        A a4 = new D();          B b4 = new D();

   }
}
```

Types of referenced type conversion:

Java supports two types of reference type conversions

- Upcasting / automatic conversion
- Downcasting / casting

Upcasting: It is the implicit reference type conversion.

The process of *storing subclass object reference in super class reference* variable is called upcasting.

For Example:

```
A a = new B();
```

Note: It is not possible to store super class object in sub class reference variable, it leads to **"incompatible types"** compile time error.

For Example:

```
A a = new A();      ✓

B  b = a;           ✗ CE: incompatible types
B  b = new A();     ✗ CE: incompatible types
```

Downcasting: it is the explicit reference type conversion, casting.

Retrieving subclass object reference from super class referenced variable and storing it in the same sub class referenced variable is called downcasting.

For Example:

```
A a = new B();      ✓
B b = (A) a;        ✓
```

Here, in casting we are informing to compiler that the object stored in **e** variable is **B** type object. Hence compiler allows compiling the program as they are having IS-A relation.

Rule in using cast operator:

The cast operator type and source type should have inheritance relation else it leads to Compile time error **"inconvertible types"**

For Example:

```
A a = new A();
D d = (D) a;      ✗ CE: inconvertible types
```

java.lang.ClassCastException:

In casting the object coming from source variable, if it is not compatible with cast operator type JVM throws above exception "*ClassCastException*".

In casting compiler cannot identify the object coming from the source variable, because it checks only source variable type and cast operator type has IS-A relation or not. So when the source variable is superclass type and cast operator is subclass type compiler always compiles

this casting. But if the object coming from the referenced variable is sibling type of cast operator type then JVM throws ClassCastException

Below code throws CCE

```
Object obj    = new A();      ✓
D      d      = (D) obj;      ✗RE: ClassCastException
```

In the above statement **D d = (D)obj;** compiler only checks **obj** and **D** has inheritance relation or not, it does not check the **object stored** in **obj** variable because compiler checks only type of the variable. Since both types have inheritance relation compiler allows above conversion. But at runtime JVM identifies the **object coming** from **obj** variable is of type **A** which is **not compatible** with **D**, hence JVM terminates program execution by throwing **ClassCastException.**

Q) What Compiler and JVM do in reference type conversion?

Compiler checks the source variable type & cast operator type has inheritance relation or not.

In the above example, compiler

- first checks obj type "IS-A" D => No
- then it checks D "IS-A" obj type => Yes

Then it compiles this casting statement

JVM checks the source type object "IS-A" cast operator type or not.

In the above example source type object is "A" and cast operator type is "D", they are siblings so JVM throws CCE.

How can we solve ClassCastException?

To solve CCE exception we should use **instanceof** operator.

It returns *boolean* value by checking source type object with the given class. It works exactly as like cast operator, the only difference is for siblings comparison *cast operator* throws ClassCastException where as *instance operator* returns false.

So to solve CCE before downcasting we should check object type using **instanceof** operator. Below is the syntax to use instanceof operator

Syntax: **referenced variable** instanceof **classname**

Here *referenced variable* is the source variable and *class name* is the cast operator type name It returns true, if referenced variable contains object **IS-A** class type. Else returns false.

Below code shows doing downcasting with instanceof operator condition

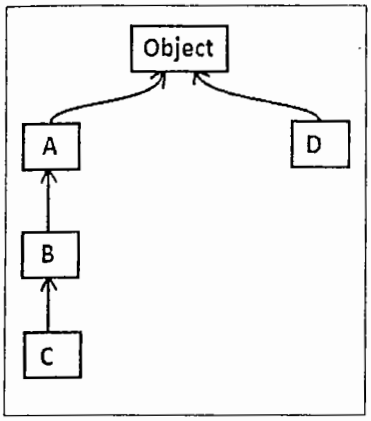
```
Object obj    = new A();
if(obj instanceof D){
        D  d    = (D) obj;
}
```

Now we do not get CCE. JVM does not execute casting since instanceof operator returns false because the source variable object is A.

Rule of instanceof operator:

Source variable type and *classname* type should have IS-A relation else compiler throws CE: inconvertible types. If the object coming from referenced variable is same class or subclass object then instanceof operator returns true. If it is super class or sibling type object instanceof operator returns false,

Consider below inheritance hierarchy, and Find out what is the output in the below statements?




```
A a = new B();
System.out.println( a instanceof Object);
System.out.println( a instanceof A);
System.out.println( a instanceof B);
System.out.println( a instanceof C);
System.out.println( a instanceof D);

Object obj = new B();
System.out.println( a instanceof Object);
System.out.println( a instanceof A);
System.out.println( a instanceof B);
System.out.println( a instanceof C);
System.out.println( a instanceof D);
```

When we should implement upcasting and downcasting in Project:

Need of upcasting

In projects upcasting is implemented to develop loosely coupled runtime polymorphic applications means to store all subclasses objects into a single referenced variable and further to execute the invoked method from different subclasses based on the object stored in the referenced variable. You will understand more about upcasting in OOPS chapter

Limitation in Upcasting

In upcasting compiler allows us to invoke only superclass members, we cannot invoke subclass specific members with superclass type variable, it leads to **CE: cannot find symbol**, because compiler checks only variable type but not object stored in superclass reference variable.

Check below example

```
class Example{
        void m1(){
                Sopln("m1");
        }
}

class Sample extends Example{
        void m2(){
                Sopln("m2");
        }
}
```

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

                Example e = new Sample();
                e.m1();   ✓
                e.m2();   ✗CE: c f s
        }
}
```

In this example, e.m2() method call leads to CE because m2() method definition is not available in superclass Example.

Need of downcasting

We should implement downcasting to invoke subclass specific members, because when we store subclass object in superclass referenced variable we cannot invoke that subclass specific member using super class referenced variable. In the above example in Test class we must implement downcasting to call m2() method from Sample class as shown below:

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

                Example e = new Sample();
                e.m1();

                Sample s = (Sample)obj;
                e.m2();
        }
}
```

SCJP Question:

```
Given:
10. interface Foo {}
11. class Alpha implements Foo {}
12. class Beta extends Alpha {}
13. class Delta extends Beta {
14. public static void main( String[] args ) {
15. Beta x = new Beta();
16. // insert code here
17. }
18. }
Which code, inserted at line 16, will cause a java.lang.ClassCastException?
A. Alpha a = x;
B. Foo f = (Delta)x;
C. Foo f = (Alpha)x;
D. Beta b = (Beta)(Alpha)x;
```

Project code – need of upcasting

In projects objects are created and passed as arguments at run-time, these objects are of different types, like Person, Animal, Vehicle, Shape etc... We process all these objects in a single method by taking them as argument. To make sure that we are reading and processing only a particular type of objects we must define method with parameter of type which can hold only that particular type of objects (super class type).

In this case Upcasting should be implemented; it means we should take method parameter of type which is a super class of a group of sub classes.

What is the output from the below programs?

```
class Example{
        void m1(){
                Sopln("m1");
        }
}
class Sample extends Example{
        void m2(){
                Sopln("m2");
        }
}
```

```
class Test{
        static void m3(Example e){
                e.m1();
                e.m2();
        }
}
```

```
class Test{
    static void m3(Example e){
        e.m1();

        Sample s = (Sample)e;
        s.m2();
    }

    public static void main(String[] args){
        Test.m3(new Sample());

        Test.m3(new Example());
    }
}
```

```
class Test{
    static void m3(Example e){
        e.m1();

        if(e instanceof Sample){
            Sample s = (Sample)e;
            s.m2();
        }
    }
    public static void main(String[] args){
        Test.m3(new Sample());

        Test.m3(new Example());
    }
}
```

```
class Test{
  static void m3(Object obj){
        if (obj instanceof Example)
        {
            Example e = (Example)obj;
            e.m1();

        }
        else if (obj instanceof Sample)
        {
            Sample s = (Sample)obj;
            s.m1();
            s.m2();
        }
  }
```

```
    public static void main(String[] args)
    {
        Test.m3(new Example());

        Test.m3(new Sample());

        Test.m3(new Object());
    }
}
```

Automatic Type Promotions in an Expression

If an expression has arithmetic operators with different data types, its result type is the highest range data type used in that expression.

JVM follows below standard rules in evaluating an expression

1. It replaces all variables and method calls with their values
2. It promotes all lesser range data type values to the highest range variable type that is used in that expression, then it starts calculation

   For example:

```
int     i1 = 10;
float   f1 = 20;

float   f2 = i1 + f1;
```

The above expression result type is float. So the destination variable type should be either float or its next range data type.

Q) Can we store the above expression value in int variable?

No, it leads to CE: possible loss of precision

```
int     i2 = i1 + f1;  ╳ CE:  possible loss of precision
```

We can store this result in int variable with cast operator as show below

```
int     i2 = (int) (i1 + f1);
```

Q) Check if below statements are compiled fine?

```
int     i2 = (int) i1 + f1;
int     i2 =  i1 + (int)f1;
```

Rule #1: In an expression both operands must be compatible types else it leads to compile time error: "**operator cannot be applied .....**"

Ex:

```
int       i1  = 10;
boolean bo = true;

int     i2  = i1 + bo;  ╳ CE: operator + cannot be applied between int, boolean
```

Rule #2: In an expression "byte, short, and char" variables are automatically promoted to int datatype, because the minimum memory location used to calculate an expression is 4 bytes. For more details check JVM architecture chapter.

Ex: below statement leads to Compilation error

```
byte b1= 50;
byte b2= 100;
byte b3= b1+ b2; ╳ CE: possible loss of precision
                       found: int
                       required: byte
```

We should use cast operator to store the result in byte variable, as shown below

```
byte b3 = (byte) (b1 + b2);
```

Self practice bits:

Rule: All operands in an expression must be compatible, else it leads to CE: operator can not be applied .....

JVM performs below steps in evaluating an expression

It replaces

1. all variables with thier values.
2. all method calls with their returned values
3. promotes lowest range data type values to highest range data type values that is used in that expression.

ex:

case 1:

```
int a = 10;
int b = 20;

int c = a + b;
```

same types so no promotion

case2:

```
int a = 20;

float f = 30;

float f = a + f;
```

=> 20 + 30.0
=> 20.0 + 30.0
=> 50.0

int is promotted to float

case 3:

```
int a = 50;
blooean b = true;

int c = a + b;
```

CE: operator + cannot be applied to int,boolean
incompatible types, So CE

In an expression "byte, short, and char" are automatically promotted to int.
Why? this is the secret of JVM, check JVM architecture.

case1:

```
1. byte b1 = 10;
2. byte b2 = 20;
3. byte b3 = b1 + b2 ;

4. Sopln(b3);

byte b3 = b1 + b2 ;
=> byte b3 = 10 + 20 ;
=> byte b3 = int + int ;
=> byte b3 = int  ;
```

case2:

```
1. short s1 = 10;
2. short s2 = 20;
3. short s3 = s1 + s2 ;

4. Sopln(s3);

=>short s3 = s1 + s2 ;
=>short s3 = 10 + 20 ;
=>short s3 = int + int ;
=>short s3 = int ;
```

case3:

```
1. char ch1 = 'a';
2. char ch2 = 'b';
3. char ch3 = ch1 + ch2 ;

4. Sopln(ch3);

=>char ch3 = ch1 + ch2 ;
=>char ch3 = 'a' + 'b' ;
=>char ch3 = 97 + 98 ;
=>char ch3 = int + int ;
=>char ch3 = int ;
```

Note: Compiler does not check values range in an expression, it checks only type.
Hence all above cases expression assignment leads to CE because those expressions results **int** type value.

hence compiler think it can not stored in lesser range variable types.

Conditions apply

it is true only if expression contains variables, if it contains int literals directly, it evalues expression and checks the result is with in the range of destination type. if that reslut is with in the range of desination type,assignment is possible else it throws CE: PLP

Ex:

```
byte b3 = 10 + 20;

byte b3 = 126 +1;

byte b3 = 126 +2;
```

```
byte b1 =  10L + 20;
byte b2 =  (byte)10L + 20;
byte b3 =  (int)10L + 20;
```

```
char ch1 = 'a' + 'b';
char ch2 =  ch1 + 'c';
```

Working with *System.out.println*

Using *System.out.println* statement we can print all types of literals

- Directly
- Using variable
- Using expression
- Using non-void methods

Write program to add two integer numbers and print the result as

The addition of 10 and 20 is 30

```
class Addition{
        public static void main(String[] args){
                int a = 10;
                int b = 20;
                int c = a  + b;
                System.out.println(c);
                System.out.println("The addition of " + a +" and " + b + " is " + c);
        }
}
```

Evaluation process of above statement

=> System.out.println("The addition of " + 10 +" and " + 20 + " is " + 30);
=> System.out.println("The addition of " + "10" +" and " + 20 + " is " + 30);
=> System.out.println("The addition of 10" +" and " + 20 + " is " + 30);
=> System.out.println("The addition of 10 and " + 20 + " is " + 30);
=> System.out.println("The addition of 10 and " + "20" + " is " + 30);
=> System.out.println("The addition of 10 and 20" + " is " + 30);
=> System.out.println("The addition of 10 and 20 is " + 30);
=> System.out.println("The addition of 10 and 20 is " + "30");
=> System.out.println("The addition of 10 and 20 is 30");

Arithmetic Operators

Java supports 5 Arithmetic operators

1. Addition +
2. Subtraction -
3. Multiplication *
4. Division /
5. Reminder %

In Java all operators in an expression are executed from LEFT to RIGHT means from "=" operator to ";" according to their precedence.

For operators precedence order check Operators chapter

Arithmetic Operators precedence

- *, /, % operators have highest and same precedence
- +, - operators have next and same and precedence

What is the output from the below expression

System.out.println (4 * 2 + 8 / 2 – 3 * 3 + 4 / 2);

Evaluation:

- ⇨ System.out.println (4 * 2 + 8 / 2 – 3 * 3 + 6 / 3);
- ⇨ System.out.println (8 + 8 / 2 – 3 * 3 + 6 / 3);
- ⇨ System.out.println (8 + 4 – 3 * 3 + 6 / 3);
- ⇨ System.out.println (8 + 4 – 9 + 6 / 3);
- ⇨ System.out.println (8 + 4 – 9 + 2);
- ⇨ System.out.println (12 – 9 + 2);
- ⇨ System.out.println (3 + 2);
- ⇨ System.out.println (5);

Addition "+" Operator: A overloaded operator

In Java, + is the only overloaded operator.
It can be used as both

1. Addition operator
2. Concatenation operator

If + operator has it's both operands as numbers, it acts as an addition operator.
If one of the operand is String, it acts as concatenation operator

For Example

int i1 = 10 + 20; <= in this expression it act as *addition* operator
System.out.println(i1); => **30**

String s = "a"+"b"; <= in this expression it act as *concatenation* operator
System.out.println(s); => **ab**

Concatenation means appending both operands as single value

String s = "a" + 10; <= in this expression it act as *concatenation* operator
System.out.println(s); => **a10**

Division "/" Operator

Just calculate and tell what is the output from the below statement

System.out.println(22/7 * 10 * 10);

Your favorite expression (**PI * r^2**), answer is **3.14** right☺ It is ***wrong answer***,
It is Java mathematics, not your general mathematics, and Answer is ***300.***

$\frac{22}{7} = 3$ because $\frac{\text{int}}{\text{int}} = \text{int}$ So, result of above expression is **300**

Some other important point of / operator

1. We cannot divide an integer number by ZERO, it leads to RE: ***java.lang.ArithmeticException***, not Infinity
2. But we can divide a floating point number by ZERO, its output is **Infinity**
3. We cannot divide a Integer zero by a ZERO it also leads to RE: *ArithmeticException*
4. We can divide a floating point ZERO by ZERO its output is **NaN**

Answer Below questions, find out if there are any CEs.

```
//TypeConversion.java
class TypeConversion{
        public static void main(String[] args) {
        byte b1 = 10;
        int i1 = b1;

        byte b2 = i1;
        byte b2 = (byte)i1;

        int i = true;

        int i = (int)true;

        int i2 = 254;
        byte b3 = i2;
        byte b3 = (byte)i2;

        char ch1 = 'a';
        int i3 = ch1;

        int i4 = 97;
        char ch2 = i4;
        char ch2 = (char)i4;

        long l1 = 10;
        float f1 = l1;

        long l2 = f1;
        long l2 = (long)f1;

        System.out.println("b1: "+b1);
        System.out.println("i1: "+i1);
        System.out.println("b2: "+b2);
        System.out.println("b3: "+b3);
        System.out.println("ch1: "+ch1);
        System.out.println("i3: "+i3);
        System.out.println("ch2: "+ch2);
        System.out.println("i4: "+i4);
        System.out.println("l1: "+l1);
        System.out.println("f1: "+f1);
        System.out.println("l2: "+l2);
    }
}
```

```
// ThinkAsCompilerAndJVM.java
class ThinkAsCompilerAndJVM {
        public static void main(String[] args) {

        System.out.println( 10        );
        System.out.println( 'a'       );
        System.out.println( "a"       );
        System.out.println( 10.0      );
        System.out.println( 10.345f   );
        System.out.println( 30L       );
        System.out.println( 30l       );
        System.out.println( 50 + 20   );
        int a = 30;   int b = 40;
        System.out.println( a + b     );
        System.out.println( "a + b"   );

        System.out.println( "a + b"+a+b      );
        System.out.println( "a + b"+ ( a+b)  );

        System.out.println( "a - b"+ a – b    );
        System.out.println( "a - b"+ ( a – b ) );

        System.out.println( "a * b"+a*b      );
        System.out.println( "a * b"+ ( a*b )  );

        System.out.println( ""+10 + 20        );
        System.out.println( 10 + ""+ 20       );
        System.out.println( 10 + 20 +""       );

        System.out.println( 22/7 * 10 * 10   );
        System.out.println( 22F/7 * 10 * 10);
        System.out.println( 22D/7 * 10 * 10);

        System.out.println( 10 / 0    );
        System.out.println( 10.0 / 0  );
        System.out.println( -10.0 / 0 );

        System.out.println( 0 / 0     );
        System.out.println( 0.0 / 0   );
        System.out.println( -0.0 / 0  );

    }
}
```

Equality Operators

We have two equality operators to compare two values

1. **== equals**
2. **!= Not Equals**

Both operators compare values of the variables.
If both variables have *same values* == operator returns *true, and* != operator returns *false*
If both variables have *different values* == operator returns *false,* != operator returns *true*

For example

```
int a = 50;
int b = 50;
System.out.println( a == b );
System.out.println( a != b );
System.out.println( a = b);
System.out.println( a = b == b);
System.out.println( (a = b) == b);
```

Q) What is the output from below comparison?

```
System.out.println ( 10 == 10.0 );
```

A) **true**
Reason: int value 10 promoted to double 10.0

Special case:
Comparing *float* value with *double* value

We get true only if we compare round floating point "float" and "double" values. In remaining all other floating point float and double values comparison returns false.

```
System.out.println( 3.0f == 3.0 ); //true
System.out.println( 3.5f == 3.5 ); //true

System.out.println( 3.3f == 3.3 ); //false
System.out.println( 3.7f == 3.7 ); //false
```

Types of Languages based on types conversion

Based on the types conversion / typing concept the entire available languages can be broadly classified in to three categories

1. **Weakly typed**
   - The languages which allow us to assign big range variable value to small range variable without any restriction up to greater extent are called weakly typed programming languages.
   - Such types of languages are "C, C++", and most other high-level languages.
   - For example:
     - We can assign a "float" variable to "int" variable.
     - But we can't assign "structure" variable to "int" variable.

2. **Strongly/strictly typed**
   - In this type of languages we are not allowed to perform the operations like above. It leads to compile time error "possible loss of precision".
   - In compilation phase compiler of this type of languages does type checking thoroughly.
   - Such type of language is "JAVA".
     - for example:
       - We can't assign a "float" variable to an "int" variable directly.

- Before doing type conversion in java, java compiler checks below semantics:
  - *Every variable and expression should have a type, and every type is strictly defined,* else leads to CE.
  - All assignments, whether explicit or via parameter passing in method calls, re checked for type compatibility.
  - Java compiler checks all expressions and parameters to ensure that the types are compatible.
  - Any type mismatches are errors. That must be corrected to finish the compiler compiling the class.

3. **Untyped:**
   - The languages which do not use primitive variables to store data, instead if they used objects to store data, are called **untyped** programming languages.
   - These languages are also called as pure object-oriented programming languages.
     - Ex: Smalltalk
   - In these languages everything is an object.

Is Java 100% pure object-oriented programming language?
It is 99.9999999% object-oriented, remaining 0.00000001% not object-oriented because it uses primitive types to store data.

In Java 5 this limitation is solved by introducing a concept called
Auto Boxing and Auto Unboxing.

So from Java 5 onwards we can declare *JAVA also pure* object oriented programming language.

By using this concept we can avoid using primitive variables for storing data, instead we can use objects directly. But still primitive variables support is also available to provide backward compatibility.

Using auto boxing and unboxing we can perform calculation as shown below

```
Integer io1 = 10;
Integer io2 = 20;
Integer io3 = io1 + io2;
System.out.println("Result: "+io3);
```

Note: If you use jdk1.4 compiler, this code error out at compilation.

Check next chapter for details on Autoboxing and unboxing

Chapter 5

Wrapper Classes

with Auto Boxing and Unboxing

- In this chapter, You will learn
  - Definition and need of Wrapper Classes
  - Types of Wrapper Classes
  - Different types of conversions
  - *java.lang.NumberFormatException*
  - Special case with Boolean class
  - Character class special methods
  - Casting in wrapper classes
  - Wrapper Classes comparison
  - Auto Boxing and Unboxing
- By the end of this chapter- you will be comfortable in working with wrapper classes.

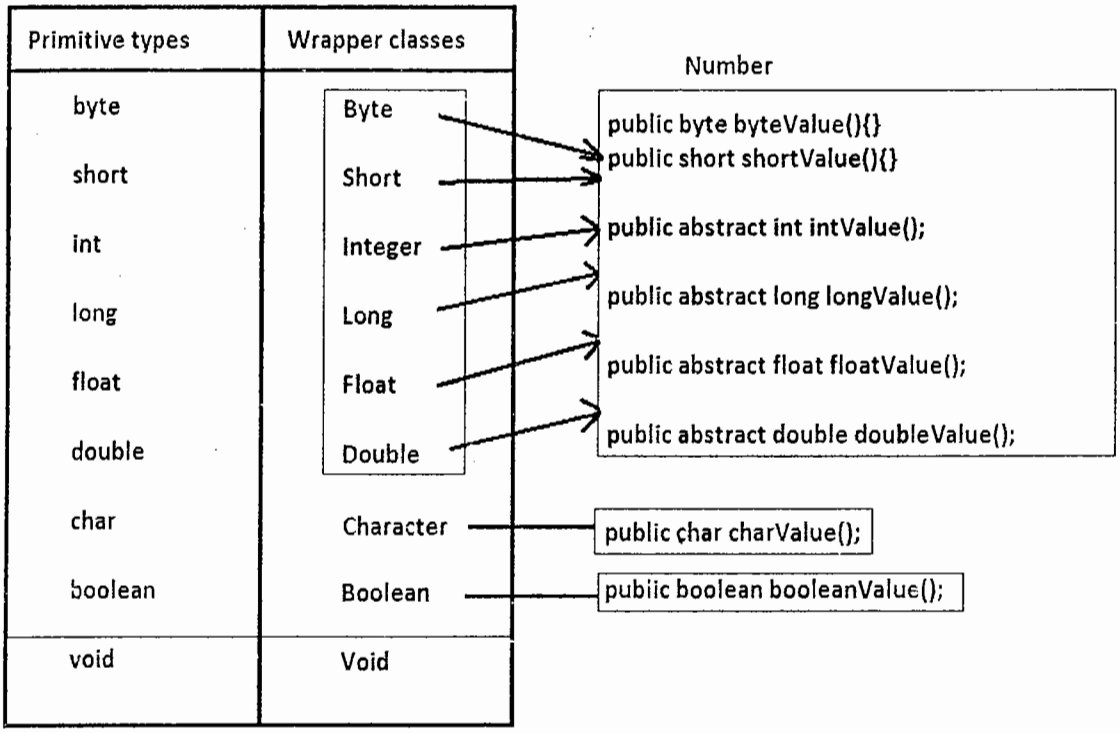
Interview Questions

By the end of this chapter you answer all below interview questions

- Definition of Wrapper classes
- Need of wrapper classes
- Types of wrapper classes
- Different conversions can done using wrapper classes
- Common wrapper class constructors
- java.lang.NumberFormatException
- Wrapper classes casting
- Wrapper class comparison, equality, hashCode
- Character Wrapper class special methods to operate characters.
- Autoboxing and unboxing

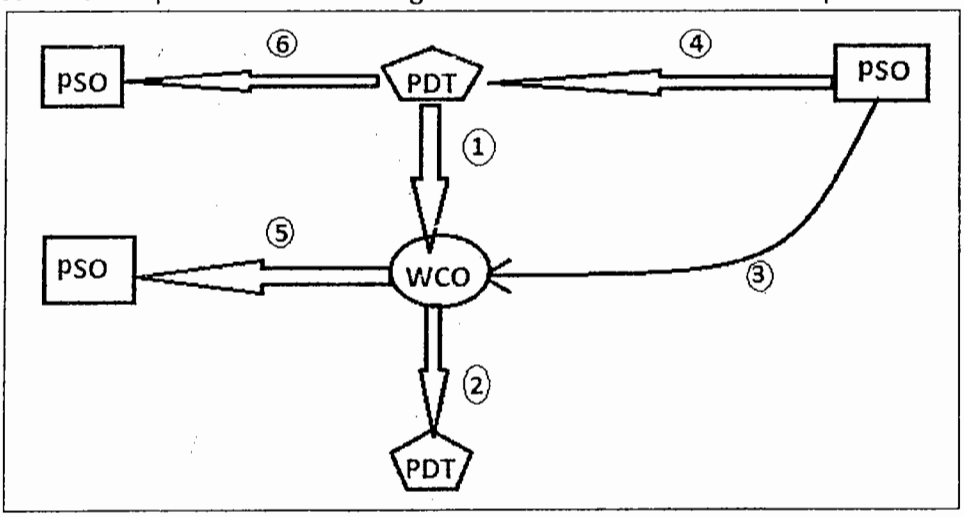
The classes which are used to represent primitive values as object are called wrapper classes. In *java.lang* package we have 8 wrapper classes one per each primitive type to represent them as object. Among them 6 wrapper classes represent number type values these wrapper classes are called number wrapper classes. These 6 number wrapper classes are subclasses of a class ***Number*** which is an abstract class. This class has 6 xxxValue() method to read primitive value from wrapper class object. Except *byteValue()* and *shortValue()* remaining all 4 methods are abstract methods so it is abstract class. These 6 methods are implemented in all 6 number wrapper classes by returning the current object's internal primitive value.

Below diagram shows primitive types and their wrapper classes



Need of Wrapper classes

Basically wrapper classes are used in project to perform conversion operations. We have 6 conversion operations. Below diagram shows these 6 conversion operations



Wrapper classes constructors and methods

To perform above 6 conversions every wrapper class has the required number of constructors and methods. Check API documentation for more details.

Below diagram show all constructors and methods list

	Constructor(PDT) valueOf(PDT) (static method)	Constructor(S) valueOf(S) (static method)	xxxValue()	parseXxx(S) (static method)	toString(PDT) (static method)	toString()
Byte	✓	✓	✓ 6	✓	✓	✓
Short	✓	✓	✓ 6	✓	✓	✓
Integer	✓	✓	✓ 6	✓	✓	✓
Long	✓	✓	✓ 6	✓	✓	✓
Float	✓ + Float(double)	✓	✓ 6	✓	✓	✓
Double	✓	✓	✓ 6	✓	✓	✓
Character	✓	✗	✓ 1	✗	✓	✓
Boolean	✓	✓	✓ 1	✓	✓	✓

Below program shows performing all above 6 conversions

```
class WrapperClassesDemo {

    public static void main(String[] args) {

        //1. PDT => WCO conversion
        int i = 10;
        Integer io1 = new Integer(i);
        Integer io2 = Integer.valueOf(i);

        System.out.println("i: "+i);
        System.out.println("io1: "+io1);
        System.out.println("io2: "+io2);
```

Find out compile time errors in the below lines of code:

```
Byte b1 = new Byte(50);
Byte b2 = new Byte((byte)50);
Character ch1 = new Character('a');
Character ch2 = new Character(97);
Character ch3 = new Character((char)97);
Boolean bo = new Boolean(true);
Float f1 = new Float(70);
Float f2 = new Float(32.56f);
Float f3 = new Float(50.45);
Double d1 = new Double(67.54);
Double d2 = new Double(98);
Double do = new Double('a');
```

```
//2. WCO => PDT conversion
Integer io3     = Integer.valueOf(254);
int     x       = io3.intValue();
byte    b       = io3.byteValue();
short   s       = io3.shortValue();
float   f       = io3.floatValue();

System.out.println("x: "+x);
System.out.println("b: "+b);
System.out.println("s: "+s);
System.out.println("f: "+f);
System.out.println("ch: "+ch);
```

Find out compile time errors in the below lines of code

```
char    ch1             = io3.charValue();
char    ch2             = io3.intValue();
char    ch3             = (char)io3.intValue();

boolean         bo1     = io3.booleanValue();
boolean         bo2     = io3.intValue();
boolean         bo3     = (boolean)io3.intValue();
```

```
//3. PSO => WCO conversion
Integer io1 = new Integer("10");
Integer io2 = Integer.valueOf("1");

Byte bo1 = Byte.valueOf("1");
//Byte bo2 = Byte.valueOf("128"); RE: java.lang.NumberFormatException: Value
                                                                out of range.

//Integer io3 = Integer.valueOf("a");  RE: java.lang.NumberFormatException: For
                                                                input string: "a"
//Integer io3 = new Integer("5.4");    RE: java.lang.NumberFormatException: For
                                                                input string: "5.4"
//Integer io3 = new Integer("5L");     RE: java.lang.NumberFormatException: For
                                                                input string: "5L"

Float fo1 = new Float("5");
Float fo2 = new Float("5.4");
Float fo3 = new Float("567.432F");
System.out.println("io1: "+io1);
System.out.println("io2: "+io2);
System.out.println("bo1: "+bo1);
System.out.println("fo1: "+fo1);
System.out.println("fo2: "+fo2);
System.out.println("fo3: "+fo3);
```

```
//boolean  String Object => Boolean WCO
//"false" | "true" => WCO

Boolean bo1 = new Boolean("false");        System.out.println("bo1: "+bo1);
Boolean bo2 = new Boolean("true");         System.out.println("bo2: "+bo2);

Boolean bo3 = Boolean.valueOf("false");    System.out.println("bo3: "+bo3);
Boolean bo4 = Boolean.valueOf("true");     System.out.println("bo4: "+bo4);

Boolean bo5 = Boolean.valueOf("True");     System.out.println("bo5: "+bo5);
Boolean bo6 = Boolean.valueOf("TrUe");     System.out.println("bo6: "+bo6);
Boolean bo7 = Boolean.valueOf("FALSE");    System.out.println("bo7: "+bo7);

Boolean bo8 = Boolean.valueOf("FASLE");    System.out.println("bo8: "+bo8);
Boolean bo9   = Boolean.valueOf("TURE");   System.out.println("bo9: "+bo9);
Boolean bo10  = Boolean.valueOf("Hari");   System.out.println("bo10: "+bo10);
Boolean bo11  = Boolean.valueOf("");       System.out.println("bo11: "+bo11);
Boolean bo12  = Boolean.valueOf(null);     System.out.println("bo12: "+bo12);

Integer io13    = Integer.valueOf(null);   System.out.println("io13: "+io13);


//4. PSO => PDT conversion
//1. PSO => WCO => PDT
//2. PSO => PDT

int i1 = Integer.parseInt("10");
//int i2 = Integer.parseInt("10.0"); //RE: java.lang.NumberFormatException: For
                                                        input string: "10.0"

//byte b1 = Byte.parseInt("40"); CE: c f s
byte b1 = Byte.parseByte("40");
//byte b2 = Byte.parseByte("128"); //java.lang.NumberFormatException: Value
                                                                  out of range.

float f1 = Float .parseFloat("10");
float f2 = Float .parseFloat("50.456");
float f3 = Float .parseFloat("606.678F");

boolean bo1 = Boolean.parseBoolean("TRUE");
boolean bo2 = Boolean.parseBoolean("FALSE");
boolean bo3 = Boolean.parseBoolean("Hari");
boolean bo4 = Boolean.parseBoolean("TURE");
```

```
System.out.println("i1: "+ i1);
System.out.println("b1: "+ b1);
System.out.println("f1: "+ f1);
System.out.println("f2: "+ f2);
System.out.println("f3: "+ f3);
System.out.println("bo1: "+ bo1);
System.out.println("bo2: "+ bo2);
System.out.println("bo3: "+ bo3);
System.out.println("bo4: "+ bo4);

//5. WCO => String object conversion
Integer io = new Integer(299);
System.out.println(io);
System.out.println(io.toString());

//6. PDT => NSO
//String s1 = 10;        CE: incompatible types
String s1 = "10";
String s2 = Integer.toString(10);

//String s3 = Byte.toString(10); //CE: c f s
String s3 = Byte.toString((byte)10);

String s4 = Integer.toString('a');

//String s5 = Integer.toString("a"); //CE: c f s
//String s5 = Integer.toString(10.0);//CE: c f s

String s5 = Float.toString(20);
String s6 = Float.toString(30L);
String s7 = Float.toString(40.0f);
String s8 = Float.toString(50.0F);
//String s9 = Float.toString(60.0);      //CE: c f s

String s9 = Boolean.toString(false);
String s10 = Boolean.toString(true);
//String s11 = Boolean.toString(TRUE); CE: c f s variable TRUE
System.out.println("s1: "+ s1);
System.out.println("s2: "+ s2);
System.out.println("s3: "+ s3);
System.out.println("s4: "+ s4);
System.out.println("s5: "+ s5);
System.out.println("s6: "+ s6);
System.out.println("s7: "+ s7);
System.out.println("s8: "+ s8);
System.out.println("s9: "+ s9);
System.out.println("s10: "+ s10);
    }
}
```

WrapperClassesComparision

```
class WrapperClassesComparision {
	public static void main(String[] args) {
		int i1 = 10;
		int i2 = 10;

		System.out.println(i1 == i2);
		//System.out.println(i1.equals(i2));//CE: int cannot be dereferenced

		Integer io1 = new Integer(10);
		Integer io2 = new Integer(10);

		System.out.println(io1 == io2);
		System.out.println(io1.equals(io2));

		//Wrapper classes type conversion
		/*Wrapper classes are not compatible to each other, because they are siblings.
		If we use "==" operator to compare their objects it leads to CE: incomparable
		types, but we can compare them using equals() method, it returns false => No
		CE or No RE.*/

		Double do1 = new Double(10.0);
		//System.out.println(io1 == do1); CE: incomparable types: java.lang.Integer and
								java.lang.Double
		System.out.println(io1.equals(do1)); //false

		double d1 = 10.0;
		System.out.println(i1 == d1); //true

		//=> System.out.println(10 == 10.0);
		//=> System.out.println(10.0 == 10.0);
	}
}
```

Auto Boxing and Unboxing

- Converting primitive type to wrapper class object automatically is called Auto Boxing
- Converting wrapper class object to primitive type automatically is called Auto Unboxing

So as per Autoboxing and unboxing we can assign primitive value to wrapper class referenced variable and wrapper class objects to primitive variable directly. Then the required conversion is done automatically by compiler.

For example below code is correct as per Java 5 compiler and above

```
Integer io = 50;
int i = new Integer(50);
```

Q) Who does Autoboxing and unboxing is it Compiler or JVM?

Compiler does auto boxing and unboxing based on the primitive literal. This feature required code is added in compiler software. So JVM does not know about this feature.

Auto boxing

Let us understand how compiler does Auto boxing in the below statement

```
Integer io = 50;
```

In the above line, compiler converts *int literal* 50 to *Integer object* as shown below

```
Integer io = Integer.valueOf(50);
```

So, in "*.class*" file we do not have 50 as int literal we have it as Integer object.
So, JVM process the value 10 as Integer object.

Q) On what basis compiler converts primitive values to wrapper class object?

Based on the type of primitive value it converts it into its associated wrapper class object.
For example, *int* is converted to *Integer*, *float* to *Float*, *char* to *Character*, *boolean* to *Boolean*.

Check below lines of code:

//AB.java *DWC*

```
class AB {
   public static void main(String[] args) {
        Byte      b    = 40;
        Short     s    = 50;
        Integer   i    = 60;
        Long      L    = 70L;
        Float     f    = 80F;
        Double    d    = 90D;
        Character ch   = 'a';
        Boolean   bo   = true;
   }
}
```

//AB.class *CCC*

```
class AB {
   public static void main(String[] args) {
      Byte    b  = Byte.valueOf((byte)40);
      Short   s  = Short.valueOf((short)50);
      Integer i  = Integer.valueOf(60);
      Long    L  = Long.valueOf(70L);
      Float   f  = Float.valueOf(80F);
      Double d = Double.valueOf(90D);
      Character ch= Character.valueOf('a');
      Boolean bo = Boolean.valueOf(true);
   }
}
```

Identify Compile time errors in the blow lines of code
Do you remember wrapper class objects are not compatible?

```
byte    b1    = 40;          Byte    bo1    = 40;
byte    b2    = 128;         Byte    bo2    = 128;
Int     i     = 'a';         Integer io     = 'a';
long    L     = 50;          Long    Lo     = 50;
```

Auto Unboxing
Let us understand how compiler does Auto Unboxing in the below statement

```
int i = new Integer(50);
```

In the above line, compiler converts *Integer object* 50 to *int value 50* as shown below

```
int i = new Integer(50).intValue();
```

So, in "*.class*" file we do not have new Integer(50) as Integer object, since intValue() method is called on this Integer object its value 50 is returned and stored in "i" variable as int value.
So, JVM process the value 50 as int value.

Q) On what basis compiler converts wrapper class object to primitive values?
Based on the type of wrapper class object compiler internally calls *xxxValue()* method on the wrapper class object to retrieve primitive value from this wrapper class object and converts it into its associated primitive type.
For example, *on Integer object it calls intValue()*, on *Float* object *floatValue()*, on *Character* object *charValue()*, on *Boolean* object *booleanValue()*.

Check below lines of code:

//AUB.java *DWC*

```
class AUB {
   public static void main(String[] args){
      byte    b  = new Byte((byte)40);
      short   s  = new Short((short)50);
      int  i     = new Integer(60);
      long    L  = new Long(70);
      float   f  = new Float(80);
      double  d  = new Double(90);
      char    ch = new Character('a');
      boolean bo = new Boolean(true);
   }
}
```

//AUB.class *CCC*

```
class AUB {
   public static void main(String[] args) {
      byte  b = (new Byte((byte)40)).byteValue();
      short s = (new Short((short)50)).shortValue();
      int   i = (new Integer(60)).intValue();
      long  l = (new Long(70L)).longValue();
      float f = (new Float(80F)).floatValue();
      double d = (new Double(90D)).doubleValue();
      char  c = (new Character('a')).charValue();
      boolean b =
                (new Boolean(true)).booleanValue();
   }
}
```

Identify Compile time errors in the blow lines of code

Do you remember primitive types are compatible except boolean?

byte b1 = 50;
int i1 = b1;
 	
int i2 = 50;
byte b2 = i2;
byte b3 = (byte)i2;
 	
int i3 = 'a';
 	
double d1 = 50;
double d2 = 60L;
double d3 = 70.34f
double d4 = 30.45;
 	
double d5 = true;
	byte b1 = new Integer(50);
	int i1 = new Byte(b1);
	
	int i2 = new Integer(50);
	byte b2 = new Integer(i2);
	byte b3 = (byte)new Integer(i2);
	
	int i3 = new Character('a');
	
	double d1 = new Integer(50);
	double d2 = new Long(60L);
	double d3 = new Float(70.34f);
	double d4 = new Double(30.45);
	
	double d5 = new Boolen(true);

Identify compile time errors in the below program?

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

                Integer io1 = new Integer(10);
                Integer io2 = 10;

                int a = new Integer(10);
                int b = io2;

                Double d1 = 10;

                Integer io3 = 'a';

                Byte b1 = 10;
                Byte b2 = 128;

                Character ch1 = 97;
                Character ch2 = (Character)97;

                Double d2 = 40.43;
                Integer io4 = d2;
                int c = d2;

        }
}
```

Write a program to add two integer numbers without using primitive types

Addition.java

```
class Addition {
        public static void main(String[] args) {
                Integer io1 = 50;
                Integer io2 = 60;
                Integer io3 = io1 + io2;

                System.out.println("Result: "+ io3);
        }
}
```

Addition.class

```
class Addition {
        public static void main(String[] args) {
                Integer io1 = Integer.valueOf(50);
                Integer io2 = Integer.valueOf(60);
                Integer io3 = Integer.valueOf( io1.intValue() + io2.iintValue() );

                System.out.println("Result: "+ io3);
        }
}
```

From Java 5 onwards we can also define switch with wrapper class variable

```
class ABUBWithSwitch {
  static void m1(Integer io){
        switch(io){
                case 1:
                        System.out.println("1");
                        break;
                case 2:
                        System.out.println("2");
                        break;

                default:
                        System.out.println("other");
        }
  }
```

```
        public static void main(String[] args) {
                m1(1);
                m1(2);
                m1(3);
                m1(-1);
                m1(null);
        }
}
```

Very important point: If we create wrapper class objects with byte range same value, only object is created and all referenced variables are pointing to same object. check below code

```
Integer io1 = 50;                          Integer io3 = 150;
Integer io2 = 50;                          Integer io4 = 150;
System.out.println( io1 == io2); -> true   System.out.println( io3 == io4); -> false
```

Calling methods by passing primitive type and wrapper class object

Method calling by passing primitive type

We can call a primitive type parameter method by passing either the same primitive type or its wrapper class object. *Note that* – we call method by passing the same primitive type or also can call by passing its lesser range primitive type value or its associated wrapper class.

Identify compile time errors in the below program

```
class MethodwithPDT {
        static void m1(int a){
                System.out.println("int-arg: "+a);
        }

        public static void main(String[] args) {

                m1( (byte)50 );                m1( new Byte( (byte)50 ) );
                m1( 'a' );                     m1( new Character( 'a' ) );
                m1(  60 );                     m1( new Integer( 60 )  );
                m1( 70L );                     m1( new Long( 70L ));
                m1( 80.45 );                   m1( new Double( 80.45 ));
        }
}
```

Method call by passing wrapper object

We can call a wrapper class parameter method by passing either the same wrapper class object or its matched primitive type value. *Note that* – wrapper classes are not compatible so we cannot call wrapper class parameter method by passing the other wrapper class object or other primitive type values it leads to compile time error.

Identify compile time errors in the below program

```
class MethodwithWC {
        static void m1(Integer io){
                System.out.println("Integer-arg: "+io);
        }

        public static void main(String[] args) {

                m1( (byte)50 );                m1( new Byte( (byte)50 ) );
                m1( 'a' );                     m1( new Character( 'a' ) );
                m1(  60 );                     m1( new Integer( 60 )  );
                m1( 70L );                     m1( new Long( 70L ));
                m1( 80.45 );                   m1( new Double( 80.45 ));
        }
}
```

Write a program to add two numbers by reading them from command line

Addition.java

```
class Addition{
        public static void main(String[] args) {
                int a = Integer.parseInt(args[0]);
                int b = Integer.parseInt(args[1]);

                int c = a + b;
                System.out.println(c);
        }
}
```

Compilation:

>javac Addition.java

|-> Addition.class

Execution: we must pass two integer numbers from command line as shown below

```
Select C:\Windows\system32\cmd.exe

WC:javac Addition.java

WC:java Addition 5 6
Result: 11

WC:java Addition 7 8
Result: 15

WC:java Addition 7.45 4.33
Exception in thread "main" java.lang.NumberFormatException: For input string: "7
.45"
        at java.lang.NumberFormatExceptio
        at java.lang.Integer.parseInt(Unknow
        at java.lang.Integer.parseInt(Unknown
        at Addition.main(Addition.java:4)

WC:java Addition a b
Exception in thread "main" java.lang.NumberFormatException: For input string: "a
"
        at java.lang.NumberFormatException.forInputString(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at Addition.main(Addition.java:4)

WC:_
```

If we pass other than int value parseInt() method throws NumberFormatException

Q) Is above exception message understandable by end-user?

A) No, this exception message is java developer known message.

Q) Then how can we print user understandable message for the above exception?

A) Using exception handling code we can print user understandable message.

Check next chapter for more details

Chapter 6

Exception Handling

- In this chapter, You will learn
  - Definition and need of exception
  - Exception propagation
  - Need of Exception handling
  - Handling exception using *try/catch*
  - Multiple catch blocks
  - Throwable class *special methods* to print exception message
  - Need of inner try
  - Need of *finally*
  - *Throwable* class hierarchy
  - Checked and unchecked exceptions
  - Creating custom exception
  - Throwing exception manually using *throw*
  - Reporting the exception using *throws*
  - 7 compile time errors
- By the end of this chapter- you will understand usage of try/catch/finally blocks to handle exceptions throw/throws to throw and report exceptions with their complete rules list.

Interview Questions

By the end of this chapter you answer all below interview questions

- Definition exception
- Difference between normal execution and execution with exception
- Exception Propagation.
- Command-Line arguments program (reading data from keyboard)
- Need of Exception Handling
- *Throwable* class hierarchy
- Difference between *java.lang.Error* and *java.lang.Exception*
- Exception Handling procedure
- Exception Handling using ***try/catch***
- *try/catch* execution control flow
- Rules on using **try/catch** blocks
- Multiple catch blocks and rules
- Special methods to print exception message
- Inner try blocks and its execution control flow
- Use *finally* block
- finally block with return statement
- *Throwable* class hierarchy
- Types of exceptions
    a. Checked exceptions
        i. Partial checked exceptions
        ii. Pure checked exceptions
    b. Unchecked exceptions
- Creating custom exception.
- Raising and throwing exception manually using "throw" clause and its rule
- Reporting exception using "throws" clause and its rule
- Difference between *throw* and *throws*
- As per LC-RP when should we use *try/catch/finally* and *throw/throws*
- Method overriding rule with throws clause
- Seven compile time errors.

Exception Handling

Definition of Exception

In general, exceptions are run-time errors caused due to logical mistakes occurred during program execution because of *wrong input*.

For example:

Creating an array with negative number, here -ve number is the wrong input, so JVM terminates program execution by throwing an exception of type *java.lang.NegativeArraySizeException*

Reading array elements by passing index value out of range, here index value is wrong input so JVM terminates program execution by throwing an exception of type *java.lang.ArrayIndexOutOfBoundsException*

etc...

Here *NegativeArraySizeException, ArrayIndexOutOfBoundsException* are exception classes those are representing a logic mistake.

Technical definition of an exception

In Java, exception is an object. In other words it is an instance of one of the subclass of *java.lang.Throwable* class.

SUN defined different exception classes in *java.lang* package for representing different logical mistakes. All these classes are subclasses of *java.lang.Throwable* class.

For example

1. For handling wrong operation on arrays below exception classes are given
   - NegativeArraySizeException
   - ArrayIndexOutOfBoundsException
   - ArrayStoreException
2. If we divide an int number with Zero
   - ArithmeticException
3. If we try to convert *alpha numeric string* data to a number
   - NumberFormatException

etc...

Q) What happened if an exception is raised in the program?

Program execution is terminated abnormally. It means statements placed after exception causing statement are not executed but the statements placed before that exception causing statement are executed by JVM.

Q) What JVM does when a logical mistake occurred in the program?

It creates exception class object that is associated with that logical mistake, and terminates the current method execution by throwing this exception object by using "throw" keyword.
So we can say an exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions execution.

Below program shows program execution *without exception*

```
class NormalExecution {
        public static void main(String[] args){
                int a = 20;
                int b = 10;

                System.out.println("a value: "+a);
                System.out.println("b value: "+b);

                int c = a / b;

                System.out.println("c value: "+c);

        }
```

```
O/P:
====
a value: 20
b value: 10
c value: 2
```

Below program shows program execution *with exception*. This program execution is terminated with exception because we cannot divide integer number by zero.

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

                int a = 20;
                int b = 0;

                System.out.println("a value: "+a);
                System.out.println("b value: "+b);

                int c = a / b;

                System.out.println("c value: "+c);
        }
}
```

```
O/P:
a value: 20
b value: 0
Exception in thread "main"
java.lang.ArithmeticException: / by zero
```

JVM terminates this program by throwing ArithmeticException because the logical mistake we committed is **dividing integer number by integer ZERO**. As you know it is not possible divide a integer number by zero. But it is possible to divide a number with double zero (0.0).

From the above program we can define for exception technically as:

- An exception is an event, which occurs during the execution of a program that disrupts the normal execution flow of the program's instructions.
  - Exception an event because, when an exception is raised JVM internally executes some logic to prepare that exception related message.
- It is a signal that indicates some sort of abnormal condition (such as an error) has been occurred in a code sequence at run time.
  - Exception is a signal because by looking into exception message developer will take necessary actions against that exception.
- In Java, exception is an object that is an instance of some sub class of java.lang.Throwable. So to catch that exception we must define a catch block with that exception class name.
  - Exception is an object, because for throwing exception JVM or we should create appropriate exception class object.

Introduction to *Throwable* class

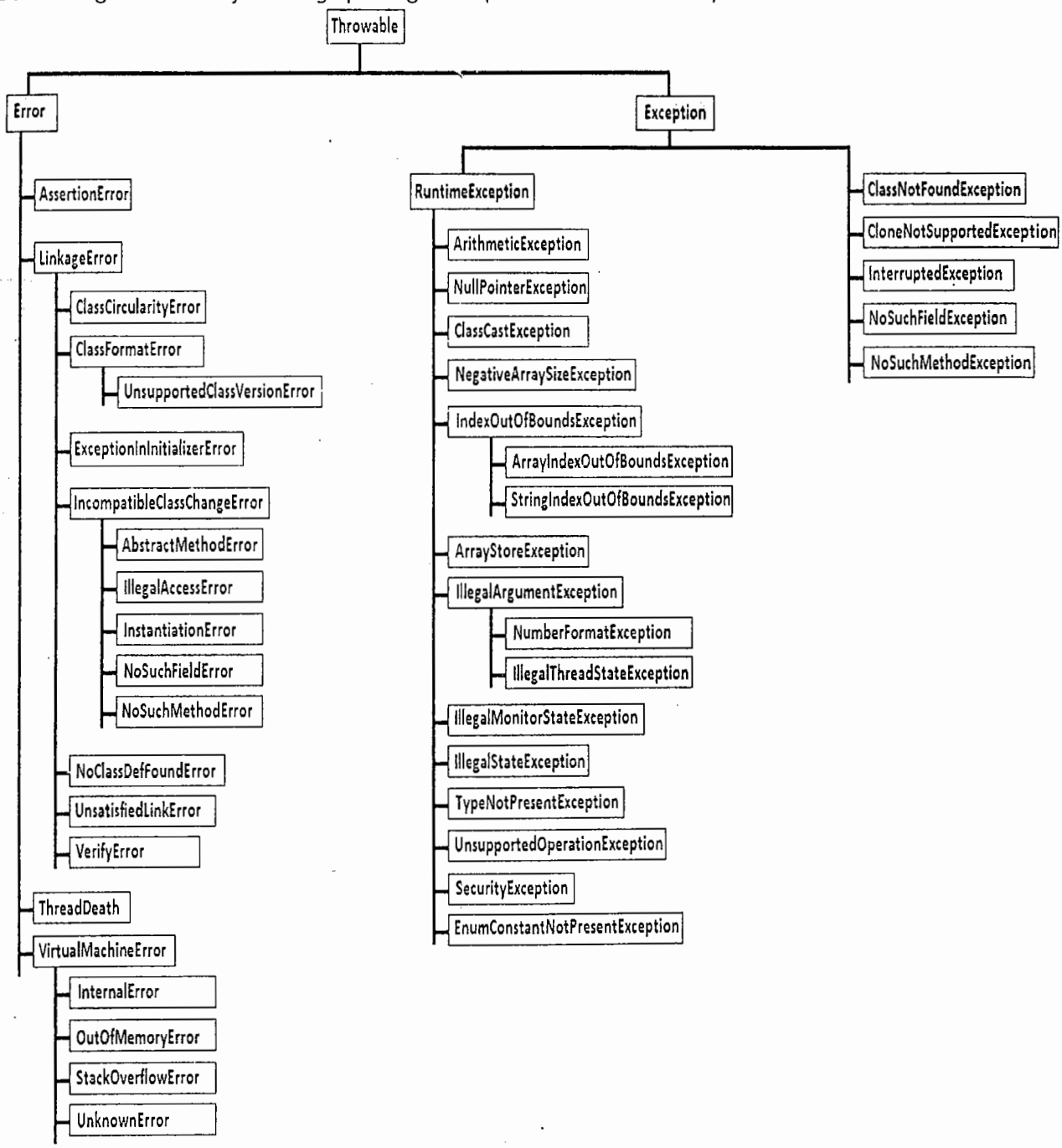
The *Throwable* class is the super class of all exceptions [logical mistakes] in the Java language. Only objects those are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java developer using *throw* statement. Similarly, only this class or one of its subclasses can be the argument type in a *catch* clause.

Throwable class hierarchy

For all types of exceptions java.lang.Throwable is super class. It has two main subclasses

1. Error
2. Exception

Basically Throwable subclasses are divided into two types by using above two subclasses. Below diagram shows "*java.lang*" package exception classes hierarchy+



What are the differences between Error and Exception?

Diff #1:

Error type exceptions are thrown due to the problem occurred in side JVM logic, like

- If there is no memory in JSA to create new stack frame to execute method, then JVM process is killed by throwing Error type exception "*java.lang.StackOverflowError*"

- If there is no memory in HA to create new object, then JVM process is killed by throwing Error type exception "*java.lang.OutOfMemoryError*"

Exception type exceptions are thrown due to the problem occurred in Java program logic execution, like

- If we divide an integer number with ZERO, then JVM terminates program execution by throwing Exception type exception "*java.lang.ArithmeticException*"

Diff #2:

- We cannot catch Error type exception, because Error type exception is not thrown in our application, and once this Error type exception is thrown JVM is terminated.

- We can catch Exception type exceptions, because Exception type exception is thrown in our application, and more over JVM is not directly terminated because of Exception type exceptions. JVM is terminated on if the thrown exception is not caught.

Two types of Exception subciasses

Exception subclasses are again grouped into two categories

1. Subclasses of RumtimeException
2. Direct subclasses of Exception

What is the difference between above types of exception classes?

RumtimeException subclasses represents logical mistakes occurred due to operator execution failure. It means these exceptions are prepared and thrown by JVM at runtime when operator execution is failed.

Exception class direct subclasses represent logical mistakes occurred due to condition failure because of wrong input. These exceptions are prepared and thrown by developer by using "*throw*" keyword.

Two categories of exceptions

All above exceptions are divided into two categories

1. Checked exceptions
2. Unchecked exceptions

When an exception is thrown by developer using throw keyword if that exception handling is checked by compiler then that exception is called checked exception, else that exception is called unchecked exception.

Definition of *checked* and *unchecked* exceptions:
Error, RuntimeException and their subclasses are called unchecked exception because these exceptions handling is not checked by compiler when they are thrown by using throw keyword. It means these exception objects catching or reporting is optional.

Throwable, Exception, and its direct subclasses are called checked exceptions because if they are thrown by using "throw" keyword compiler checks their handling and if they are not caught by using "try/catch" or not reported by using "throws" keyword, compiler throws CE: "***unreported exception must be caught or declared to be thrown***".

Keywords used in this chapter
Java has five keywords for throwing and catching exception
They are:

- ***try/ catch/* finally**: Used for catching the raised exception and for executing statements definitely for a try block
- **throw/ throws**: for throwing an exception manually from a method and for reporting the thrown exception to other programmer

First let us understand try/catch/finally keywords

Check below program, in this application there is a chance of raising three exceptions

```
// Division.java
class Division{
        public static void main(String[] args) {

                int a = Integer.parseInt(args[0]);
                int b = Integer.parseInt(args[1]);
                int c = a / b;

                System.out.println("Result: "+c);
        }
}
```

Test cases
>java Division 40 20
Result: 2

>java Division
Exception in thread "main" *java.lang.ArrayIndexOutOfBoundsException*: 0

>java Division 40 0
Exception in thread "main" *java.lang.ArithmeticException*: /by zero

>java Division 40 a
Exception in thread "main" *java.lang.NumberFormatException*: for the input string "a"

Design issue

Q) Think once, can user understand above exception message?

Definitely no, user cannot understand this exception message, because it is java based exception message. User cannot take further decision alone to resolve this problem. Developer should guide to solve this problem.

Q) What is the solution for this problem?

It is developer responsibility to convert Java exception message into user understandable message. To solve this problem developer should embed exception handling code in java program. Using exception handling code developer can catch exception and can print or pass user understandable message.

*Ultimately, by using exception handling code java program **can talk** to end-user to resolve a particular issue behalf of developer.*

Exception Handling:

The process of catching the exception for converting JVM given exception message to end-user understandable message or for stopping abnormal terminations of the program is called exception handling.

Need of Exception Handling

In projects Exception is handled

- to stop abnormal terminations and
- to provide user understandable messages when exception is raised. So that user can take decision without developer's help.

Basically by implementing Exception Handling we are providing life to a program to talk to user in behalf of developer.

Exception Handling procedure

Exception handling is a 4 step process

Step 1: *Preparing exception object* appropriate to the current logical mistake.
Step 2: *Throwing that exception* to the appropriate exception handler.
Step 3: *Catching that exception*
Step 4: *Taking necessary actions* against that exception.

How can we handle exceptions in Java?

To implement exception handling SUN has given two keywords.

1. try
2. catch

In the exception handling procedure,

The **first two** actions are done by using **try** block, and
The **next two** actions are done by using **catch** block.

try & catch blocks explanation

try

- try keyword establishes a block to write a code that causes exceptions and its related statements. Exception causing statement must be placed in try block to handle and catch exception for stopping abnormal terminations and to display end-user understandable messages.

catch

- catch block is used to catch exceptions those are thrown from its corresponding try block. It has logic to take necessary actions on that caught exception.
- catch block syntax is looks like constructor syntax. It does not take accessibility modifiers, normal modifiers, return type. It takes only single parameter of type *Throwable* (or) its sub classes.
- Throwable is the *super class* of all exception classes
- Inside catch we can write any statement which is legal in java, including raising an exception.

Syntax to use try / catch:

```
void m1(){
        ------------------------------
        ------------------------------      } Normal statements
        ------------------------------

        try{
                ------------------------------
                ------------------------------      } Exception causing or its related statements
                ------------------------------
        }
        catch(someexception1 e){
                ------------------------------
                ------------------------------      } Statements that takes proper actions against someexception1
                ------------------------------
        }
        catch(someexception2 e){
                ------------------------------
                ------------------------------      } Statements that takes proper actions against someexception2
                ------------------------------
        }
}
```

Below program shows handling and catching exceptions to print user understandable messages relevant to the thrown exception. To catch exception we must define a catch block with that exception class parameter as shown below

```
class Division{
        public static void main(String[] args) {
                try{
                        int a = Integer.parseInt(args[0]);
                        int b = Integer.parseInt(args[1]);
                        int c = a / b;

                        System.out.println(c);
                }
                catch(ArrayIndexOutOfBoundsException aiob){
                        System.out.println("Please pass atleast two integer values");
                }
                catch(NumberFormatException nfe){
                        System.out.println("Please pass only integer values");
                }
                catch(ArithmeticException ae){
                        System.out.println("Please DONOT pass second value as ZERO");
                }
        }
}
```

Run below test cases

Test cases

>java Division 40 20
Result: 2

>java Division
Please pass atleast two integer values

>java Division 40 0
Please DONOT pass second value as ZERO

>java Division 40 a
Please pass only integer values

Are above messages user understandable?
Yes right☺ these messages are user understandable. So, user can input correct values to execute this application.

Rules in using try/catch

Rule #1: try must follow either 'ZERO' or 'n' number of catch blocks or '1' finally block else it leads to CE:"***try without catch or finally***".

Rule #2: catch must be placed immediately after try block else it leads to CE: "***catch without try***"

Rule #3: finally must be placed either immediately after 'try' or after 'try/catch' else it leads to CE: "***finally without try***"

Find out CEs in the below list

case #1:
```
try{}
catch(Exception e){}
finally{}
```

case #2:
```
try{}
finally{}
```

case #3:
```
try{}
finally{}
catch(Exception e){}
```

case #4:
```
try{}
```

case #5:
```
catch(Exceptin e){}
```

case #6:
```
finally{}
```

case #7:
```
try{}
int a = 10;
catch(Exception e){}
int b = 20;
finally{}
```

Rule #4 is on catch block parameter type:
The catch block parameter must be of type java.lang.Throwable or its subclasses else it leads to CE: "**incompatible types**"

For example

```
try{}
catch(ArithmeticException ae){}
catch(String s){}
```

Rule #5:
try/catch/finally blocks are not allowed at class level directly, because logic is not allowed at class level.

```
class A{
        try{}
        catch(Exception e ){}


        void m1(){
            try{}
            catch(Exception e ){}
        }
}
```

Can we write more than one try/catch blocks in a method?
Yes, it is possible. Check below program.

```
class Example{
        public static void main(String[] args){
                try{
                        System.out.println("In try1");
                }
                catch(Exception e){
                        System.out.println("In catch1");
                }
                System.out.println("after try/catch1");
                try{
                        System.out.println("In try2");
                }
                catch(Exception e){
                        System.out.println("In catch2");
                }
                System.out.println("after try/catch2");
        }
}
```

Multiple Catch blocks
After try block we can write multiple catch blocks to catch every exception thrown from its corresponding try block.

Can we catch all exceptions using single catch block?

- Yes we can catch all exceptions with single catch block with parameter "*java.lang.Exception*"
- We should use this catch block only for stopping abnormal terminations irrespective of the exception thrown from its corresponding try statement.
- It is always recommended to write catch block with Exception parameter even though we are writing multiple catch blocks. It acts as backup catch block.

When should we write multiple catch blocks for a single try block?

- to print message specific to an exception or
- to execute some logic specific to an exception.

In the above Division program we have placed catch blocks with different exception classes to print messages relevant to the caught exception.

In the Division program, if we would have placed only single catch block with Exception parameter, what message can we print? We cannot place any specific message. We must only place statements common for all exceptions. So we must always define *Exception* parameter catch only as a backup catch of all catch blocks, or to stop abnormal terminations irrespective of the exception type raised from the try block. In this case we must use below methods.

Rules in writing multiple catch blocks

Rule #1: Catch blocks should not be duplicated
Rule #2: Super class parameter catch block should not be placed before child class parameter catch block

Violation of any of the above rules leads to CE: "***exception has already been caught***"

Find out CE from below statements

```
try{}
catch(ArithmeticException ae){}
catch(NullPointerException ae){}
catch(ArithmeticException ae){}
```

```
try{}
catch(Exception e){}
catch(ArithmeticException ae){}
catch(NullPointerException ae){}
```

We write multiple catch blocks not only for printing exception specific message also for executing some logic specific to the exception is raised in the corresponding try statement. Check below program:

Q) Write a program

- **To add the given two numbers if no exception is raised**
- **To add 4, 5 if *ArrayIndexOutOfBoundsException* is raised**
- **To add 6, 7 if *NumberFormatException* is raised**
- **To add 8, 9 if *ArithmeticException* is raised**

```
class Addition{

    public static void main(String[] args){

        try{
            int a = Integer.parseInt(args[0]);
            int b = Integer.parseInt(args[1]);
            int c = a/b;

            System.out.println("Result: "+(a + b));
        }
        catch(ArrayIndexOutOfBoundsException aiobe){
            System.out.println("Result: "+(4 + 5));
        }
        catch(NumberFormatException nfe ){
            System.out.println("Result: "+(6 + 7));
        }
        catch(ArithmeticException ae){
            System.out.println("Result: "+(8 + 9));
        }
    }
}
```

Testing

```
>java Addition
Result: 9

>java Addition a b
Result: 13

>java Addition 3 0
Result: 17

>java Addition 3 2
Result: 5
```

Throwable class methods to print exception messages from catch blocks

Throwable class has below three methods to print exception object stack trace. These methods are useful when we write catch() block with super class as parameter

- **JVM printed Exception message format**

```
Exception in thread<name> exception name: <reason of the exception>
                    <palce of the exception>
```

- **printStackTrace() printed Exception message format**

```
exception name: <reason of the exception>
      <palce of the exception>
```

- **toString() printed Exception message format**

```
exception name: <reason of the exception>
```

- **getMessage() printed Exception message format**

```
<reason of the exception>
```

Below application shows calling above three methods

```
class Test{
        public static void main(String[] args) {
                try{
                        System.out.println(10/0);
                }
                catch(ArithmeticException ae){
                        System.out.println("getMessage() method output");
                        System.out.println(ae.getMessage());
                        System.out.println("==================\n");

                        System.out.println("toString() method output");
                        System.out.println(ae.toString());
                        System.out.println("==================\n");

                        System.out.println("printStackTrace() method output");
                        ae.printStackTrace();
                        System.out.println("==================\n");

                        System.out.println("JVM default output");
                        throw ae;
                        //By using above statement we are just re-throwing the caught
                        exception, this exception is caught by JVM default handler and prints
                        full exception message along with thread name.
                }
        }
}
```

try/ Catch blocks execution control flow

1. JVM always executes try block statements.
2. Catch block statements are executed only if its parameter type exception is raised in its corresponding try statement.

→ When an exception is raised in the try block JVM creates that exception type class object and check for match in among all catch blocks. If match is found, JVM executes that catch block and also statements written after all catch blocks, finally it terminates program execution normally.

→ If catch block is not matched then JVM handles that exception itself, and terminates program execution abnormally, means statements written after try/catch blocks are not executed. Finally, it prints that java exception message on console.

Note: If we do not write try/catch block to catch an exception that exception is sent to JVM default handler and program execution is abnormal termination. If exception is caught by a catch block it is considered as normal termination, also we can pass user understandable message if required, as shown above.

Below diagram shows program execution control flow with try/catch blocks

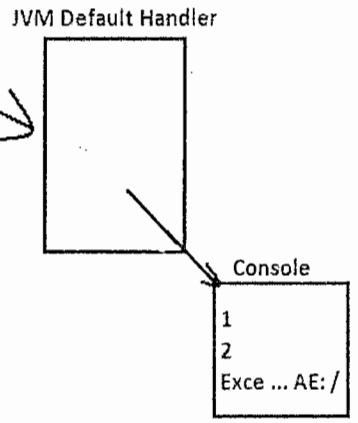
with exception handling & **without handling exception**

```
void m1()
{
  Sopln("1");
  try
  {
     Sopln("2");
     int x = 10/0;
     Sopln("3");
  }
  catch(AE ae)
  {
    Sopln("4");
  }
  Sopln("5");
}
```

O/P
1
2
4
5

NT

```
void m1()
{
  Sopln("1");
  try
  {
     Sopln("2");
     int x = 10/0;
     Sopln("3");
  }
  catch(AIOBE aiob)
  {
    Sopln("4");
  }
  Sopln("5");
}
```



Given:

```
class Example{
   void m1(){
        System.out.println("1");
        try{
                System.out.println("2");
                System.out.println("3");
                System.out.println("4");
        }
        catch(someexception e){
                System.out.println("5");
                System.out.println("6");
                System.out.println("7");
        }
        System.out.println("8");
   }
}
```

What is the output from below test cases?

Case #1: No exception is raised in the program

Case #2: exception is raised at stmt "1"

Case #3: exception is raised at stmt "3" and catch block is matched

Case #4: exception is raised at stmt "3" and catch block is not matched

Case #5: exception is raised at stmt "3" and catch block is matched, again exception is raised at stmt "6"

Can an exception be raised in a catch block, if so who will catch that exception?

We can also raise an exception in a catch block. To catch that exception we must write try/catch inside catch block. If we do not catch that exception, JVM catches that exception and terminates program abnormally.

We can write try/catch anywhere in the method to catch exception. That may be inside another try | catch | finally. Even we can write multiple try/catch blocks in the same method one after one.

What is the output from the below program? Also find out normal or abnormal termination.

```
void m1()
{
   System.out.println("1");
   try
   {
        System.out.println("2");
   }
   catch(ArithmeticException ae)
   {  System.out.println("4");
      System.out.println(10/0);
      System.out.println("5");
   }
   System.out.println("6");
}
```

Console

```
O/P
1
2        NT
6
```

```
void m1()
{
   System.out.println("1");
   try
   {
        System.out.println("2");
        System.out.println(10/0);
        System.out.println("3");
   }
   catch(ArithmeticException ae)
   {
        System.out.println("4");
        System.out.println(10/0);
        System.out.println("5");
   }
   System.out.println("6");
}
```

JDH

Console

```
1 2 4
Exception ....
                ANT
```

```
void m1()
{
   System.out.println("1");
   try
   {
        System.out.println("2");
        System.out.println(10/0);
        System.out.println("3");
   }
   catch(ArithmeticException ae)
   {
        System.out.println("4");
        try{
          System.out.println(10/0);
        }
        catch(AE ae){
          System.out.println("5");
        }
      System.out.println("6");
   }
   System.out.println("7");
}
```

Console

```
1 2 4 5 6 7
                NT
```

In first case, there is no exception in try block, so catch block is not executed
In second case, ArithmeticException is raised in try block so the control is sent to catch block. In catch block also we have raised exception. Since this exception is not caught, program terminated abnormally.
In third case, we added try/catch also in catch block so program is terminated normally.

The point to be remembered is in a method we must use try/catch where ever we want to catch exception to stop abnormal termination, it can be either

- inside a method directly or
- inside a try block or
- inside a catch or
- inside a finally

Inner try/catch (or) Nested try/catch:
The try/catch placed inside another try (or) catch (or) finally is called inner try catch and the parent try/catch is called outer try/catch.

Outer try/catch

Inner try/catch

```
try
{
    try
    {

    }
    catch(SomeException e)
    {
    }
}
catch(SomeException e)
{
}
```

Note: The exceptions raised in inner try are caught by its associated inner catch

What is the output from the below program?

```
class InnterTryDemo {
        public static void main(String[] args) {
                try{
                        System.out.println("In outer try");
                        try{
                                System.out.println("In inner try");
                                System.out.println(10/0);
                        }
                        catch(ArithmeticExceptiont nas){
                                System.out.println("In inner catch");
                        }
                        System.out.println("After inner try/catch");
                }
                catch(ArithmeticExceptiont nas){
                        System.out.println("In outer catch");
                }
                System.out.println("After outer try/catch");
        }
}
```

Need of Inner try, catch:

We should use inner try/catch to handle some exception inside try or catch or finally.

What is the output from the below program?

Below program execution is terminated if AIOBE, NFE exceptions are occurred. If NASE is raised we do not want to terminate program so we caught exception immediately in try block.

```
class InnterTryDemo {
        public static void main(String[] args) {
                try{
                        int a = Integer.parseInt(args[0]);
                        System.out.println("a: "+a);
                        try{
                                int[] x = new int[a];
                                System.out.println("array size: "+x.length);
                        }
                        catch(NegativeArraySizeExceptiont nas){
                                int[] x = new int[3];
                                System.out.println("array size: "+x.length);
                        }
                        int b = a + 20;
                        System.out.println("b: "+b);
                }
                catch (ArrayIndexOutOfBoundsException aiobe){
                        System.out.println("Pass one int value");
                }
                catch (NumberFormatException nfe)        {
                        System.out.println("Pass only int value");
                }
        }
}
```

Case #1: input **50**

Case #2: input **-50**

Case #3: no input

Case #4: input **a**

Inner try/catch control flow

- The exception raised in inner try is caught by inner catch and the statements placed after inner try/catch are executed.
- If the inner catch parameter is not matched with the exception raised in inner try, that exception is propagated to outer try and it is caught by the outer catch and the statements placed after outer try/catch are executed.
- If outer catch also not matched, that exception is propagated to JVM and program execution is terminated abnormally.

What is the output from the below program?

Case #1: Inner try catch is not matched, but outer try catch is matched

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

                try{

                        int a = Integer.parseInt(args[0]);
                        System.out.println("a: "+a);

                        try{
                                int[] x = new int[a];
                                System.out.println("array size: "+x.length);
                        }
                        catch(NullPointerExceptiont npe){
                                System.out.println("NPE is raised");
                        }

                        int b = a + 20;
                        System.out.println("b: "+b);
                }
                catch (ArrayIndexOutOfBoundsException aiobe){
                        System.out.println("Pass one int value");
                }
                catch (NumberFormatException nfe){
                        System.out.println("Pass only int value");
                }
                catch(NegativeArraySizeExceptiont nas){
                        int[] x = new int[3];
                        System.out.println("In outer try array size: "+x.length);
                }
                System.out.println("After outer try/catch");
        }
}
```

Input: **-5**

Output:

Case #2: Inner catch and also outer catch are not matched

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

                try{

                        int a = Integer.parseInt(args[0]);
                        System.out.println("a: "+a);

                        try{
                                int[] x = new int[a];
                                System.out.println("array size: "+x.length);
                        }
                        catch(NullPointerExceptiont npe){
                                System.out.println("NPE is raised");
                        }

                        int b = a + 20;
                        System.out.println("b: "+b);
                }
                catch (ArrayIndexOutOfBoundsException aiobe){
                        System.out.println("Pass one int value");
                }
                catch (NumberFormatException nfe){
                        System.out.println("Pass only int value");
                }

                System.out.println("After outer try/catch");
        }
}
```

Input: **-5**

Output:

Write a program to read two int numbers from keyboard using IOStreams API.
Condition: If user send alphabet, prompt a message asking for sending only number

Clue: To develop this application we must use ***inner try block*** one for each readLine() method call. Check below program.

```
//InnerTryProject.java
import java.io.*;
class Division{
	public static void div(){
		try{
			BufferedReader br = new BufferedReader(
								new InputStreamReader(System.in));

			int a = -1 ;
			int b = -1;

			//infinite loop to prompt message till user enter number
			while (true){
				try{
					System.out.print("Enter first number: ");
					a	= Integer.parseInt(br.readLine());

					break;
				}
				catch(NumberFormatException nfe){
					System.out.println("Wrong input: Enter only number");
				}
			}
			//infinite loop to prompt message till user enter number
			while (true){
				try{
					System.out.print("Enter second number: ");
					b	= Integer.parseInt( br.readLine() );

					try{
						int c = a / b;
						System.out.println("Result: "+c);
					}
					catch(ArithmeticException ae){
						System.out.println("Wrong input: Do not pass Zero");
						continue;
					}

					break;
				}
				catch(NumberFormatException nfe){
					System.out.println("Wrong input: Enter only number");
				}
			}
		}
		catch(IOException ioe){
			ioe.printStackTrace();
		}
	}
}
```

```
public class InnerTryProject {
        public static void main(String[] args) {
                Division.div();
        }
}
```

D:\NareshTechnologies\HariKrishna\CoreJava&OCJP\EH>javac InnerTryProject.java
D:\NareshTechnologies\HariKrishna\CoreJava&OCJP\EH>java InnerTryProject

```
Select C:\Program Files\EditPlus 2\launcher.exe
Enter first number: a
Wrong input: Enter only number
Enter first number: 10
Enter second number: b
Wrong input: Enter only number
Enter second number: 0
Wrong input: Do not pass Zero
Enter second number: 5
Result: 2

Press any key to continue...
```

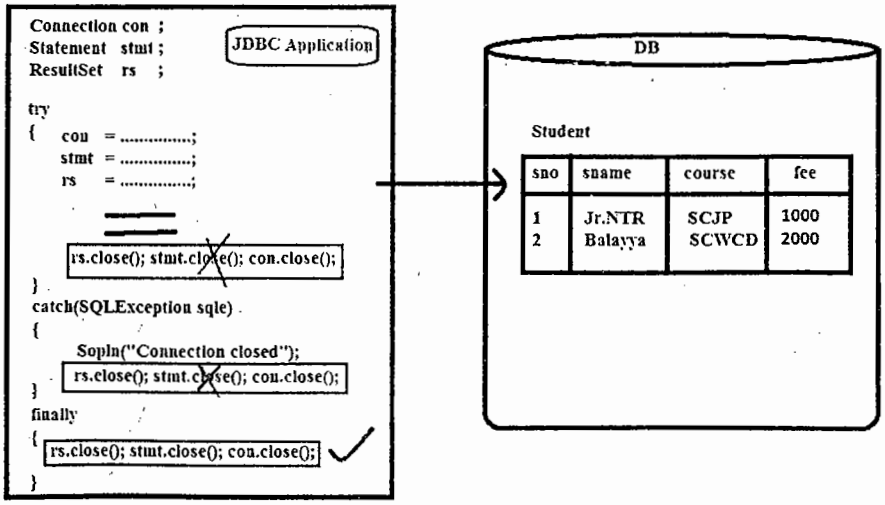
Finally block

Finally establishes a block that definitely executes statements placed in it. Statements which are placed in finally block are always executed irrespective of the way the control is coming out from the try block either by *completing normally* or by *return statement* or by *throwing exception by catching or not catching*.

Need of finally in real time projects:

As per coding standards in finally block we should write resource releasing logic (or) clean up code. Resource releasing logic means unreferencing objects those are created in try block.

For example in real time projects we create JDBC objects in try block and at the end of the try we must close those objects. Since the statements written in try and catch are not guaranteed to be executed we must place them in finally block. Check below diagram.



Basically finally block provides a common place to write resource releasing logic. Hence we can have centralized code change on that resource releasing logic. As shown in the above program- it is always best practice to place the definitely executing statements in finally rather than placing them in both try block and catch block.

For Example, if we want to close JDBC objects - Connection, Statement, ResultSet, etc... we must call close() method in both try and as well as in catch block to guarantee its execution. Instead of placing the same close() method call statement in multiple places if we write it in finally block, it is always executed irrespective of exception raised or not raised.

Syntax to define finally block

We can use finally in two ways

1. try/catch/finally
2. try/finally

```
try{}
catch(exception1){}
finally{}
```

Scenario to use *try / catch / finally*

We use this syntax to *catch exception* and also to *execute some statements definitely.*

```
try{}
finally{}
```

Scenario to use *try/finally*

We use this syntax if we do not want to catch exception, but if we want to execute some statements definitely. In this case program execution is terminated abnormally.

What is the output from the below programs?

Case #1: try/catch/finally without exception

```
class Example{

        public static void main(String[] args) {

                try{
                        System.out.println("In try");
                }
                catch(ArithmeticException ae){
                        System.out.println("In catch");

                }
                finally{
                        System.out.println("In finally");

                }
                System.out.println("After try/catch/finally");

        }

}
```

Output

Case #2: try/catch/finally with exception and catch block is matched

```
class Example{
        public static void main(String[] args) {
                try{
                        System.out.println("In try");
                        System.out.println(10/0);
                }
                catch(ArithmeticException ae){
                        System.out.println("In catch");
                }
                finally{
                        System.out.println("In finally");
                }
                System.out.println("After try/catch/finally");
        }
}
```

Output

Case #3: try/catch/finally with exception and catch block is not matched

```
class Example{
        public static void main(String[] args) {
                try{
                        System.out.println("In try");
                        System.out.println(10/0);
                }
                catch(NullPointerException npe){
                        System.out.println("In catch");
                }
                finally{
                        System.out.println("In finally");
                }
                System.out.println("After try/catch/finally");
        }
}
```

Output

Case #4: try /finally without exception

```
class Example{
        public static void main(String[] args) {
                try{
                        System.out.println("In try");
                }
                finally{
                        System.out.println("In finally");
                }
                System.out.println("After try/finally");
        }
}
```

Output

Case #5: try/finally with exception

```
class Example{
        public static void main(String[] args) {
                try{
                        System.out.println("In try");
                        System.out.println(10/0);
                }
                finally{
                        System.out.println("In finally");
                }
                System.out.println("After try/finally");
        }
}
```

Output

Finally block with return statement:

If we keep return statement in the finally block always the value returned from the finally block is transferred to calling method.

What is the output from the below program?

```
class Example{
   public static void main(String[] args) {
        System.out.println( m1() );
   }
   static int m1(){
        try{
                System.out.println("In try");
                return 10;
        }
        catch(ArithmeticException e){
                System.out.println("In catch");
                return 20;
        }
        finally{
                System.out.println("In finally");
                return 30;
        }
   }
}
```

Output

If finally block has return statement the exception raised in try block is never propagated to calling method, because the returned value overrides exception object in JDH.

What is the output from the below program?

```
class Example{
   public static void main(String[] args) {
         System.out.println( m1() );
   }
   static int m1(){
         try{
                  System.out.println("In try");
                  System.out.println(10/0);
         }
         catch(NullPointerException e){
                  System.out.println("In catch");
         }
         finally{
                  System.out.println("In finally");
                  return 30;
         }
   }
}
```

Output

In case of void method we can surpress exception by using return;
What is the output from the below program?

```
class Example{
  public static void main(String[] args) {
         m1();
         System.out.println("After m1 calling");
  }
  static  void m1(){
         try{
                  System.out.println("In try");
                  System.out.println(10/0);
         }
         catch(NullPointerException e){
                  System.out.println("In catch");
         }
         finally{
                  System.out.println("In finally");
                  return;
         }
   }
}
```

Output

What is the output from below program?
Is program execution terminated normally or abnormally?

```
class Example{
   public static void main(String[] args) {
         System.out.println( m1() );
   }
   static int m1(){
         try{
                  System.out.println("In try");
                  System.out.println(10/0);
         }
         catch(NullPointerException e){
                  System.out.println("In catch")
         }
         finally{
                  System.out.println("In finally");
         }
         return 30;
   }
}
```

Output

Unreachable Statement:
If we place return statement in finally block we cannot place statements after finally block it leads to CE: ***unreachable statements***.

```
class Example{
   public static void main(String[] args) {
         System.out.println( m1() );
   }
   static int m1(){
         try{
                  System.out.println("In try");
                  return 10;
         }
         catch(ArithmeticException e){
                  System.out.println("In catch");
                  return 20;
         }
         finally{
                  System.out.println("In finally");
                  return 30;
         }
         System.out.println("after try/catch/finally");  ╳CE: unreachable statement
   }
}
```

Inner finally block

We can also write finally for inner try block that finally block is called inner finally.

What is the output from below cases?

Case #1: No Exception is raised in inner or in outer try blocks

```
class Example{
   public static void main(String[] args) {
        m1();
   }
   static void m1(){
        try{
                System.out.println("In outer try");
                try{
                        System.out.println("In Inner try");
                }
                catch(NullPointerException e){
                        System.out.println("In Inner catch")
                }
                finally{
                        System.out.println("In Inner finally");
                }
                System.out.println("After Inner try/catch/finally");
        }
        catch(NullPointerException e){
                System.out.println("In outer catch")
        }
        finally{
                System.out.println("In outer finally");
        }
        System.out.println("After outer try/catch/finally");
   }
}
```

Output

Case #2: return statement is placed in inner finally

```
class Example{
   public static void main(String[] args) {
         System.out.println( m1() );
   }
   static int m1(){
         try{
                  System.out.println("In outer try");
                  try{
                           System.out.println("In Inner try");
                  }
                  catch(NullPointerException e){
                           System.out.println("In Inner catch")
                  }
                  finally{
                           System.out.println("In Inner finally");
                           return 10;
                  }
                  //System.out.println("After Inner try/catch/finally"); CE: u r s
         }
         catch(NullPointerException e){
                  System.out.println("In outer catch")
         }
         finally{
                  System.out.println("In outer finally");
         }
         System.out.println("After outer try/catch/finally");
         return 30;
   }
}
```

Output

Q) In the above program if we place return 30 in the outer finally what is the output change?

Case #3: exception is raised in inner try, and either inner catch or outer catch is not matched

```
class Example{
   public static void main(String[] args) {
        m1() ;
   }
   static void m1(){
        try{
                System.out.println("In outer try");
                try{
                        System.out.println("In Inner try");
                        System.out.println(10/0);
                }
                catch(NullPointerException e){
                        System.out.println("In Inner catch")
                }
                finally{
                        System.out.println("In Inner finally");
                }
                System.out.println("After Inner try/catch/finally");
        }
        catch(NullPointerException e){
                System.out.println("In outer catch")
        }
        finally{
                System.out.println("In outer finally");
        }
        System.out.println("After outer try/catch/finally");
   }
}
```

Output:

Case #4: exception is raised in inner try, and either inner catch or outer catch is not matched, and also return statement is placed inside inner finally

```
class Example{
   public static void main(String[] args) {
         System.out.println( m1() );
   }
    static int m1(){
         try{
                  System.out.println("In outer try");
                  try{
                           System.out.println("In Inner try");
                           System.out.println(10/0);
                  }
                  catch(NullPointerException e){
                           System.out.println("In Inner catch")
                  }
                  finally{
                           System.out.println("In Inner finally");
                           return 20;
                  }
                  //System.out.println("After Inner try/catch/finally"); CE: u r s
         }
         catch(NullPointerException e){
                  System.out.println("In outer catch")
         }
         finally{
                  System.out.println("In outer finally");
         }
         System.out.println("After outer try/catch/finally");
         return 30;
   }
}
```

Output:

Exception Propagation:

The process of sending exception from called method to calling method (ex: from m1() to main()) is called exception propagation. If an exception is propagated and if that exception is not caught in that calling method, not only called method execution but also calling method execution is terminated abnormally.

Exception propagation with try/catch:

Case #1: If exception is caught in m1() method then exception is not propagated to calling method. Check below programs

```
class Example{
        public static void main(String[] args) {
                System.out.println("In main method start");
                m1();
                System.out.println("In main method end");
        }
        static void m1(){
                try{
                        System.out.println("In m1 method start");
                        System.out.println(10/0);
                        System.out.println("In m1 method end");
                }
                catch(ArithmeticException ae){
                        System.out.println("In m1 catch");
                }
        }
}
```

Output:

Case 2: If the exception is not caught in m1() method, it is propagated to calling method.

```
class Example{
        public static void main(String[] args) {
                System.out.println("In main method start");
                m1();
                System.out.println("In main method end");
        }
        static void m1(){
                System.out.println("In m1 method start");
                System.out.println(10/0);
                System.out.println("In m1 method end");
        }
}
```

Output

Case #3: To have smooth execution in calling method the propagated exception should be caught with try/catch

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

                System.out.println("In main method start");

                try{
                        m1();
                }
                catch(ArithmeticException ae){
                        System.out.println("In main catch");
                }

                System.out.println("In main method end");
        }
        static void m1(){
                System.out.println("In m1 method start");
                System.out.println(10/0);
                System.out.println("In m1 method end");
        }
}
```

Output

Compiler thinking in compiling try/catch blocks

Compiler thinks, the statements written in try and catch blocks are not definitely executed as try has exception causing statements and catch block is executed only if exception is raised.

Variable initialization in try/catch blocks

If we initialize local variable in try block, and if we access it in one of the catch blocks or after try/catch it leads to CE: "***variable might not have been initialized***" because for compiler the variable initialization statement is not definitely executed.

Check below programs

Case #1: variable only initialized in try block

```
void m1(){
        int a;

        try{
          a = 10;
          System.out.println(a);
        }
        catch(ArithmeticException e){
          System.out.println(a);      CE: variable a might not have been initialized
        }
        System.out.println(a);    CE: variable a might not have been initialized
}
```

Case #2: variable initialized in both try and catch blocks

```
void m1(){
        int a;

        try{
          a = 10;
          System.out.println(a);
        }
        catch(ArithmeticException e){
          a = 20;
          System.out.println(a);
        }
        System.out.println(a);
}
```

Case #3: variable is initialized in try and only in one catch block

```
void m1(){
        int a;

        try{
          a = 10;
          System.out.println(a);
        }
        catch(ArithmeticException e){
          a = 20;
        }
        catch(NullPointerException e){
          System.out.println(a);      CE: variable a might not have been initialized
        }
        System.out.println(a);    CE: variable a might not have been initialized
}
```

return statement in try/catch blocks

If we return a value in try block, and if we do not return value at end of the method it leads to CE: "***missing return statement***" because for compiler the value returned from the try is not definitely executed. **Check below programs**

Case #1: return statement is placed only in try block

```
int m1(){

        try{
          System.out.println("In try");
          return 10;
        }
        catch(ArithmeticException e){
          System.out.println("In catch");
        }
        System.out.println("after try/catch");

}   ╳ CE: missing return statement
```

Case #2: return statement is placed in both in try and catch block

```
int m1(){

        try{
          System.out.println("In try");
          return 10;
        }
        catch(ArithmeticException e){
          System.out.println("In catch");
          return 20;
        }

        System.out.println("after try/catch");   ╳ CE: unreachable statement
}
```

In the above case the statements placed after try/catch blocks are become not reachable as control will send out of the method either from try block or from catch block.

Case #3: project scenario to use return statement with try block

It is always recommend returning a value from end of the method by using a local variable to solve the above two compile time errors.

At the beginning of the method create a local variable with some default value and initialize it with value in try or in catch based on the required conditions. Then at end of the method return this variable.

Check below program

Q) Write a program to return a value

- **4 if no exception is raised.**
- **5 if *ArrayIndexOutOfBoundsException* is raised**
- **6 if *NumberFormatException* is raised**
- **7 if *ArithmeticException* is raised**

Check below program:

```
class Division{
        static int div(String[] args){
                int res = -1;

                try{
                  int a = Integer.parseInt(args[0]);
                  int b = Integer.parseInt(args[1]);
                  int c = a / b;
                  res = 4;
                }
                catch(ArrayIndexOutOfBoundsException e){
                  res = 5;
                }
                catch(NumberFormatException e){
                  res = 6;
                }
                catch(ArithmeticException e){
                  res = 7;
                }
                System.out.println("after try/catch");

                return res;
        }
}
```

```
class Test{
    public static void main(String[] args){
        int res = Division.div(args);
        System.out.println("Result: "+res);
    }
}
```

What is the output from the below testcases

>java Test 20 10

>java Test

>java Test 20

>java Test a 10

>java Test 20 0

Now let us understand throw and throws keyword.

Use of "throw" and "throws" keywords

- throw keyword is used to throw exception manually. In most of the cases we use it from throwing checked exceptions explicitly.

- throws keyword is used to report that raised exception to the caller. It is mandatory for checked exceptions for reporting, if they are not handled.

syntax to use "throw" and "throws" keyword

throw:

- throw keyword must follow Throwable type of object.
- it must be used only in method logic

Rule: since it is a transfer statement, we cannot place statements after throw statement. It leads to CE: "***unreachable statement***".

For example:

```
void m1(){
        throw new ArithmeticException();
}
```

AE is a runtime exception, so compiler does not check its exception handling.

```
void m1(){
        throw new InterruptedException();
}
```

IE is a subclass of Exception, so we must catch or report this exception using throws keyword. Since we have not done either of both, compiler throws CE: "**unreported exception IE must be caught or declared to thrown**"

Below code shows correct syntax of throwing checked exception

```
void m1()throws InterruptedException {
        throw new InterruptedException();
}
```

throws:

- throws keyword must follow Throwable type of class name.
- it must be used in method prototype after method parenthesis.

For Example:

```
void m1() throws ArithmeticException{
        throw new ArithmeticException();
}
void m1() throws InterruptedException{
        throw new InterruptedException();
}
```

Rule: we are not allowed to write catch block with checked exception without throwing it from the try block, it leads to CE: "***exception never thrown from the corresponding try statement***"

In below program catch block statement leads to above CE

```
void m1() {
        try{
        }
        catch(InterruptedException e){
                System.out.println(e);
        }
}
```

Below program compiles fine

```
void m1() {
        try{
                throw new InterruptedException();
        }
        catch(InterruptedException e){
                System.out.println(e);
        }
}
```

We can catch and also we can report using throws

```
void m1() throws InterruptedException{
        try{
                throw new InterruptedException();
        }
        catch(InterruptedException e){
                System.out.println(e);
        }
}
```

We can report checked exception without throwing it from method

```
void m1() throws InterruptedException{

}
```

We can write catch block for unchecked exception without throwing that exception

```
void m1() {
        try{
        }
        catch(ArithmeticException e){
                System.out.println(e);
        }
}
```

Special case: we are allowed to place catch block for Exception and Throwable even though they are not thrown from try block because they are super classes of both checked and unchecked exceptions.

```
void m1() {
        try{
        }
        catch(Exception e){
                System.out.println(e);
        }
}
```

But if they are thrown using throw keyword they must be handled or reported.
Below program leads to CE: "***unreported exception java.lang.Exception must be caught or declared to be thrown***"

```
void m1() {
        throw new Exception();
}
```

Below program compiles fine

```
void m1()throws Exception {
        throw new Exception();
}
```

Method overriding with throws keyword

Case #1: If super class method is not reporting / not throwing checked exception, subclass overriding method is not allowed to throw checked exception, it leads to CE:

For example

```
class A{
        void m1(){}
}
```

Below program compiled fine

```
class B extends A{
        void m1(){}
}
```

Below program also compiled fine because it is unchecked exception

```
class C extends A{
        void m1() throws RuntimeException{}
}
```

Below program leads to CE, because it is checked exception

```
class D extends A{
        void m1() throws Exception{}
}
```

Case #2: If super class method has throws clause, subclass overriding method may or may not have throws clause. If we place throws clause in overriding method it must be either same exception class or subclass. It should not be super class or sibling, also we cannot add more exceptions to throws clause

For Example,

```
class A{
        void m1() throws InterruptedException{}
}
```

Below program compiled fine, because overriding method has same type of exception

```
class B extends A{
        void m1() throws InterruptedException{}
}
```

Below program compiled fine, because overriding method can remove throws keyword

```
class C extends A{
        void m1(){}
}
```

Below program leads to CE, because overriding method has super class type

```
class D extends A{
        void m1() throws Exception{}
}
```

Below program leads to CE, because super class method is throwing exception that must be report or must be catch in calling method also (overriding method)

```
class E extends A{
        void m1() {
                super.m1();
        }
}
```

Below class define a method by throwing checked exception

```
class F{
        static void m2() throws ClassNotFoundException{
        }
}
```

Calling the above method from the overriding method, below program leads to CE, because m2 method throwing checked exception ClassNotFoundException

```
class G extends A
        void m1(){
                F.m2();
        }
}
```

Solution: we must report or must catch that ClassNotFoundException

Below program leads to CE, because m1() method in A class does not throw ClassNotFoundException, and CNFE is not a subclass of InterruptedException

```
class G extends A{
        void m1()throws ClassNotFoundException{
                F.m2();
        }
}
```

Solution #1: must catch CNFE in m1() method by using try/catch

```
class G extends A
        void m1(){
                try{
                        F.m2();
                }
                catch(ClassNotFoundException e){}
        }
}
```

Solution #2: if you do not want to catch this exception, wrap this exception object in the exception class that is throwing from that method. If that exception class exception does not have "Exception" parameter constructor, use RuntimeException class.

In this example case, IE does not have "Exception" parameter constructor to wrap CNFE. So, we must use RuntimeException, to escape from "CE: unreported exception" as shown below

```
class G extends A
        void m1(){
                try{
                        F.m2();
                }
                catch(ClassNotFoundException e){
                        throw new RuntimeException(e);
                }
        }
}
```

Custom exception development

The new exception class develop by a developer is called custom exception or user defined exception. These classes must be subclass of either Throwable or any one of its sub class. Most of the cases in projects custom exception classes are derived from Exception class.

All exception classes defined by SUN are based on the Java language and API requirements. So according to the business requirements developer must write their own exception class.

Procedure to develop custom exception class

It is a two steps process

1. Define a packaged public class deriving from java.lang.Exception.
2. Define public no-arg and String parameter constructors with super() call.
    - No-arg constructor for creating exception object without message
    - Parameterized constructor for creating exception object with message.

Q) Why did you extend custom exception class from java.lang.Exception class?
A) We created custom exception classes for throwing these exceptions when a condition is failed and that exception's handling want to be validated by compiler so they must be extended from java.lang.Exception.

We must extend them from java.lang.RuntimeException class; if we do not want validate these exceptions handling by compiler

Since we do not write exception classes for handling errors in JVM internal logic we do not derive them from java.lang.Error.

Also we do not derive them from java.lang.Throwable as it creates new category of exception and is not recommended to create new category of exceptions because this custom exception is not caught by catch(Exception e). In projects developers write catch(Exception e) to catch all types of exceptions and in this case our exception will not be caught.

Define custom exceptions *InvalidAmountException, InsufficientFundsException* to handle wrong operations done by customer in *deposit* and *withdraw* operations.
Test cases are:

- Throw *InvalidAmountException* if user enter ZERO or –ve number in deposit or withdraw
- Throw *InsufficientFundsException* if user enter amt > balance

//InvalidAmountException.java

```
package com.nareshit.exceptions;
public class  InvalidAmountException extends Exception{
        public InvalidAmountException() {
                super();
        }
        public InvalidAmountException(String msg) {
                super(msg);
        }
}
```

//InSufficientFundsException.java

```
package com.nareshit.exceptions;
public class  InSufficientFundsException extends Exception{

        public InSufficientFundsException() {
                super();
        }
        public InSufficientFundsException(String msg) {
                super(msg);
        }
}
```

//Bank.java

```
package com.nareshit.blogic;

import com.nareshit.exceptions.*;

public interface Bank  {

        public void deposite(double amt) throws InvalidAmountException;
        public double withdraw(double amt)  throws InSufficientFundsException;
        public void balanceEnquiry();

}
```

//HDFCBank.java

```
package com.nareshit.blogic;

import com.nareshit.exceptions.*;
public class  HDFCBank implements Bank{

        private double balance;

        public void deposite(double amt) throws InvalidAmountException{

                if ( amt <= 0 ){
                        throw new InvalidAmountException(amt + " is invalid amount");
                }

                balance = balance + amt;
        }

        public double withdraw(double amt) throws InSufficientFundsException{

                if (balance < amt){
                        throw new InSufficientFundsException("InSufficient Funds");
                }

                balance = balance - amt;
                return amt;

        }

        public void balanceEnquiry(){
                System.out.println("Current Balance: "+ balance);
        }
}
```

```
//Clerk.java
package com.nareshit.user;

import java.io.*;

import com.nareshit.blogic.*;
import com.nareshit.exceptions.*;

public class  Clerk{
        public static void main(String[] args) {
                try{
                        BufferedReader br =
                                new BufferedReader(new InputStreamReader(System.in));

                        Bank acc1 = new HDFCBank();
                        String option = "";
                        do{

                                System.out.println("1. Deposite");
                                System.out.println("2. Withdraw");
                                System.out.println("3. Balance Enquairy");

                                System.out.print("Enter option: ");
                                option = br.readLine();

                                switch(option){

                                        case "1":
                                        {
                                                System.out.print( "Enter deposite amount: ");
                                                String s = br.readLine();
                                                double amt = Double.parseDouble( s );

                                                acc1.deposite( amt );
                                                acc1.balanceEnquiry();

                                                break;
                                        }
                                        case "2":
                                        {
                                                System.out.print("Enter withdraw amount: ");

                                                String s = br.readLine();
                                                double amt = Double.parseDouble( s );

                                                double withDrawAmt = acc1.withdraw(amt);
```

```
                        System.out.println(
                                "Withdrawn amount: "+ withDrawAmt
                        );
                        acc1.balanceEnquiry();
                        break;
                }
                case "3":
                {
                        acc1.balanceEnquiry();
                        break;
                }

                default:
                        System.out.println("Invalid Option");
            }

            System.out.print("Do you want to continue(Yes/No): ");
            option = br.readLine();

        }while (option.equalsIgnoreCase("Yes"));

    }
    catch(InvalidAmountException ie){
        System.out.println( ie.getMessage() );
    }
    catch(InSufficientFundsException e){
        System.out.println( e.getMessage() );
    }
    catch(NumberFormatException e){
        System.out.println( "Please enter ONLY number");
    }
    catch(IOException e){
        e.printStackTrace();
    }
  }
}
```

7 compile time errors

1. try without catch or finally
2. catch without try
3. finally without try
4. exception has already been caught
5. unreachable statement
6. unreported exception must be caught or declared thrown
7. exception has never thrown from corresponding try statement
   +
8. cannot find symbol
9. variable might not have been initialized
10. missing return statement
11. incompatible types
    a. found: <something>
    b. required: Throwable
12. illegal start of expression
13. method in subclass is not overriding method in superclass

Chapter 7

Packages

- In this chapter, You will learn
  - Definition and Need of packages in project
  - Creating package programmatically & linking it with class
  - Need of javac tool option "-d"
  - Class loader working functionalities
  - Organizing classes as packages
  - Creating sub packages
  - Using other package members
  - Understanding fully qualified name
  - Understanding Import statement
  - Using sub package members
  - Java source file structure
  - Inbuilt packages

- By the end of this chapter- you will understand creating and using package members from same and different package members with proper rules.

Interview Questions

By the end of this chapter you answer all below interview questions

Package keyword

- Definition and need of package.
- Package rule
- Creating packages manually and programmatically
- When should we call a folder is a package?
- Class path settings to access package members from other packages.
- How can we store multiple public and non-public classes in a single package?
- Defining Sub-packages.
- What is the fully qualified name of the class?

Import keyword

- Using existed packages
    a. Fully qualified name
    b. import statements
- What are the difference between
    a. import <packagename>.*;
    b. import <packagename>.<classname>;
- import statement rule
- What is the information import statement provides?
- Is import statement load class into JVM?
- Give a scenario that force you to use both fully qualified name and import statement to access a class/ interface.
- Accessing packaged classes from non-packaged class?
- Accessing non-packaged classes from packaged class?
- importing sub packages
- static imports
- Java program source file structure
- Inbuilt packages.

- Accessibility modifiers for package members to access from other packages
    a. private
    b. <default>
    c. protected
    d. public
- protected accessibility modifier rule

So far we have learnt creating class individually. In this chapter we will learn how can group classes and how to separate one class from other class if both have same name using package.

Definition and Need of package

A folder that is linked with java class is called package. It is used to group related classes, interfaces and enums. Also it is used to separate new classes from existed classes. So by using package we can create multiple classes with same name, also we can create user defined classes with predefined class names.

Package creation

To create a package we have a keyword called "package".

Syntax

package <package name>;

For example: package p1;

Rule on package statement

package statement should be the first statement in a java file.

Default package

package statement is optional. If we define a class without package statement, then that class is said to be available in "default package i.e.; current working directory".

Below program shows creating a class with package

//Example.java

```
package p1;
class Example{
        public static void main(String[] args) {
                System.out.println("In Example main");
        }
}
```

Compilation

Compiler does not create package physically in current working directory just with *javac* command. Packaged classes must be compiled with "-d" option to create package physically.

Syntax: *javac –d <path in which package should be copied> source filename*

For Example

> javac -d . Example.java

With this command compiler creates package "p1" with "Exmple.class" and places it in current working directory. Operator "." represents current working directory.

> javac -d C:\test Example.java

With this command compiler creates package "p1" with "Exmple.java" in C:\test folder.

Rule: test folder must be existed in C drive before this program compilation, else it leads to CE.

"-d" functionality

Its actual functionality is creating package with the name mentioned in java file and moving all .class files in that package, and finally storing that package in the given path.

Packaged class code changed by compiler

After compilation compiler replaces class name and its constructor name with its *packagename.classname*. It is called *fully qualified* name.

Check below diagram

//Example.java DWC

```
package p1;

class Example{
        public static void main(String[] args){
                System.out.println("Hi");
        }
}
```

//Example.class CCC

```
class p1.Example extends java.lang.Object{
        p1.Example(){
                super();
        }
        public static void main(String[] args){
                System.out.println("Hi");
        }
}
```

Execution:

We must use package name in executing a packaged class else it leads to exception

>java p1.Example

Hi

Q) Why we must use package name in execution?

As you observed, in Example.class the class is changed as p1.Example. Since name is p1.Example, it must be executed with the same name means in execution we must use package name.

Q) Can we execute a class from CWD that is placed in another directory?

No, it leads to exception "java.lang.NoClassDefFoundError"

For example

>javac -d C:\test Example.java

>java p1.Example

Exception in thread "main" java.lang.NoClassDefFoundError: p1/Example

If package is placed in another directory, we must update its path in Classpath environment variable, else it leads above exception.

Updating Classpath environment variable

We can update Classpath in 3 ways

1. By using java command option "–cp" or "–classpath"
2. By using "Set Classpath" command
3. By using "Environment Variables window"

In *first approach* we will have Classpath setting only for the *current java command* execution
In *second approach* we will have Classpath settings only for the *current command prompt*
In *third approach* we will have Classpath settings for *all command prompts forever*.

Usage:

```
>javac –d C:\test Example.java
>java –cp C:\test p1.Example
Hi
```

or

```
>java –classpath C:\test p1.Example
Hi

>Set Classpath=C:\test
>java p1.Example
Hi
```

ClassLoader working functionality

- ClassLoader loads classes into JVM based on Classpath environment variable setup.
- To load class bytes into JVM it searches in the folders those are configured in Classpath environment variable. It searches all folders till it finds given class's ".class" file. If it not found in any one of the folder then it throws "*NoClassDefFoundError*" exception.
- If Classpath is not at all created, then it loads classes only from current working directory.
- If Classpath is created it is mandatory to place "." operator to load classes from current working directory.
- If the classpath environment variable has the character ";" not as a separator, it is treated as "." so that ClassLoader loads classes from current working directory.

Find out from which folder class is loaded and executed from the below syntax

- set classpath=C:\test <= class is loaded from C:\test
- set classpath=. <= class is loaded from current working directory
- set classpath=.;C:\test <= class is loaded from current working directory
- set classpath=C:\test;. <= class is loaded from C:\test
- set classpath=NareshIT;C:\test <= class is loaded from C:\test
- set classpath= NareshIT;;C:\test <= class is loaded from current working directory
- set classpath=NareshIT;C:\test; <= class is loaded from C:\test
- set classpath=NareshIT <= JVM throws "java.lang.NoClassDefFoundError"
- set classpath=NareshIT; <= class is loaded from current working directory

Q) When we are using a class from another class, should I compile that class first?
No need to compile. Compiler automatically compiles that class. For example assume we are calling Example class method from Sample class method we can compile Sample class directly without compiling Example class.

Compiler follows below procedure to compile Example class

1. First it searches for that Example class definition in Sample.java, if not found
2. It searches for Example.class in Sample class package, if not found
3. It searches for Example.java in Sample class package , if not found
4. It searches for Example.class in imported packages, if not found
5. It searches for Example.java in the imported packages, if not found
6. Then compiler terminates Sample.java file compilation by throwing
   CE: cannot find symbol

7. If Example.java is found, it searches for Example class definition in Example.java file. If it is found, compiler compile entire java file, it means it also compiles other class definitions and generates those class's .class files. Else terminates Sample.java file compilation with above compilation error.

8. If Example.class is found, it also searches for Example.java. If not found, compiler uses Example.class file directly.

9. If Example.java is also available, it checks modified time of both files, if Example.java file modified date is greater than Example.class modified date, it compiles Example.java again for generating Example.class with its latest changed java code.

Test all above points using below two programs
Case #1: Example class definition in another java file

Example.java

```
class Example
{
        static int a = 50;
        static int b = 60;
}
```

Sample.java

```
class Sample{
        p s v main(String[] args){
                Sopln(Example.a);
                Sopln(Example.b);
        }
}
```

Output

```
>javac Sample.java
>java Sample
50
60
```

Case #2: change "a" and "b" variables to 70 and 80 and save Example.java. Then compile and execute Sample.java file directly. Now Example.class is regenerated with new values.

Example.java

```
class Example
{
        static int a = 70;
        static int b = 80;
}
```

Sample.java

```
class Sample{
        p s v main(String[] args){
                Sopln(Example.a);
                Sopln(Example.b);
        }
}
```

Output

```
>javac Sample.java
>java Sample
70
80
```

Case #3: Define Example class in Sample.java with "a" and "b" values as 15 and 16. Then compile and execute Sample.java file directly. Now Example.class is regenerated with the Example class values defined in Sample.java – local preference.

Example.java

```
class Example
{
        static int a = 70;
        static int b = 80;
}
```

Sample.java

```
class Sample{
        p s v main(String[] args){
                Sopln(Example.a);
                Sopln(Example.b);
        }
}

class Example
{
        static int a = 15;
        static int b = 16;
}
```

Output

```
>javac Sample.java
>java Sample
15
16
```

Case #4: From the below code how many class files are generated after compiling Sample.java.

Example.java

```
class Example
{
        static int a = 70;
        static int b = 80;
}

class A{}
class B{}
```

Sample.java

```
class Sample{
        p s v main(String[] args){
                Sopln(Example.a);
                Sopln(Example.b);
        }
}
```

Output

```
>javac Sample.java
        |->A.java
        |->B.java
        |->Example.java
        |->Sample.java

4 class files are generated
```

Case #5: Change Example class name to Test in Example.java file, delete Example class definition in Sample.java file, also Example.class from current working directory. Then compile and execute Sample.java file, now you will get CE: cannot find symbol.

Example.java

```
class Test
{
        static int a = 70;
        static int b = 80;
}
```

Sample.java

```
class Sample{
        p s v main(String[] args){
                Sopln(Example.a);
                Sopln(Example.b);
        }
}
```

Output

```
>javac Sample.java
CE: cannot find symbol
Symbol: variable Example
location: class Sample
```

Case #6: Delete Example.java file and now compile Sample.java, in this case also you will get same above compile time error.

Creating sub packages:

Syntax:
pacakge parentpackagename.subpackagename;

For example: package p1.p2;

Below program shows creating sub package
//Example.java

```
package p1.p2;

class Example{
        public static void main(String[] args) {
                System.out.println("In subpackage");
        }
}
```

Compilation
>javac -d . Example.java

Execution
>java p1.p2.Example
In subpackage

Can we create classes with predefined class name?
Yes, we can create user defined classes or custom classes with predefined class name.

Then how can we differentiate these two classes?
By using package name If there is any class defined locally with same predefined class name, we must access predefined class name with its package name. Else it is loaded from current working directory if Classpath environment variable is setup with "." operator.

If there is a class with name "String" in your current working directory, will the classes defined in that directory compiled and executed?
Programs are compiled but will not be executed. Because JVM consider main method parameter is current local String class not predefined class. To solve this problem we must access main method parameter with package name "java.lang"

What is the output from below program?

```
class String{
        public static void main(java.lang.String[] args){
                java.lang.String String = "abc";
                System.out.println(String);
        }
}
```

Q) How can we access other package classes from our package classes?
There are two ways

1. By using fully qualified name or
2. By using import keyword

Understanding "import" keyword

import keyword is used to "access" other package members from this package classes.

Actually it does not import other package members into this package; instead it shows the path of the other package member to compiler and JVM.

We have 2 syntaxes to use import statement

```
syntax:
        import packagename.*;
                or
        import packagename.classname;
```

```
For example
        import p1.*;
                or
        import p1.Example;
```

What is the difference between above two import syntaxes?

First syntax allows Compiler & JVM to access all public members (classes, interfaces & enums) of that imported package, where as second syntax allows Compiler and JVM to access only that imported class.

Rule: import statement must be placed before all class definitions, and after package statement.

How many import statements are allowed in one Java file?

In a Java file "more than one import" statements and "only one package" statement are allowed.

Rule: To access packaged members from another package class its members must be declared as public, else it leads to compile time error while access that member.

Below program shows accessing other package members

```
package p1;
public class A{
        public static void m1(){
                System.out.println("A m1");
        }
}
```

Case #1: If user class is also defined in same package, import statement and fully qualified name is optional.

```
pacakge p1;
class B{
        public static void main(String[] args) {
                System.out.println("B main");
                A.m1();
        }
}
```

Output:

```
>javac -d . A.java
>javac -d . B.java
>java p1.B
B main
A m1
```

Case #2: If the class is using from another package, either import or fully qualified name must be used. Else it leads to CE: cannot find symbol.

```
package p2;
class C{
        public static void main(String[] args) {
                System.out.println("C main");
                A.m1(); //CE: cannot find symbol Class A
        }
}
```

```
package p2;
import p1.*;
class C{
        public static void main(String[] args) {
                System.out.println("C main");
                A.m1();
        }
}
```

Output:
```
>javac -d . C.java
>java p2.C
C main
A m1
```

What is the output from below program? Is there any compile time errors?

```
//A.java – class with
                default no-arg constructor
package p1;
public class A{
        public void m1(){
                System.out.println("A m1");
        }
}
```

```
//B.java – class with
                non-public non-arg constructor
package p1;
public class B{
        B(){
                System.out.println("B constructor");
        }
        public void m1(){
                System.out.println("B m1");
        }
}
```

```
//C.java – class with
        public parameterized constructor
package p1;
public class C{
        private C(){
          Solln("C no-arg constructor");
        }
        public C(String s){
          Sopln("C String constructor");
        }
        public void m1(){
                Sopln("C m1");
        }
}
```

```
//Test.java
package p2;
import p1.*;

public class Test{
        public static void main(String[] args){
                A a = new A();
                B b = new B();
                C c1 = new C();
                C c2 = new C("abc");
        }
}
```

```
//Test.java
package p2;
import p1.X;

public class Test{
        public static void main(String[] args){
                A a = new A();
                B b = new B();
                C c1 = new C();
                C c2 = new C("abc");
        }
}
```

What is the benefit we get in using import statement over fully qualified name?
If we do not use import statement, we must use package name every wherever we are using class name or constructor. This code is considered as redundant code and also code is not readable. To solve this problem SUN introduced "import" concept So, if we use import statement we no need to use package name in referring class name and its constructor.

Below program shows above problem

```
package p2;
class Sa{
        public static void main(String[] args) {
                p1.A a1 = new p1.A();
                p1.A a2 = new p1.A();

                p1.C c1 = new p1.C("a");
                p1.C c2 = new p1.C("b");
        }
}
```

Now this code is not readable.

Below program is the conversion program with import – we used package name only once

```
package p2;
import p1.*;

class Sa{
        public static void main(String[] args) {
                A a1 = new A();
                A a2 = new A();

                C c1 = new C("a");
                C c2 = new C("b");
        }
}
```

Now this code is readable.

Give a scenario where both import and fully qualified name should be used?
If a class with same name is available in two packages, to that class from both packages we must use fully qualified name to differentiate class from another packaged class. In this case if we use just import statements compiler throws ambiguous error in accessing that class.

Below program shows above CE

//A.java

```
package p3;
public class A{
        public A (){
            Soln("p3.A constructor");
        }
}
```

//Test.java

```
package p2;
import p1.*;
import p3.*;

public class Test{
        public static void main(String[] args){
                //A a = new A(); CE:
                p1.A a = new p1.A();
                p3.A a = new p3.A();
                C c = new C("abc");
        }
}
```

Using sub package members
we must import sub package separately to access its members. Because by importing parent package sub package members are not imported vice versa is also not possible.

Why sub package members are not imported, when we import parent package?
Because parent package may have more than one sub package, so compiler and JVM cannot take decision from which sub package that class must be accessed.

//D.java

```
package p1.p4;
public class D{
        public void m1(){  Sopln("D m1" ); }
}
```

Find out CEs in the below programs

//Test.java

```
package p2;
import p1.*;

class Test{
        public static void main(String[] args) {
                A a = new A();
                D d = new D();
        }
}
```

//Test.java

```
package p2;
import p1.*;
import p1.p4.*;

class Test{
        public static void main(String[] args) {
                A a = new A();
                D d = new D();
        }
}
```

Static imports:
This feature is introduced in Java 5 to import static members of a class.

By using this feature we can access all

- non-private static members without using class name from other classes with in the package and
- protected and public members from outside package class members without using class name.

Syntax:

```
import static packagename.classname.*;
                    or
import static packagename.classname.staticmembername;
```

- first syntax allows to call all static members of the class.
- second syntax allows only to call the imported static member.

Q) What is the difference between below statements?

import p1.*;

- We can access all classes from p1 package

import p1.A;

- We can access only class A from p1 package

import static p1.A.*;

- We can access all static members of class A from p1 package
- Using this import statement we cannot access non-static members, we cannot create object, we cannot develop subclass from A class.
- for this purpose we must also write import statement separately for accessing class A as "import p1.A;"

import static p1.A.a;

- We can only access the static variable "a"
- if "a" is a non-static variable it leads to CE: cannot find symbol "static a"

import static p1.A.m1;

- We can only access the static method "m1"
- if "m1" is a non-static method it leads to CE: cannot find symbol "static m1"

Below program shows accessing static members of a class with "static import" concept.

//Example.java

```
package p1;
public class  Example {
        public static int a = 10;
        public int x = 20;

        public static void m1(){
                Sopln("m1");
        }
        public void m2(){
                Sopln("m2");
        }
}
```

//Sample.java

```
package p2;
import static p1.Example.*;

public class Sample {
   public static void main(String[] args){
        //accessing static members without using classname
        System.out.println(a);
        m1();

        //accessing static members with classname
        System.out.println(Example.a); //CE:
        Example.m1(); //CE:

        //accessing non-static members
        Example e = new Example(); CE:
        System.out.println(e.x);
        e.m2();
   }
}
```

Note: To solve above compile time errors we must also import class Example with normal import statement as "import p1.Example;"

Q) If the current class also contains the imported static member, how can we differentiate both of them? We must use fully qualified name of the that static member that is "packagename.classname.staticmembername" else current class static member is used.

Check below program

//Sample.java

```
package p2;
import static p1.Example.*;
class Sample{
        static int a = 70;

        public static void main(String[] args){
                System.out.println(a);
                //System.out.println(Example.a); //CE: cfs varaible Example
                System.out.println(p1.Example.a);

                m1();
        }
}
```

Q) Write a program to print data without using class name System. You should use only "out.println()" to print "hi", "hello", "hru?"

//Test.java

```
package p2;
import static java.lang.System.*;
class Test{
        public static void main(String[] args){
                out.println("Hi");
                out.println("Hello");
                out.println("Hru?");
        }
}
```

Find out valid syntaxes from the below list

```
import java.lang.*;
import java.lang.System;
import java.lang.System.*;
import java.lang.System.out;

import static java.lang.System.*;
import static java.lang.System;
import static java.lang.System.out;
import static java.lang.System.out.*;
static import java.lang.System.out;
```

Q) In the below program at what line number CE is raised?

//SISyntax2.java

```
1. import p1.A.*;

2. class SISyntax2{
3.      public static void main(String[] args){
4.              m1();
5.              A.m1();
6.              p1.A.m1();
7.      }
8. }
```

Java source file structure

In a java file we can have package statement, import statement, interface, abstract class, concrete class, final class, main method class, and documentation.

All these are organized as shown below according to coding standards and compiler rules

Documentation section
Package statement
Import statement
Interface
Abstract class
Concrete class
Final class
Main method class

In-built packages

- SUN given packages are called in-built packages, and developer given packages are called custom or user defined packages.
- SUN also organized all predefined classes, interfaces and enums in packages.
- We have two root packages for in-built packages
- They are
    - java and javax

- java package has basic and fundamental or core classes and interface for design of java programming language.
- javax package has extension classes and interfaces.
    - javax stands for "Java eXtension"
- Below diagram shows the Java SE important sub packages of *java* and *javax* packages

java	**javax**
|- lang	|- swing
|- io	|- sql
|- net	|- xml
|- util	|- naming
|- awt	|- transaction
|- applet	
|- sql	
|- rmi	
|- math	
|- text	

Chapter 8

Accessibility Modifiers

- In this chapter, You will learn
  - Different levels of accessibility permissions
  - Default accessibility modifier of class and its members
  - Default accessibility modifier of interface and its members
  - Working with Accessibility modifiers with package
  - Rule of protected accessibility modifier
- By the end of this chapter- you will learn how to protect your data and logic at different levels.

Interview Questions

By the end of this chapter you answer all below interview questions

- Definition of Accessibility modifier
- Different levels of Accessibility permission levels.
- Accessibility modifier keywords.
- What are the accessibility modifier keywords allowed for a class?
- What are the accessibility modifiers allowed for a class members including inner classes?
- What is the default accessibility modifier of class and its members?
- What is the default accessibility modifier of interface and its members?
- Why private and protected accessibility modifiers are not allowed for outer class?
- Sample programs with and without package.

Accessibility Modifiers

Definition
The keywords which define accessibility permissions are called accessibility modifiers.

Different levels of Accessibility permission levels
Java supports four accessibility levels to define accessibility permissions at different levels.

In Java, we have below 4 accessibility levels

1. only within the class
2. only within the package
3. outside the package but only in subclass by using the same subclass object
4. from all places of project

Accessibility modifier keywords
To define the above four levels we have 3 keywords

1. **private**: The class members which have *private* keyword in its creation statement are called private members. Those members are only accessible with in that class.

2. **protected**: The class members which have *protected* keyword in its creation statement are called protected members. Those members can be accessible with in package from all classes, but from outside package only in subclass that too only by using subclass object (This rule is only for non-static protected members. Static protected members can be accessible by using same class name or by using subclass name).

3. **public**: The class and its members which have *public* keyword in its creation statement are called public members. Those members can be accessible from all places of Java application.

Note: if we do not use any of the above 3 accessibility modifiers, *package* level is the default accessibility modifier of class and its members. It means that class and its members are not accessible from outside of that package.

Q) What are the accessibility modifiers allowed for a class?
default and public. The keywords private and protected are not allowed to outer class because it is not the member of another class.

Q) What are the accessibility modifiers allowed for a class members including inner classes?
All 4 accessibility modifiers are allowed.

Q) What is the default accessibility modifier of class and its members?
package level

Q) What is the default accessibility modifier of interface and its members?

- The default accessibility modifier of interface is *package* level and
- It's member's default accessibility is *public*

Below application shows above points

```
//Example.java
public class Example{
        private         int a = 10;     //<= private variable
                        int b = 20;     //<= package level variable
        protected       int c  = 30;    //<= protected variable
        public          int d  = 40;    //<= public variable

        public static void main(String[] args){
                Example e = new Example();
                System.out.println("a: "+e.a);
                System.out.println("b: "+e.b);
                System.out.println("c: "+e.c);
                System.out.println("d: "+e.d);
        }
}
```

```
D:\Naresh IT\HariKrishna\06AM>javac Example.java
D:\Naresh IT\HariKrishna\06AM>java Example
a: 10
b: 20
c: 30
d: 40
```

The above program is compiled and executed without errors, but outside of the class only non-private members are accessible. If we access private members from outside class members it leads to compiler throws CE:

```
//Sample.java
public class Sample{
        public static void main(String[] args){
                Example e = new Example();
                //System.out.println("a: "+e.a);  //CE: a has private access in Example
                System.out.println("b: "+e.b);
                System.out.println("c: "+e.c);
                System.out.println("d: "+e.d);
        }
}
```

```
D:\Naresh IT\HariKrishna\06AM>javac Sample.java
D:\Naresh IT\HariKrishna\06AM>java Sample
b: 20
c: 30
d: 40
```

Below diagram shows the four levels of accessibility permissions with packages

```
//A.java
package p1;    //<= we define specific package level using package keyword.
public class A{
        private         int a = 10;
                        int b = 20;
        protected       int c = 30;
        public          int d = 40;

        public static void main(String[] args){
                A a = new A();
                System.out.println("a: "+a.a);
                System.out.println("b: "+a.b);
                System.out.println("c: "+a.c);
                System.out.println("d: "+a.d);
        }
}
```

Compilation:

D:\Naresh IT\HariKrishna\06AM>javac -d . A.java **<=**We must use "–d" option to create package

Execution:

D:\Naresh IT\HariKrishna\06AM> java p1.A **<=** we must use package name to execute class

a: 10
b: 20
c: 30
d: 40

Accessing above four members from another class with in the same package.

```
//B.java
package p1;
class B{
        public static void main(String[] args)  {
                A a = new A();
               //System.out.println("a: "+a.a);          CE: a has private access in p1.A
                System.out.println("b: "+a.b);
                System.out.println("c: "+a.c);
                System.out.println("d: "+a.d);
        }
}
```

Within the same package except private variable all other variables are accessible

Compilation:

D:\Naresh IT\HariKrishna\06AM>>javac -d . B.java

Execution:

D:\Naresh IT\HariKrishna\06AM>>java p1.B

b: 20 c: 30 d: 40

Accessing above four members from another package from subclass.

```
//C.java
package p2;
import p1.A;            <= to use from another package members we must use import statement
class C extends A {     <= class C is created as subclass of A class
        public static void main(String[] args) {
                A a = new A();
                //System.out.println("a: "+a.a);        //CE: a has private access in p1.A
                //System.out.println("b: "+a.b);        //CE: b is not public in p1.A
                //System.out.println("c: "+a.c);        //CE: c has protected access in p1.A
                System.out.println("d: "+a.d);

                C c1 = new C();
                //System.out.println("a: "+c1.a);
                //System.out.println("b: "+c1.b);
                System.out.println("c: "+c1.c);
                System.out.println("d: "+c1.d);
        }
}
```

Only *protected* and *public* members are accessible.

Compilation:
D:\Naresh IT\HariKrishna\06AM>>javac -d . C.java

Execution:
D:\Naresh IT\HariKrishna\06AM>>java p2.C
c: 30 d: 40

Accessing above four members from another package from normal class.

```
//D.java
package p2;
import p1.A;
class D {
        public static void main(String[] args) {
                A a = new A();
                //System.out.println("a: "+a.a);        //CE:  a has private access in p1.A
                //System.out.println("b: "+a.b);        //CE:  b is not public in p1.A
                //System.out.println("c: "+a.c);        // CE:  c has protected access in p1.A
                System.out.println("d: "+a.d);

                C c1 = new C();
                //System.out.println("c: "+c1.c);       // CE:  c has protected access in p1.A
                System.out.println("d: "+c1.d);
        }
}
```

It is not subclass so only *public* members are accessible.

Compilation:
D:\Naresh IT\HariKrishna\06AM>>javac -d . D.java

Execution:
D:\Naresh IT\HariKrishna\06AM>>java p2.D
d: 40

Chapter 9

Methods and Types of Methods

- In this chapter, You will learn
  - Definition of method
  - Method terminology
  - Types of methods
  - Rules in calling different methods
  - Methods execution control flow
  - Modifiers allowed for a method

- By the end of this chapter- you will learn defining, declaring, calling methods, and its logic execution control flow.

Interview Questions

By the end of this chapter you answer all below interview questions

- Method definition
- Method terminology
  - Method prototype
  - Method body and Logic
  - Parameters and arguments
  - Method Signature
  - Method return type
  - Method Modifier
- Main method's terminology
- Defining, declaring and invoking/calling a method
- Can we define a method inside another method?
- Types of methods
  - Static and non-static
  - Void and non-void
  - Parameterized and Non-Parameterized
  - Final
  - Abstract
  - Native
  - Synchronized
  - Strictfp
- What are the modifiers allowed for a method?
  - CE: modifier not allowed here
- Rules in calling static and non-static methods
  - CE: non-static method cannot be referenced from static context
- Rules in calling parameterized and non-parameterized methods
  - CE: cannot find symbol
- Rules in calling void and non-void methods
  - CE: cannot return a value whose result type is void
  - CE: missing return statement
  - CE: missing return value
  - CE: incompatible types
  - CE: possible loss of precision
  - CE: void type is not allowed here
- Need of break, continue, return statement
- Rules on above three statements
  - CE: unreachable statement
  - CE: break outside loop or switch
  - CE: continue outside loop

Methods and Types of Methods

Definition
Method is a sub block of a class that is used for implementing logic of an object's operation.

Rule: logic must be placed only inside a method, not directly at class level.
If we place logic at class level compiler throws error.

- So at class level we are allowed to place only variable and method creation statements.
- The logical statements such as method calls, validations, calculations, and data printing related statements must be placed inside method, because these statements are considered as logic.

Find out CEs from the below program

```
//Example.java
class Example{
        static int a = 10;
        static int b = a + 10;

        a = 20;
        System.out.println(a + " ... " + b);

        m1();

        if (true){
                System.out.println("Hi");
        }
        public static void main(String[] args){
                System.out.println(a + " ... " + b);

                m1();

                if (true){
                        System.out.println("Hi");
                }
        }
        static void m1(){
                System.out.println("m1");
        }
}
```

class Level

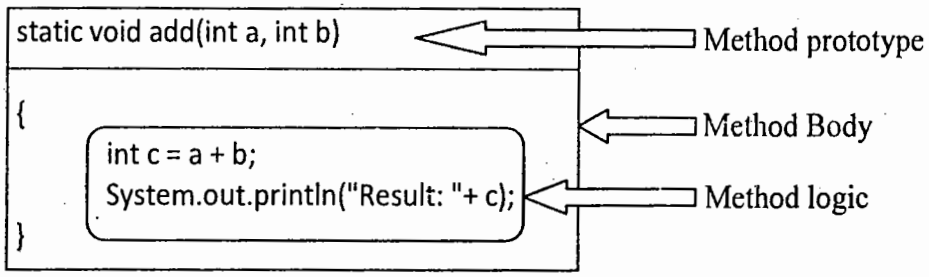
Method Level

Method Terminology

1. **Method prototype**: The head portion of the method is called method prototype.

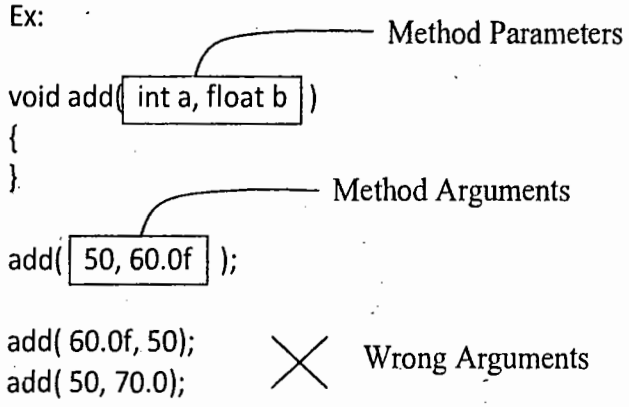
2. **Method body and logic**: The "{ }" region is called method body, and the statements placed inside method body is called logic.

   Ex:



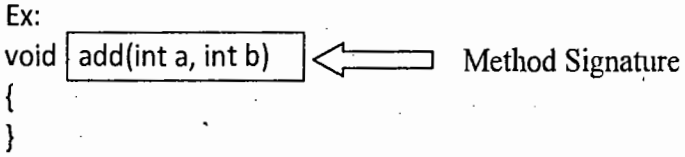
3. **Method parameters and arguments:**
   - The *variables declared* in method parenthesis "( )" are called parameters. We can define method with ZERO to 'n' number of parameters.
   - The *input values passing* to parameters are called arguments. In method invocation we must pass arguments according to method parameters order and type.

   Ex:



4. **Method signature**: The combination of [Method name + parameters list] is called method signature.

   Ex:

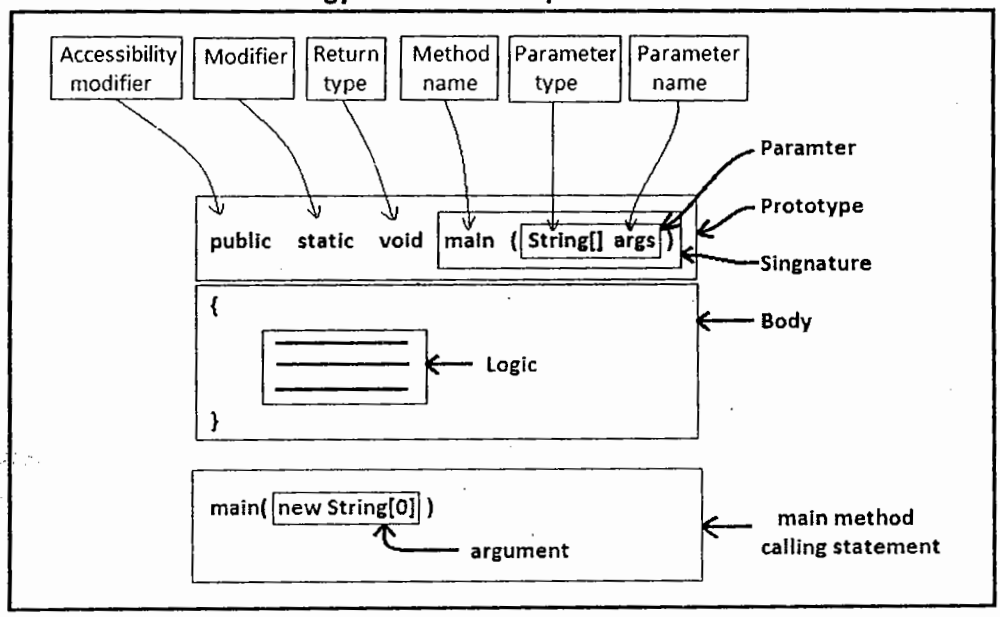


5. **Method return type:**
   - The datatype keyword that is placed before method name is called method return type. It tells to Compiler, JVM and developer about the type of the value is returned from this method after its execution. If we don't want to return value for a method we must use *void* keyword as return type for that method.

***void* return type keyword**

- If we do not want to return any value from a method, we must use "void" as return type. It tells that the method does not return any value.
- If we want to return some value, we must place datatype keyword as return type. For example if we want to return integer value we must place either 'int' or 'long'.
  Note: *byte, short* are not allowed because by default integer value type is *"int"*

Main method's terminology with all above parts



Method creation and execution syntax

Defining, declaring and invoking/calling a method:

We can create method in two ways

1. Method creation with body - The process of creating method with body is called method definition. Technically this method is called *concrete* method.

Syntax:

```
      OP                OP        M              M                OP
<Accessibility Modifier> <Modifier> <return type> <method name>(<parameters list>)
{
                    ----------------
M                   ----------------   OP
                    ----------------

}; OP
```

For Example:

```
public static void add(int a , int b){
        System.out.println( a + b );
};
```

```
void add(){

}
```

2. Creating a method without body - Creating a method without body is called declaring a method / method declaration. Technically this method is called abstract method.
Rule: In method declaration the modifier "***abstract***" is mandatory and also should be terminated with ';'.

Syntax:

```
        OP                    M        M                    M          OP          M
<Accessibility Modifier> abstract <return type> <method name>(<parameters list>) ;
```

For example::

```
public abstract void add(int a, int b);
```

```
abstract void add();
```

Method invocation/ method calling/ method execution
Sending cursor to a method body to execute its logic is called method invocation or calling.
Syntax:

```
            M              M/OP     M
        <methodName>(<argumets list>) ;

arguments are mandatory only if method has parameters else we should not pass.
```

Below program explains defining methods and calling them from main method.

```
class Example
{
        static void m1(int a)
        {
                System.out.println("m1");
        }

        static void m2()
        {
                System.out.println("m2");
        }
```

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

                m1(50);

                m2();

        }

}
```

Conclusions from the above program

1. When we call a method control send to that method.
2. If we pass argument that value is stored in parameter variable.
3. After method execution that parameter variable is destroyed and control is sent back to calling method.
4. Control is sent back to calling method with value if method return type is not void.

Types of methods

Basically concrete methods are divided into 3 types

1. Based on *static* modifier we have two types of methods
   a. static methods
   b. non-static methods

2. Based on *return type* we have two types of methods
   a. void methods
   b. non-void methods

3. Based on *parameter* we have two types of methods
   a. parameterized methods
   b. non-parameterized methods.

Static and Non-static methods

If a method has static keyword in its definition (prototype) then that method is called static method, else it is called non-static method.

Ex:

//static method
static void m1()
{
 	
}
	//non-static method
	void m1()
	{
	
	}

Calling static and non-static methods

We can call static methods directly from main method, but we cannot call non-static methods directly from main method. It leads CE: "***non-static method cannot be referenced from static context***", because class level members will not get memory directly. JVM provides memory only if we use either "static or new" keywords.

Below program shows calling static and non-static methods

//Example.java

```
class Example {

        static void m1(){
                System.out.println("In m1");
        }

        void m2() {
                System.out.println("In m2");
        }
```

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

                System.out.println("In main");

                m1();
                //m2(); CE: non-static method m2 cannot be referenced from static context.

                //Below we are using new keyword to provide memory for m2() method.
                Example e = new Example();

                //It gets memory with reference to "e" variable.
                //So we must call it as shown below.
                e.m2();
        }
}
```

Void and Non-void methods

If the method return type is 'void', it is called void method; else it is called non-void method.

Rule: Non-void method must return a value after its execution. Also that value type must be compatible with method return type and its range must be less than or equals to method return type. Else it leads to compile time error.

For example:

//**static void** method
static void m1()
{
 	
}
	//**non-static void** method
	void m2()
	{
	
	}

//**static non-void** method
static int m1()
{
return 10;
}
	//**non-static non-void** method
	double m2()
	{
	return 23.45;
	}

Q) Are statements optional or mandatory in a method?

In void methods statements are optional, but in non-void methods return statement is mandatory with value range **less than or equals to** method return type range.

Rule: If we do not place return statement in non-void methods compiler throws **CE:**"*missing return statement*"

Types of return statements:

We have two types of return statements

1. return;
2. return <value>;

Rule on return statements

- "*return;*" is only allowed in void methods and constructor, and it is optional.
- "*return <value>;*" is only allowed in non-void methods, and it is mandatory.

Find out compile time errors in below method definitions.

```
1. void m1(){}

2. void m1(){return; }

3. void m1(){
     return 10;
   }

4. int m1(){}

5. int m1(){ return;}

6. int m1(){ return 10;}
7. int m1(){ return 'a';}
8. int m1(){ return 10.345;}

9. int m1(){ return true;}
```

Calling Void and Non-void methods

In general, methods are called in three ways

1. Directly -> m1();
2. As variable initialization statement -> int x = m1();
3. As Sopln() argument -> Sopln(m1());

Non-void method can be called in all three ways.

- In the first way the returned value is lost
- In the second way the returned value is stored in the destination variable
- In the third way the returned value is printed on console

Only in second way we can use returned value in further program logic.

Void method can only be called in first way.

If we call it either in second or third ways it leads to CE.

Check below program

```
class Example
{ static void m1()                          p s v main(String[] args)
  {                                         {
  Sopln("m1");
  }                                           m1();

  static int m2()
  {                                           m2();
  Sopln("m2");
  return 10;                                  int x = m2();   ⇨ int x = 10;      x [10]
  }                                   10
                                      10
                                      10
                                              Sopln(m2());    ⇨ Sopln(10);

                                              // int y = m1();  CE: incompatible types
                                              // Sopln(m1());   CE: 'void' type not allowed here

                                            }
                                          }
```

```
O/P:
====
m1
m2
m2
m2
10
```

Purpose of return statement

Basically return statement is used to terminate method execution and for sending control back to calling method.

- "return;" sends control back to calling method without value.
- "return <value>;" sends control back to calling method with value.

What is the output from the below program:

```
class Example{
        static void m1(int a){
                System.out.println("Before if");
                if(a == 10){
                        System.out.println("In if");
                        return;
                }
                System.out.println("after if");
                System.out.println("Hi");
        }
        static int m2(int a){
        System.out.println("Before if");
                if(a == 10){
                        System.out.println("In if");
                        return a + 10;
                }
                System.out.println("After if");
                System.out.println("Hi");
                return 50;
        }
```

```
        public static void main(String[] args){
            m1(10);
            m1(20);

            m2(10);
            m2(20);
        }
    }
```

3. Parameterized and Non-parameterized methods

If a method is created with parameters, it is called parameterized method, else it is called non-parameterized | no-arg method | ZERO arg method.

Ex:

//non-parameterized	//parameterized

static void m1()
{
}
	static void m1(int x)
	{
	}

Calling parameterized and non-parameterized methods

Parameterized methods must be called by passing the parameter type argument, else it leads to CE: cannot find symbol.

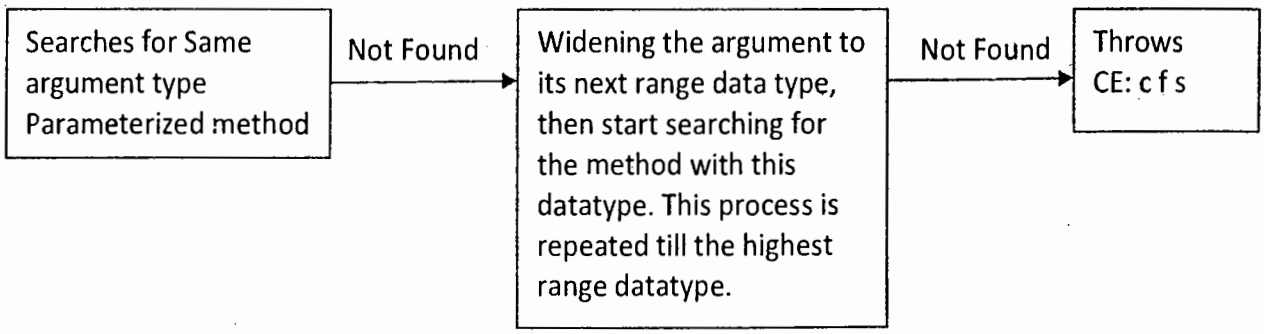
Check below program

```
class Example
{
        static void m1()
        {
                System.out.println("m1");
        }

        static void m2(int a)
        {
                System.out.println("m2");
        }
```

```
        public static void main(String[] args){
                m1();
                //m1(50); CE: cannot find symbol

                //m2(); CE: cannot find symbol
                m2(50);
                m2('a');
                //m1(50.34); CE: cannot find symbol
                //m1(true);    CE: cannot find symbol
        }
}
```

Conclusion: Searching for Parameterized method definition

Compiler searches for parameterized method definition as shown below



The above flow chart is common for both *primitive* and *referenced datatypes*.

For primitive data types the highest range datatype is *double*, and for referenced data type the highest range datatype is *java.lang.Object*, it is the super class of all referenced datatypes.

//Passing object as an argument and return type –What is the output from below program?

```
class A{}

class Example{

        static void m1(A a){
                System.out.println("m1");
        }

        static A m2(String s){
                System.out.println("m2");
                return new A();
        }

        public static void main(String[] args){
            A a1 = new A();
            m1(a1);

            m2("Hari");
            A a2 = m2("Krishna");

            System.out.println( m2("NareshIT") );
        }
}
```

Note:

- If we pass primitive values as an argument, the value is passed directly and is stored in parameter variable.
- If we pass object as an argument, it's reference is passed not its memory, and that reference is stored in parameter variable. Then that parameter variable is also pointing to the same object as shown above.

Q) What are the modifiers not allowed for a method?

Except *transient* and *volatile* all other 9 modifiers are allowed.

Chapter 10

Variables and Types of Variables

- In this chapter, You will learn
  - Definition of variable
  - Limitation of variable
  - Variables creation syntax
  - Types of variables
  - Local variable and its rules
  - Modifiers allowed for variables
  - 8 Types of members defined in a single class and their execution control flow

- By the end of this chapter- you will learn defining, declaring, calling variables, and its execution control flow.

Interview Questions

By the end of this chapter you answer all below interview questions

- Definition
- How can we create a variable?
- Limitation on variable
- Defining, declaring, initializing, reinitializing, and calling/using a variable
- Types of variables
    - Method Level
        - Parameter
        - Local
            - final
    - Class Level
        - static
        - non-static
            - final
            - transient
            - volatile
- Rules on local variables and Parameters?
- What are the modifiers allowed for a variable?
- Default values for variables
- Life-time and scope of a variable
- How many members we can define inside a class?

Variables and Types of Variables

Definition

Variable is a named memory location used to store data temporarily. During program execution we can modify that data.

How can a variable be created?

A variable can be created by using Datatype. As we have two types of datatypes we can create two types of variables

1. **Primitive variables** - These variables are created by using primitive datatypes.
2. **Referenced variables** - These variables are created by using referenced datatypes.

The difference between primitive and referenced variables is "primitive variables stores data directly, where as referenced variables stores reference of the object, not direct values".

Note: As per compiler and JVM we do not have a separate name called "referenced variable". It means, the variables created by using referenced datatype are also considered as like normal variables only.

Below program shows creating primitive and referenced variables

```
//Example.java
class Example
{
        int x = 10;
        int y = 20;
}
```

```
//Test.java
class Test{
        public static void main(String[] args) {
                //primitive variables
                int     p = 50;
                int     q = m1();

                //referenced variables
                String  s1 = "a";
                String  s2 = new String("a");

                Example e = new Example();
        }
        static int m1() {
                return 60;
        }
}
```

Referenced variables are initialized with object reference that is created and returned by "new" keyword, as shown in the left diagram.

Limitation of variable

It can only store single value at a time. If we assign new value, old value is replaced with new value. It means, always it returns latest modified value.

So

- If we modify primitive value previous value is replaced with new value
- If we modify referenced variable previous object's reference is replaced with new object's reference and now this referenced variable is referencing to this new object.

Show memory structure from the below program and also output

```
class Sample {
        public static void main(String[] args) {
                int a = 50;
                System.out.println("a: "+a);

                a = 70;
                System.out.println("a: "+a);

                Example e1 = new Example();
                System.out.println("e1: "+e1);

                e1 = new Example();
                System.out.println("e1: "+e1);

        }
}
```

Output:

a: 50
a: 70
e1: Example@addbf1
e1: Example@42e816

Conclusion:

- The value of primitive variable is mathematical data based on its data type.
- The value of referenced variable is its class object's reference.

Defining, declaring, initializing, reinitializing, and calling/using a variable

Defining a variable

Variable creation with value is called defining a variable.

Syntax:

```
  OP                          OP        M          M        M   M   M
<Accessibility Modifier> <Modifier> <datatype> <variablename> = <value> ;
```

Ex:

```
public static int        x = 10;
public static Example  e = new Example();
```

Declaring a variable

Variable creation without value is called declaring a variable.

Syntax:

```
  OP                          OP        M          M        M
<Accessibility Modifier> <Modifier> <datatype> <variablename> ;
```

Ex:

```
public static int x;
public static Example  e ;
```

Assigning and reassigning a variable

Storing a value in a variable at the time of its creation is called initializing a variable.
Storing a value in a variable after its creation is called *assigning* a value to a variable.

Syntax:

```
     M          M   M    M
<variablename> = <value>;
```

Ex:

```
//declaring variables
int p;
Example e;

//assigning variables
p = 50;
e = new Example();

// reassignment
p = 70
e = new Example();
```

Calling a variable

Reading a value from a variable is called calling a variable.

```
Ex:
        int x = 10;

        //calling x variable for printing
        System.out.println(x); // 10

        //calling x variable for initializing y
        int y = x;
```

Q) What is the difference between executing a variable and executing a method?

Executing a variable means creating variable memory location with initialized value. JVM executes variables automatically when variable creation statement (definition or declaration) is appeared in the program.

Executing a method means executing method logic. JVM does not execute method logic automatically. It executes method only when that method calling statement is appeared, not by using just definition.

Q) What is the difference between variable calling and method calling?

Calling a variable means *reading its value*.

Calling a method means *executing that method logic*.

```
Ex:
class Test{

        static void m1(){
                System.out.println("m1");
        }

        public static void main(String[] args){
                int x = 10;

                //from below statement x variable value is read, variable is not executed again.
                System.out.println(x); //10          <= variable calling statement

                //from the below statement method is executed.
                m1();           <= method calling statement
        }
}
```

Types of Variables

Based on class scopes variables are divided in two types

1. Local Variables, parameters
2. Class Level Variables

- The variables created inside a method (or) block are called local variables
- The variables created at class level are called class level variables.

2 types of class level variables

Class level variables are divided into two types, based on the time they are getting memory location. They are

- Static variables
- Non-static variables

Definitions

The class level variable which has static keyword in its creation statement is called static variable, else it is called non-static variable.

Memory location of all above three variables

- *Local variables* get memory location when method is called and their creation statement is executed. They get memory with respect to method, so they are also called *method variables*. Local variables are automatically created when method is executing and are destroyed automatically after method execution is completed, so they are also called *auto variables*.
- *Static variables* get memory location when class is loaded into JVM. They get memory with respect to class name, so they are also called "*class variables*" also called "*Fields*".
- *Non-static variables* get memory location when object is created using *new* keyword. They get memory with respect to object, so they are also called "*object variables* or *instance variables* or *properties* or *attributes*" are also called "*Fields*".

Below diagram shows different scopes in class and all above three variables creation.

```
class Example{
        //static variables
        static int a = 10;
        static int b = 20;

        //non-static variables
        int x = 30;
        int y = 40;

        public static void main(String[] args)
        {
                //local variables
                int p = 50;
                int q = 60;
        }
}
```

Q) How many variables are created in above program?

A) 5 variables. They are *a, b, args, p, q*.

Variables **x, y** are not created, because object is not created.

Local variables and its rules

While working with local variables, we must follow below 3 rules

Rule #1: Local variable cannot be accessed from another method. Because its scope is restricted only within its method, also we cannot guarantee variable creation, because that method may or may not be called. It leads to **CE: cannot find symbol**

For example:

```
class Example
{
        public static void main(String[] args)
        {
                int a = 10;
                System.out.println("a: "+a);
        }
        static void m1()
        {
                System.out.println("a: "+a);   CE: cannot find symbol
        }                                          symbol: variable a
}
```

Rule #2: Local variable should not be accessed without initialization.

It leads to **CE: "variable might not have been initialized"**

```
class Example {
        public static void main(String[] args){
                int a = 10;
                int b;

                System.out.pritnln("a: "+ a);
                //System.out.pritnln("b: "+b);   //CE: variable b might not have been initialized

                a = a + 10;
                //b = b + 10;   //CE: variable b might not have been initialized

                b = 20;
                System.out.pritnln("b: "+ b);

                b = b + 10;

                System.out.pritnln("a: "+ a);
                System.out.pritnln("b: "+ b);
        }
}
```

Rule #3: local variable must be accessed only after its creation statement; because method execution is sequential execution from top to bottom. If we access it before its creation statement it leads CE: "**cannot find symbol**"

```
class Example{
        static void m1(){
                System.out.println("a: "+a);
                int a;
                System.out.println("a: "+a);
                a = 10;
                System.out.println("a: "+a);
        }
        static void m2(){
                System.out.println("a: "+a);
        }
}
```

Q) How can we access a local variable value from other methods?

A) There are two ways

1. Pass it as an argument
2. Store it using a class level variable

- If we want to use a value in one or two methods, store it using local variable in one method and pass it as an argument to another method. This approach is recommended to save memory.
- If we want to use a value from multiple methods of a class, store it using class level variables.

Q) What is the output from the below program?

```
class Example{
        static void m1(){
                int p = 10;
                System.out.print(p);
                m2();
        }
        static void m2(){
                int q = p + 10;   CE: c f s
                System.out.print(q);
        }
        public static void main(String[] args)
        {
                m1();
        }
}
```

1. 10 10
2. 10 20
3. CE
4. RE

Q) In the above program, if we create p variable **at class level as static, what is the output?**

A) No CE, output is: 10 20

Q) In the above program, if we create p variable **at class level as non-static, what is the output?** A) It leads to **CE**: non-static variable p cannot be referenced from static context

Class Level variables

We must create class level variable only if we want to access a value throughout the class from all its methods.

Q) What is the output from the below program?

```
class Example{
        int x = 10;

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

Choose one option

1. 10
2. CE
3. RE
4. No output

Rule: Non-static variables and methods must be accessed with object from static methods; else it leads to above CE: *non-static variable cannot be referenced from static context*, because non-static variable and non-static method does not get memory location directly at the time of class loading.

Below program shows creating static, non-static, and local variables and accessing them from main method (static method).

```
class Example{
        static int a = 10;
        static int b = 20;

        int x = 30;
        int y = 40;

        public static void main(String[] args) {
                int p = 50;
                int q = 60;

                System.out.println("a: "+a);
                System.out.println("b: "+b);

                //System.out.println("x: "+x); CE:
                //System.out.println("y: "+y); CE:

                Example e = new Example();
                System.out.println("x: "+e.x);
                System.out.println("y: "+e.y);

                System.out.println("p: "+p);
                System.out.println("q: "+q);
        }
}
```

Q) Can we declare *local variable* or *parameter* as static?
No, local variables can't be declared as static it leads to CE: *illegal start of expression*. Because local variable should get memory location only if method is called. But if we declared as static, it should get memory at the time of class loading, this is violating contract, so it leads to compile time error.

For Example

```
class Example{
        static int a = 10;

        static void m1(){
                int p = 20;
                static int q = 30;   CE: illegal start of expression
        }
}
```

Q) The variables created inside a static method are static? => No, they are still local
Q) The variables created inside a non-static method are non-static? => No, they are still local

Final variables
The class level or local variable that has final keyword in its definition is called final variable.

Rule: once it is initialized by developer its value cannot be changed. If we try to change its value it leads to compile time error.

Below program shows creation final variables

```
class Example{

        static int a = 10;              <= normal static variable
        static final int b = 20;        <= final static variable

        int x = 30;                     <= normal non-static variable
        final int x = 40;               <= final non-static variable

        public static void main(String[] args) {

                int p = 50;             <= normal local variable
                final int q = 60;       <= final local variable

                //q = 70; CE: cannot assign a value to final variable q

                final int r;
                r = 70;

                //r = 80; CE: variable r might already have been assigned
        }
}
```

transient variables

The class level variable that has transient keyword in its definition is called transient variable.

Rule: local variable cannot be declared as transient. It leads to CE: illegal start expression

Find out CE in program if any?

```
class Example{
        static transient int x = 10;
        transient int y = 20;

        static void m1(){
                transient int z = 30;
        }
}
```

Note: We declare variable as transient to tell to JVM that we do not want to store variable value in a file in object serialization. Since local variable is not part object, declaring it as transient is illegal. For more details on object serialization and transient variable refer IOStreams material.

Volatile variable:

The class level variable that has volatile keyword in its definition is called volatile variable.

Rule: local variable cannot be declared as volatile. It leads to CE: illegal start expression

Find out CE in the below program if any?

```
class Example{
        static volatile int x = 10;
        volatile int y = 20;

        static void m1(){
                volatile int z = 30;
        }
}
```

Note: We declare variable as volatile to tell to JVM that we do not want to modify variable value concurrently by multiple threads. If we declare variable as volatile multiple thread are allowed to change its value in sequence one after one.

Since local variable is not directly accessible by thread, declaring it as volatile is illegal.
For more details on volatile and synchronized keywords refer Multithreading material.

Q) What are the modifiers allowed for a variables and methods?

Modifiers	private	Protected	public	static	final	abstract	native	volatile	transient	synchronized	strictfp
Local Variable	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
Class Level Variable	✓	✓	✓	✓	✓	✗	✗	✓	✓	✗	✗
Method	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓
class	✗	✗	✓	✗	✓	✓	✗	✗	✗	✗	✓
interface	✗	✗	✓	✗	✗	✓	✗	✗	✗	✗	✓

Initial or default values of variables

- Default values are application only for class level variables.
- JVM initializes all types of class level variables with default values based its datatype even though it is assigned with explicit value.

What is the output from below program?

```
class Example{
        static int a;
        static boolean b;
        static String s;

        public static void main(String[] args){
                System.out.pritnln(a);  //0
                System.out.pritnln(b);  //false
                System.out.pritnln(s);  //null
        }
}
```

Life-time and scope of a variable

+ Lifetime is the time period between variable creation and destruction.
+ Scope is the region in which it can accessible.

Local variable life time and scope

- Local variable gets life only when its method is called and its variable creation statement is executed. It is destroyed automatically once method execution is completed.
- Its scope is only within its method after its creation statement.

How many variables are created in the below program?

Find out CE if any?

```
class Example{
        static void m1(int p){
                int q = 10;

                if (p == 10){
                        int r = 20;

                        System.out.println(p);
                        System.out.println(q);
                        System.out.println(r);
                }
                System.out.println(p);
                System.out.println(q);
                System.out.println(r);
        }
        public static void main(String[] args){
                Q) How many variables are created from the below method calls?
                m1(10);
                m1(20);
        }
}
```

Static variable life time and scope

- Static variable gets life when class is loaded. It is destroyed either if class is unloaded from JVM or if JVM is destroyed.

- Its scope is throughout class, and also in the place where class is accessible. It means where ever class is available there static variable is available provided it is public.

```
class Example{
        static int a = 10;
        static void m1(){
                System.out.pritnln(a); //10
        }
        public static void main(String[] args){
                System.out.pritnln(a);  //10
        }
}
```

Non-static variables life time and scope

- Non-Static variable gets life when object is created. It is destroyed when object is destroyed. object is destroyed when it is unreferenced or its referenced variable is destroyed.

- Its scope is the scope of object, object is available only if its referenced variable is available.

```
class Example{
        int x = 10;
        static void m1(){
                Example e1 = new Example();
                System.out.pritnln(e1.x);
        }
        public static void main(String[] args){
                Example e2 = new Example();
                System.out.pritnln(e2.x);

                System.out.pritnln(e1.x);
        }
}
```

In the above program you have one compiler time error, can you find it out?

You cannot access ***e1*** variable in main() method because it is local to m1() method. So main method ***e1.x*** statement leads to CE: cannot find symbol
symbol: variable e1

Q) How many members we can define inside a class ?
A) We can define **9** types of members in a single class

They are:

static members	non-static members
1. static variables	5. non-static variables
2. static blocks	6. non-static blocks
3. static methods	7. non-static methods
4. main method	8. constructors

Syntaxes

Static variable *Ex:* static int a = 10;	**Non-Static variable** *Ex:* int x = 10;

Static block
Ex:
static{
}
	Non-Static block
	Ex:
	{
	}

Static method
Ex:
static void m1(){
}
	Non-Static method
	Ex:
	void m1(){
	}

main method
Ex:
public static void main(Sting[] args){
}
	constructor
	Ex:
	Example(){
	}

Q) Why two types of members are given in OOPS?

- Static members are meant for storing data and operate that data common to all instances of an object
- Non-static members are meant for storing data and operate that data separately and specific to every instance of an object.

For more details on static and non-static members refer below two chapters

Chapter 12:- Static members and their control flow
Chapter 13:- Non-Static members and their control flow

Chapter 11

JVM Architecture

- In this chapter, You will learn
  - Definition of VM and JVM
  - JVM Block diagram
  - Runtime areas
  - Class Loader subsystem
  - Thread and StackFrame architecture
  - Five phase diagram
  - JVM architecture with all types of variables and methods

- By the end of this chapter- you will learn defining, declaring, calling variables, and its execution control flow.

Interview Questions

By the end of this chapter you answer all below interview questions

- Definition of JVM
- JVM Architecture block diagram
- JVM Runtime areas
- What does happen when we run "java" command
- Five phases in program compilation and execution
- ClassLoader sub system architecture
- Thread architecture
- Stack Frame architecture
- Program execution process in stack
- JVM architecture with all types of variables and method execution

JVM Architecture

In this chapter I presented only the essential information. To get more detailed information on JVM architecture refer "*SUN specification*" or a text book "*Inside the JVM*, by Bill Venners".

The source of this chapter is "*Inside the JVM*, by Bill Venners".

Definition of JVM

Virtual Machine
In general terms VM is a SW that creates an environment between the computer platform and end user in which end user can operate programs.

Original meaning for VM
As per its functionality is Creation of number of different identical execution environments on a single computer to execute programs is called VM.

Java Virtual Machine
It is also a VM that runs Java bytecode by creating five identical runtime areas to execute class members. This bytecode is generated by java compiler in a JVM understandable format.

Q) How can we start JVM process?
A) By using "*java*" tool.

The Java launcher, *java*, initiates the Java virtual machine instance.

Types of JVMs:
The Java 2 SDK, Standard Edition, contains two implementations of the Java virtual machine

- Java HotSpot Client VM
- Java HotSpot Server VM

Java HotSpot Client VM:
The Java HotSpot Client VM is the default virtual machine of the Java 2 SDK and Java 2 Runtime Environment. As its name implies, it is tuned for best performance when running applications in a client environment by reducing application start-up time and memory footprint.

Java HotSpot Server VM:
The Java HotSpot Server VM is designed for maximum program execution speed for applications running in a server environment. The Java HotSpot Server VM is invoked by using the *-server* command-line option when launching an application, as in

```
java -server MyApp
```

Some of the features Java HotSpot technology, common to both VM implementations, are the following.

Adaptive compiler

- Applications are launched using a standard interpreter, but the code is then analyzed as it runs to detect performance bottlenecks, or "hot spots".
- The Java HotSpot VMs compile those performance-critical portions of the code for a boost in performance, while avoiding unnecessary compilation of seldom-used code (most of the program).
- The Java HotSpot VMs also uses the adaptive compiler to decide, on the fly, how best to optimize compiled code with techniques such as in-lining.
- The runtime analysis performed by the compiler allows it to eliminate guesswork in determining which optimizations will yield the largest performance benefit.

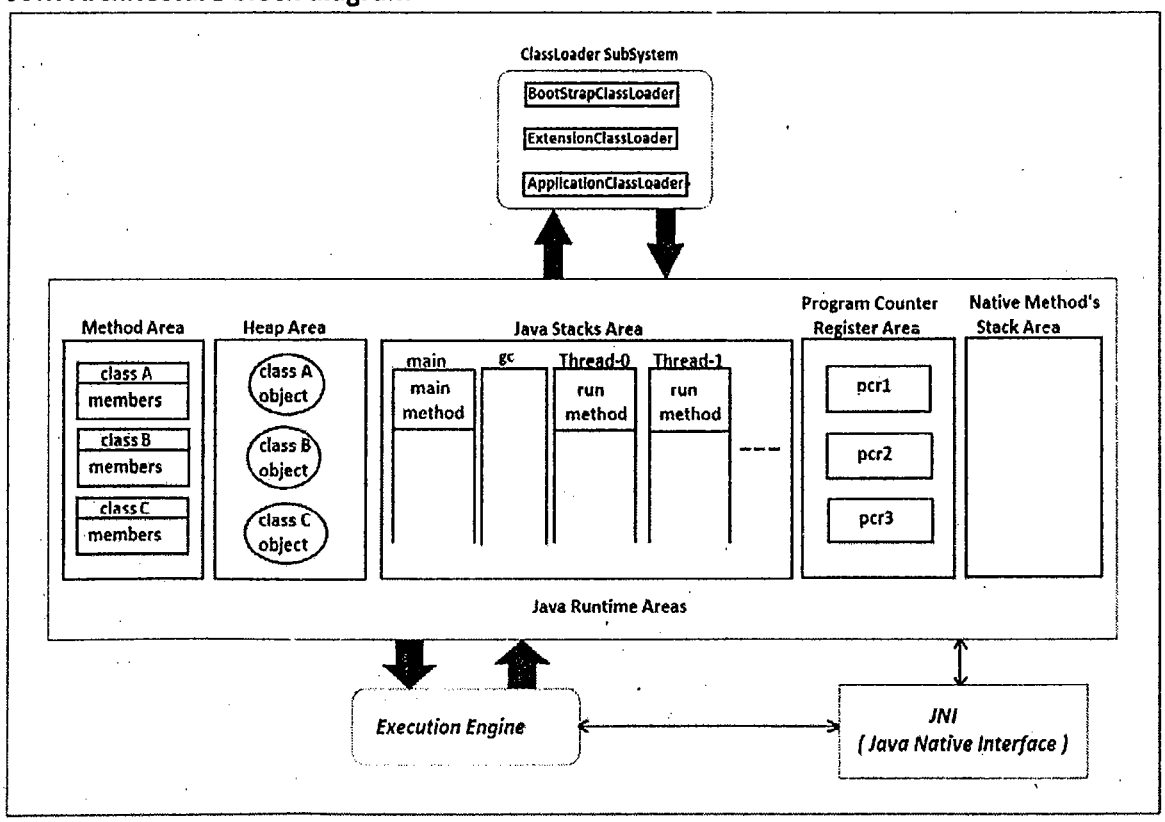
Rapid memory allocation and garbage collection:

- Java HotSpot technology provides for rapid memory allocation for objects, and it has a fast, efficient, state-of-the-art garbage collector.

Thread synchronization:

- The Java programming language allows for use of multiple, concurrent paths of program execution (called "threads").
- Java HotSpot technology provides a thread-handling capability that is designed to scale readily for use in large, shared-memory multiprocessor servers.

JVM Architecture block diagram



Explanation on Runtime Areas

Whenever we execute a class by specifying its corresponding class name by using the command "*java <ClassName>*", the Java launcher, **java**, immediately initiates the Java Runtime environment for the class execution as a layer on top of OS, and further the entire setup is divided in to 5 Java Runtime Areas named as

1. Method Area
2. Heap Area
3. Java Stacks Area
4. Program counter registers area
5. Native Methods Stacks area

Method Area

- All classes' bytecode is loaded and stored in this runtime area, and all static variables are created in this runtime area.

Heap Area

- It is the main memory of JVM. All *objects* of classes - *non-static variables memory*- are created in this runtime area. This runtime area memory is a finite memory.
- This area can be configured at the time of setting up of runtime environment using non standard option like

  java ***-xms <size>*** classname
- This area can be expandable by its own, depending on the objects creation.
- *Method area and Heap area both are sharable memory areas.*

Java Stacks area

- In this runtime area all Java methods are executed.
- In this runtime JVM by default creates two threads, they are
  - main thread
  - garbage collector thread
- Main thread is responsible to execute Java methods stats with main method, also responsible to create objects in heap area if it finds "new" keyword in any method logic
- Garbage collector thread is responsible to destroy all unused objects from heap area.

  Note: Like in C++, in Java, we do not have destructors to destroy objects.
- For each method execution JVM creates separate block in main thread. Technically this block is called Stack Frame. This stack frame is created when method is called and is destroyed after method execution.

  Note: Java operations are called "stack based operations (sequential)", because every method is executed only in stack.

Program Counter Registers Area

- In this runtime area, a separate program counter register is created for every thread for tracking that thread execution by storing its instruction address.

Native Methods Stacks Area

- In Native Methods stack area, all Java native methods are executed.

Q) What is a native method?
The Java method that has logic in C, C++ is called native method. To create native method, we must place native keyword in its prototype and it should not have body.

For example

```
class Example{
        public static native int add(int x, int y);

        public static void main(String[] args) {
                add(10, 20);
        }
}
```

The above program compiled fine, but in execution it leads RE: ***java.lang.UnsatisfiedLinkError.*** We have defined native method, but we have not defined it required C program and not linked.

Q) Suppose if we provide C, or C++ program for the above native method, who will take care of linking this java native method prototype to original native definition?
A) *JNI* - Java Native Interface - is a mediator between Java native method and original native method for linking its method calls.

Execution engine

- All executions happening in JVM are controlled by Execution Engine.

ClassLoader subsystem
ClassLoader is a class that is responsible to load classes into JVM's method area.

We have basically three types of class loaders

1. ApplicationClassLoader
2. ExtensionClassLoader
3. BootstrapClassLoader

- *ApplicationClassLoader* is responsible to load classes from Application Classpath, (current working directory). It basically uses Classpath environment variable to locate the class's ".class" file.
- *ExtensionClassLoader* is responsible to load classes from Extension Classpath, ("%JAVA_HOME%\jre\lib\ext" folder)
- *BootStrapClassLoader* is responsible to load classes from BootStrap Classpath (%JAVA_HOME%\jre\lib\rt.jar). These classes are predefined classes.

ClassLoader Working procedure:

- When JVM come across a type, it check for that class bytecodes in method area.
- If it is already loaded it makes use of that type.

- If it is not yet loaded, it requests class loader subsystem to load that class's bytecodes in method area form that class respective Classpath.

- Then ClassLoader subsystem, first handovers this request to *ApplicationClassLoader*, then application loader will search for that class in the folders configured in *Classpath* environment variable

- If class is not found, it forwards this request to *ExtenstionClassLoader*. Then it searches that class in *ExtensionClasspath*.

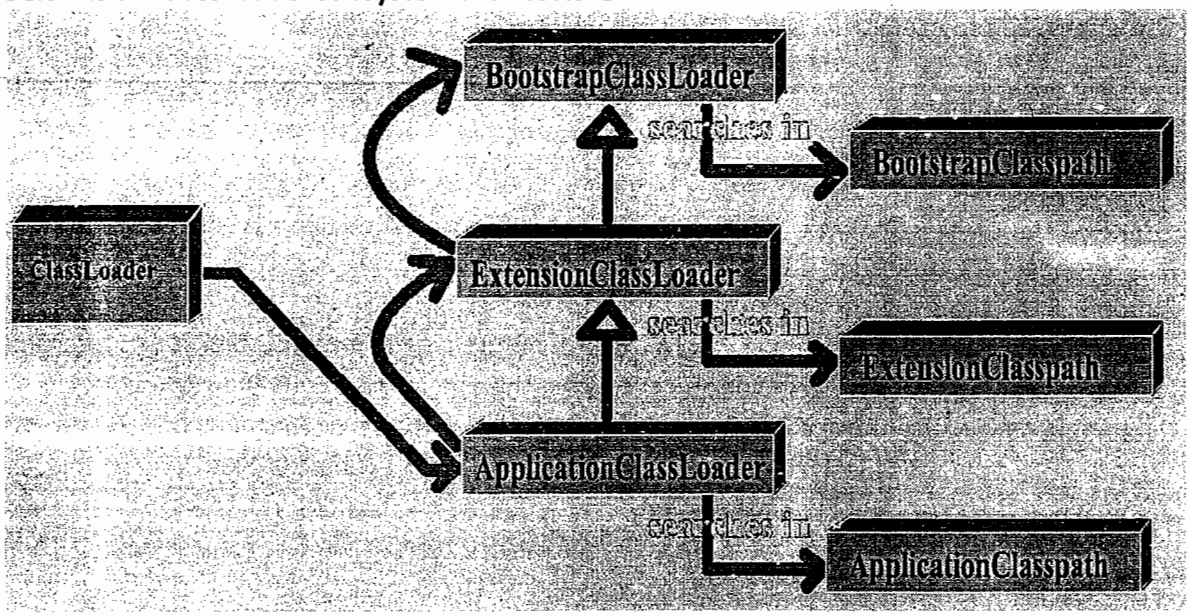
- If class is not found, it forwards this request to *BootStrapClassLoader*. Then it searches that class in *BootStrapClasspath*.

- If here also class not found, JVM throws an exception "*java.lang.NoClassDefFoundError*" or "*java.lang.ClassNotFoundExcetion*"

- If class is found in any one of the Classpaths, the respective ClassLoader loads that class into JVM's method area.

- Then JVM uses that loaded class bytecodes to complete the execution.

Below is the class loader subsystem architecture

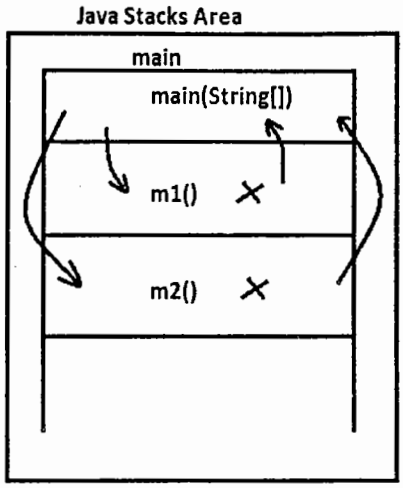


Thread & StackFrame Architecture

- thread is an independent sequential flow of execution created in JSA.
- StackFrame is a sub block created inside a thread for executing a method or block, and is destroyed automatically after the completion of that method execution.
- If this method has any local variables, they are all created inside that method's stack frame, and are destroyed automatically when StackFrame is destroyed.

Below architecture shows main thread with stack frames

```
class Example{

        static void m1(){
                System.out.println("m1");
        }

        static void m2(){
                System.out.println("m1");
        }

        static void m3(){
                System.out.println("m1");
        }

        public static void main(String[] args){
                System.out.println("main");
                m1();
                m2();
        }
}
```



StackFrame architecture

StackFrame is internally divided into three blocks to create that method's local variables, to store instructions, and to execute its logic.

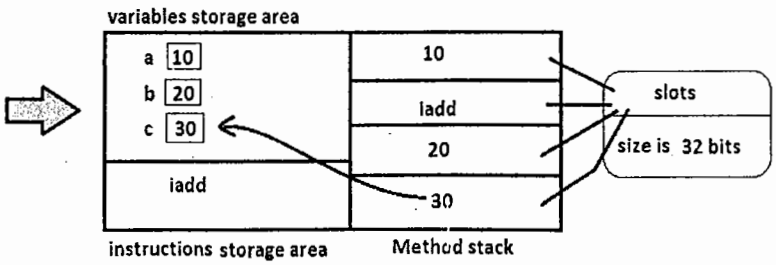
They are:

1. Variables storage area
2. Instructions storage area
3. Method stack

- All local variables are created in Variables storage area
- All method instructions are stored in Instructions storage area.
- Method logic is executed in method stack. To execute method logic, method stack is divided into number of blocks each block is called slot. This slot size is 32bits (4 bytes - int / float datatype depending of the variable type).
- Due to the slot size - byte, short, char datatypes are automatically promoted to int datatype, when they are used in an expression.
- So, the minimum result type coming out from an expression is *int*.

Below is the StackFrame architecture for add method

```
static void add(){

        int a = 10;
        int b = 20;
        int c = a + b;
        System.out.println("Result: "+c);

}
```



Draw JVM architecture for the below program

```
class Example{
        static int a = 10;
        static int b = 20;

                int x = 30;
                int y = 40;

        public static void main(String[] args){
                int p = 50;
                int q = 60;

                Example e = new Example();
        }
}
```

JVM Architecture final diagram with conclusions

- Complete class bytecodes are stored in method area with *java.lang.Class* object and all static variables get memory in method area
- All non-static variables i.e. objects, are created in Heap area when object is created.
- All Java methods, blocks and constructors are executed in Java stacks area in main thread by creating separate stack frame.
- So, all local variables are created in its method's stack frame.

What happened in JVM when we execute java command?

- When java command is executed, JVM is created as a layer on top of OS, and is divided in 5 runtime areas as shown above.
- For executing the requested classes, JVM internally performs below three phases.

They are:

1. Classloading:

JVM requests class loader to load the class from its respective Classpath. Then class is loaded in method area by using java.lang.Class object memory.

2. Bytecode verification phase:

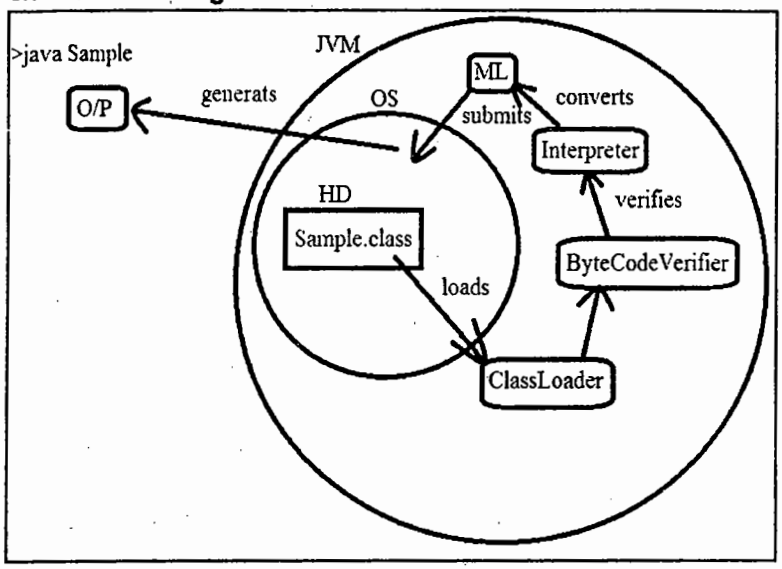
After Classloading, BytecodeVerifier verifies the loaded bytecode's internal format. If those bytecodes are not in the JVM understandable format, it terminates execution process by throwing exception "*java.lang.ClassFormatError*". If loaded bytecodes are valid, it allows interpreter to execute those bytecodes

3. Execution / interpretation phase:

Then interpreter converts bytecodes into current OS understandable format.

Finally, JVM generates output with the help of OS.

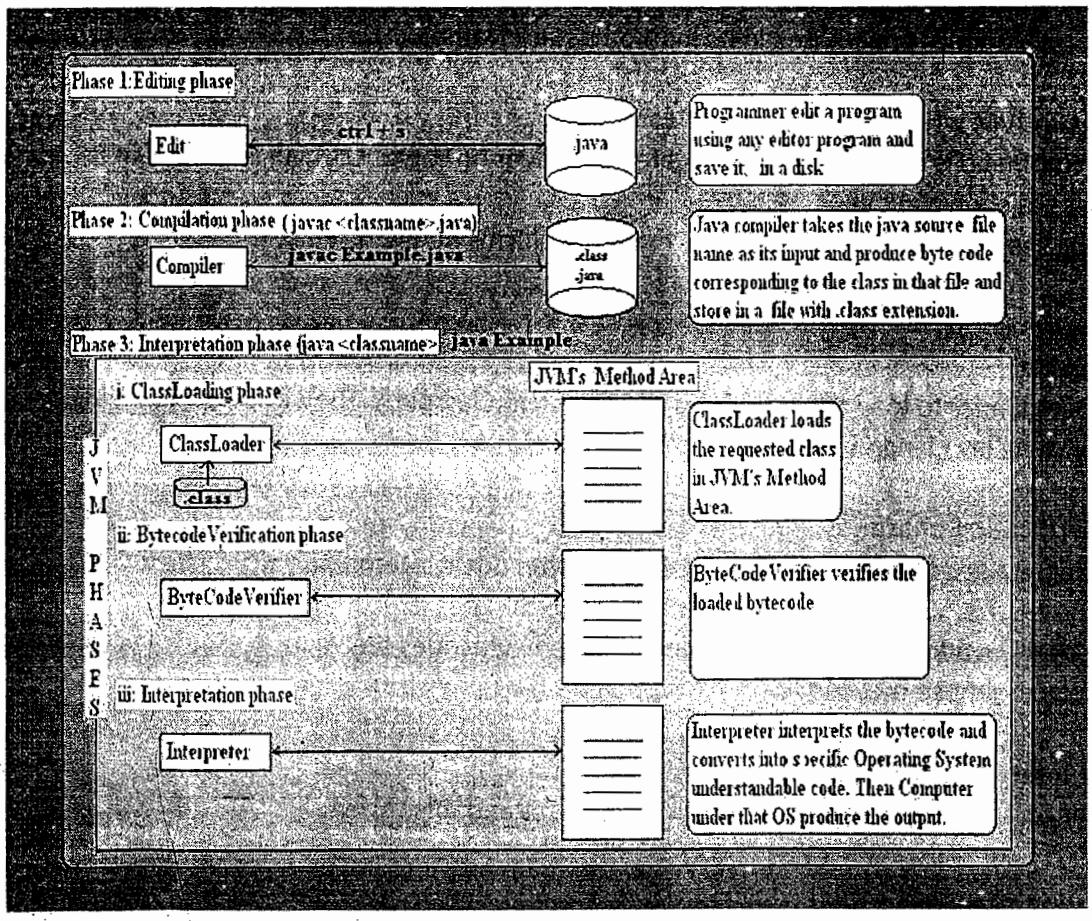
Check below diagram



How many phases are involved in developing, compiling and executing a java application?

We have 5 phases, they are

1. Editing phase
2. Compilation phase
3. Classloading
4. BytecodeVerification
5. Execution



Chapter 12

Static Members

& their execution *control flow*

- In this chapter, You will learn
  - Need of static variable, static block, static method
  - Storing, modifying and using data common for all objects
  - Executing logic commonly for all objects
  - Executing logic only once at the time of class loading
  - JVM activities at the time of class loading.
  - Static members execution flow

- By the end of this chapter- you will be in a position to tell right answer yourself without using Computer and Java Software.

Interview Questions

By the end of this chapter you answer all below interview questions

1. What members are called Static members?
2. Types of Static members?
    a. Static variables
    b. Static blocks
    c. Static methods
    d. Main method
3. When do all these members get memory location and by whom?
4. Static Variable
    a. When a variable can be called as Static variable?
    b. Does static variable executed by JVM by default?
    c. Can developer execute static variable?
    d. How many static variables can we define in a class?
    e. What is the order of execution of all static variables?
    f. When, where, how and by whom memory location is provided?
    g. What is the lifetime and scope of static variable?
    h. Can we declare local variables or parameters as static?
    i. JVM Architecture with static variables
    j. Duplicate Variables
    k. Can we create a local variable with parameter name?
    l. How Static variable can be differentiated from local variable when both have same name? Local preference/Shadowing.
    m. What is the order of execution of static variables and main method?
    n. How JVM execute the static variable before main method even if it is defined after main method?
    o. Identification and execution phases
5. Static method
    a. When a method is called as static method?
    b. Does JVM execute static methods automatically?
    c. Can developer execute static methods?
    d. When static methods are executed, what is the order of execution?
    e. Where static method's logic is stored and where it is executed?
    f. Variable initialization with its own name or parameter
    g. Initializing static variable with parameter
    h. If we modify static variable in one class is that modification affected to all classes of the project in side that JVM?
    i. Modularity and advantages of modularity

j. Order of execution of static variables, Static methods and main method

6. Static Blocks (SB)
    a. Definition of SB
    b. Does JVM execute SB automatically?
    c. Can developer execute SB?
    d. Where SBs logic is stored and where its logic is executed and when?
    e. How many static blocks can be defined in a class?
    f. Can we nest static blocks?
    g. Order of execution of all SBs?
    h. Order of execution of SB and MM?
    i. Class execution with SB and with/without main method?
    j. Order of execution of SV and SB (illegal forward reference)?
    k. Static control flow -> identification and execution phases
    l. Order of execution of SV, SB, SM and MM?

7. Main Method FAQs
    a. Why main() method is declared as public?
    b. Why main method has static keyword in its definition?
    c. When user defined methods should declare as static methods?
    d. Why JVM executes only main method, why it does not execute user defined static methods? Or Why main method is the initial point of class logic execution?
    e. Definition of main() method
    f. Why main() method return type is void?
    g. Why its name is main?
    h. Why main() parameter type is String[]?
    i. Why parameter name is args, can we change it?
    j. Can we invoke main() method?
    k. Can we declare local variables as static?

8. Executing static members of a class from another class

Static Members and their control flow

What members are called static members?
The Class level members which have static keyword in their definition are called static members.

Types of Static Members
Java supports four types of static members
1. Static Variables.
2. Static Blocks
3. Static Methods.
4. Main Method.

When do all these members get memory location and by whom?
All static members are identified and get memory location at the time of class loading by default by JVM in method area.

Static Variable
A class level variable which has static keyword in its creation statement is called static variable.

Does JVM execute static variables by default?
Yes, it executes static variables - means provides memory location- at the time of class loading.

How many static variables can we define in a class?
Multiple static variables we can define, but every variable name should be different

What is the order of execution of all static variables?
In the order they are defined from top to bottom.

When, where, how and by whom memory location is provided to static variable?
When class is loaded JVM provides *individual single copy* of memory location to each static variable in method area only once in a class life time.

What is the lifetime and scope of static variable?
Static variable get life as soon as class is loaded into JVM and is available till class is removed from JVM (or) JVM is shutdown. Its scope is class scope means it is accessible throughout the class directly by its name, and outside class by its class name provided it is non-private.
For example

```
class Example{
        static int a = 10;
        static void m1(){
                System.out.println( a );
        }
}
```

```
class Sample{

        static void m2(){
                System.out.println(Example.a);
        }
}
```

Can we declare local variables or parameters as static?

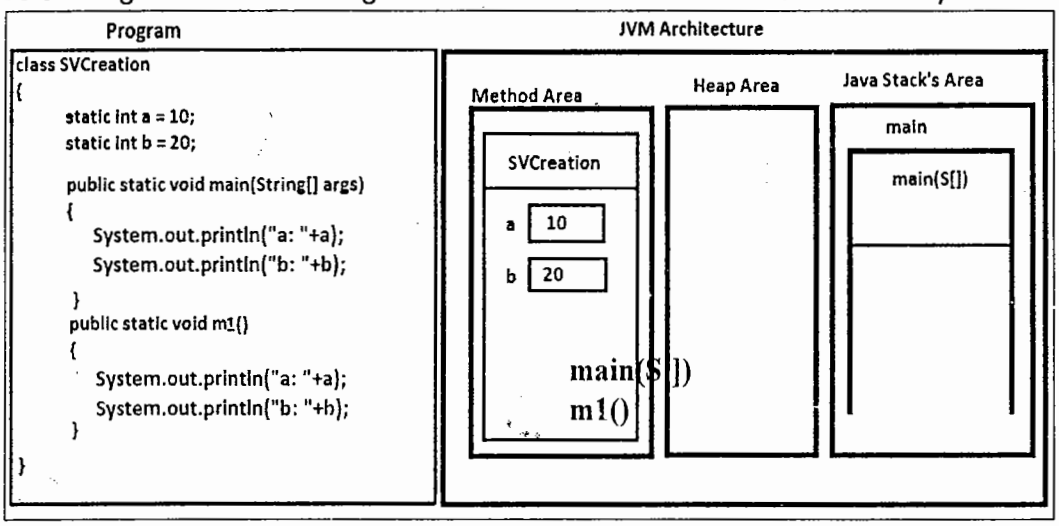
No, it is not possible. It leads to CE: *"illegal start of expression"*. Static keyword is not allowed inside a block "{ }", because as being a static variable it must get memory at the time of class loading which is not possible to provide memory to local variable at the time of class loading.

For Example:

```
class Example{
        static int a = 10;
        public static void main(String[] args){
                static int b = 20; ╳ CE: illegal start of expression
        }
}
```

JVM Architecture with Static variables

Below diagram shows creating a class with static variables and their memory structure.



Duplicate Variables

A variable that is created with existed variable name in the *same scope* is called duplicated variable. It leads to compile time error "**variable is already defined**". Even if we change data type or modifier or its assigned value we cannot create another variable with same name.

But it is possible to *create multiple variables* with *same name* in *different scopes*, for example the variable created in class scope can be created in method scope. *Check below program*

```
class Example
{
        static int a = 20;
        //int a = 10; //CE: a is already defined in Example

        public static void main(String[] args)
        {
                //it is allowed to define "a" variable in this method
                int a = 20;

                //creating local variable
                int p = 10;
                //double p = 30; //CE: a is already defined in Example
        }
        static void m1()
        {
                int p = 30;
        }
}
```

Q) Can we create local variable with parameter name?

A) No, because both parameter and local variables are created in the same method scope, so local variable is considered as duplicate variable

```
class Example{
        static void m1(int a){
                int a = 20; ╳ CE: a is already defined
                int b = 30; ✓
        }
}
```

Shadowing

Creating a local variable *or* parameter with same static or non-static variable name is called shadowing. It means local variable is a shadow of class level variable.

In this case when you access a variable in the method you will get local variable's value, but not from class level variable.

Local preference

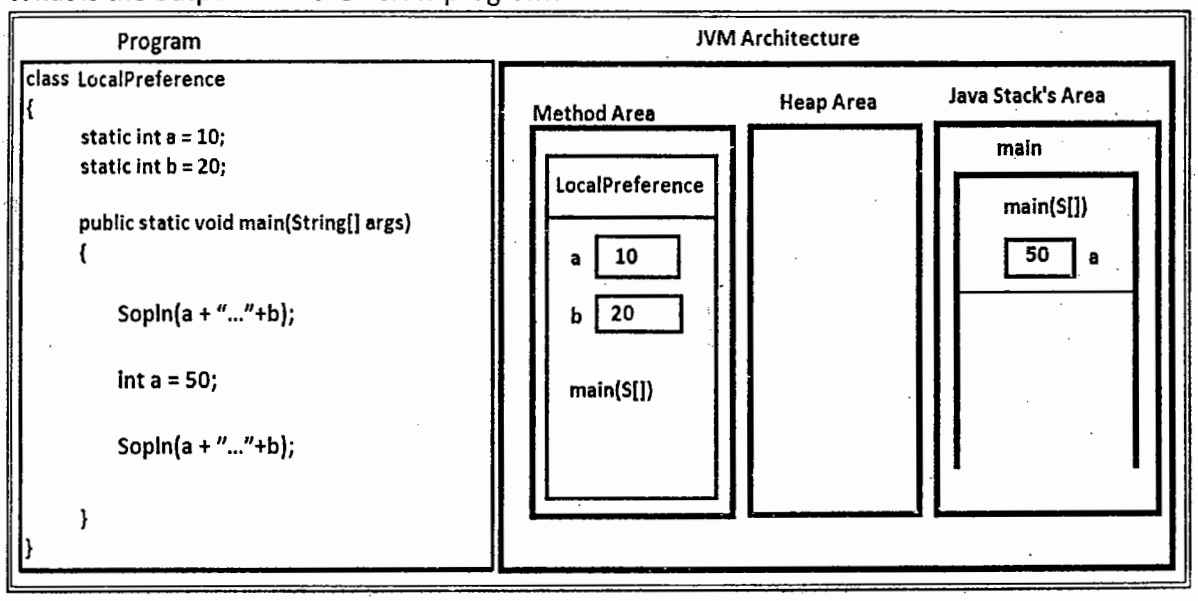
How compiler and JVM search for variable definition?

When we call a variable, compiler and JVM search for its creation statement in the current method. If it is not found in the current method, compiler and JVM next search for its creation statement at class level. If that variable creation statement is not found at class level also, compiler throws CE: **cannot find symbol**.

This phenomenon is called local preference.

Hence, if a variable is created in both method and also in class with same name, compiler and JVM access that variable from method, because always local variable has first priority.

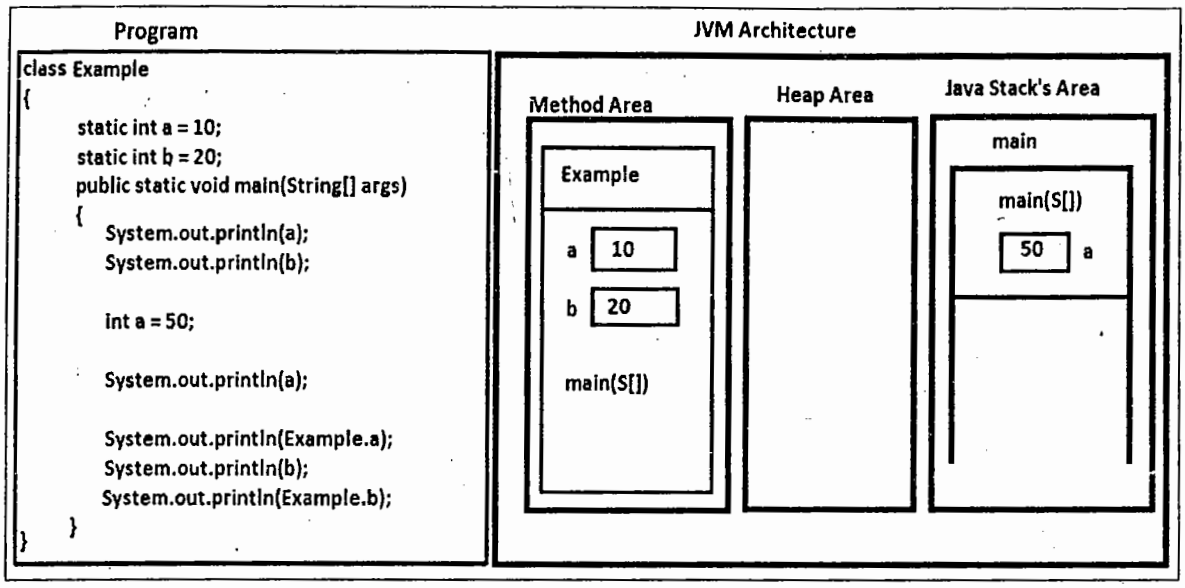
What is the output from the below program?



How static variable can be differenced from local variable or parameter when both have same name?

We should use ***class name*** to differentiate static variable from local variable.

As you observed in the above diagram static variable get memory location with respect to class, and local variables get memory location with respect to method. So to differentiate static variables from local variable we should use class name, as shown in the below diagram.



Note: We can access static variables with class name even though there is no local variable by static variable name. But it is optional.

Q) What is the order of execution of static variables and main method?

First all static variables are executed in the order they are defined from top to bottom then main method is executed. To prove this point assign static variables with a static non-void method and in that method print some debug message using *Sopln* as shown in the below

What is the output from below programs?

```
class Example {

        static int a = m1();
        static int m1(){
                Sopln("Variable 'a' is created");
                return 10;
        }

        static int b = m2();
        static int m2(){
                Sopln ("Variable 'b' is created");
                return 20;
        }

        public static void main(String[] args) {

                System.out.println("main");
                System.out.println("a: "+ a);
                System.out.println("b: "+ b);
        }
}
```

```
class Example {

        static int a = m1();
        static int m1(){
                Sopln("Variable 'a' is created");
                return 10;
        }

        static int m2(){
                Sopln ("Variable 'b' is created");
                return 20;
        }

        public static void main(String[] args) {

                System.out.println("main");
                System.out.println("a: "+ a);
                System.out.println("b: "+ b);
        }
        static int b = m2();
}
```

In the above program, how JVM can find variable *b* that is defined after main method?
Because of *Identification phase*, actually JVM does not execute class members directly. It executes class members in two phases

They are:

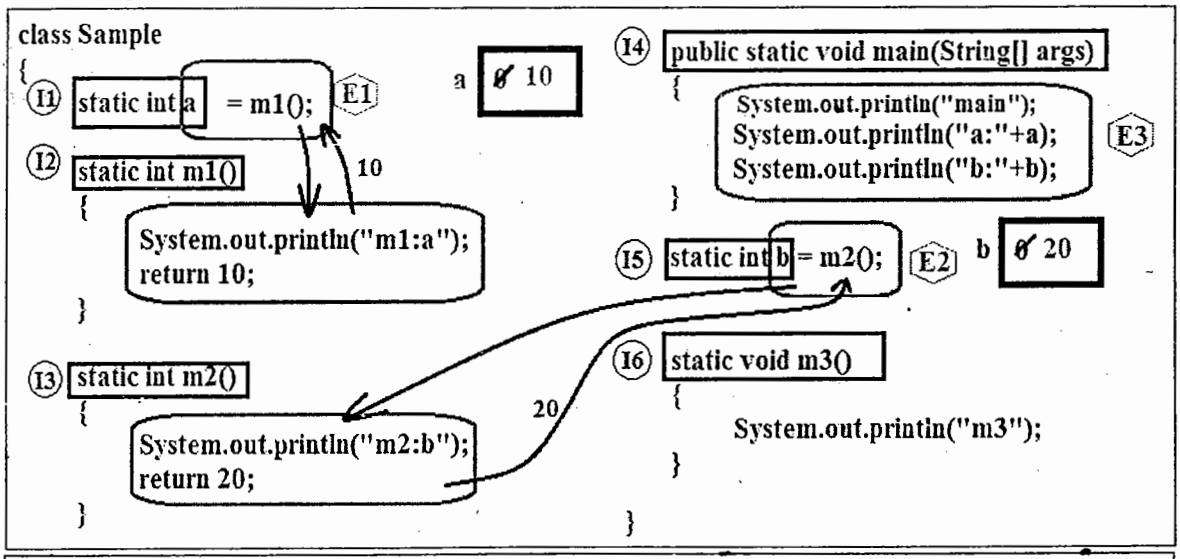
1. ***Identification phase***
2. ***Execution phase***

- First JVM identifies complete class static members at the time of class loading from top to bottom. After identification then it starts the static member's execution according to their priority from top to bottom.

- Identifying a variable means,
  - Creating its memory with default value based on its datatype
- Identifying a method means,
  - Remembering its prototype.

- Executing a variable means,
  - Storing its assigned value if any.
- Executing a method means,
  - Executing its logic if it is called.

Below diagram shows static variable and main method execution order




```
O/P:
====
m1:a
m2:b
main
a: 10
b: 20
```

Note: m3() method is identified but it is not executed as it is not called.

Static Method

A method which has static keyword in its definition is called static method.

Ex:

```
static void m1() {
  System.out.println("m1");
}
```

Does JVM execute static methods by default like static variables?

No, JVM does not execute static methods by itself. They are executed only if they are called explicitly by developer either *from main method*, or *from static variable* as its assignment statement or *from static block*.

What is the order of execution of static methods?

In the order they are called, not in the order they are defined.

Where static methods logic is stored and where they are executed?

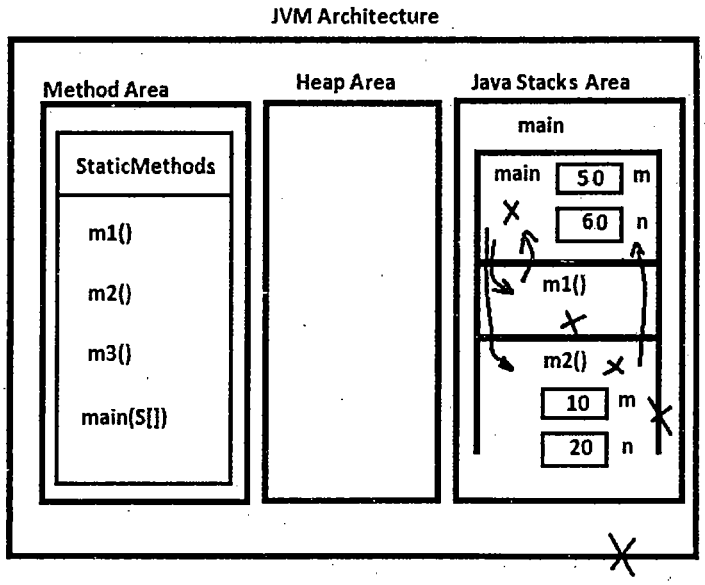
All static methods' logic is stored in method area and that logic executed in java stacks area in main thread by creating separate stack frame.

JVM architecture with Static method execution flow

When a static method is called from main method, JVM creates stack frame in main thread, loads that method complete logic and executes line by line of that method logic. The stack frame is destroyed immediately after method execution is completed.

Below diagram shows all above points

```
class StaticMethods{
        static void m1()    {
                System.out.println("In m1");
        }
        static void m2()    {
                System.out.println("In m2, start");
                int m = 10;
                int n = 20;
                System.out.println("m2 end");
        }
        static void m3()    {
                System.out.println("In m3, start");
                int m = 10;
                int n = 20;
                System.out.println("m2 end");
        }
        public static void main(String[] args) {
                System.out.println("main start");
                m1(); m2();
                int m = 50;  int n = 60;
                System.out.println("main end");
        }
}
```



Variable initialization with same variable

We can initialize a variable with same variable name, this assignment is valid. In this case the variable value is replaced with same value.

Ex:

```
int a = 10;
a = a ;
```

Assigning a static variable with a local variable or parameter

If a local variable or parameter is created with static variable name we must use class name in assigning static variable with that local variable or parameter. Else the modification is stored in that local variable or parameter not in static variable.

Check below programs

In the below program is static variable value is modified?

```
class Example
{
        static int a = 10;
        public static void main(String[] args)
        {
                int a = 20;

                a = a ;

                System.out.println(a);
                System.out.println(Example.a);
        }
}
```

To intialize static varaible with same name local variable we must refer left hand side variable with class name explicitly as shown below

```
class Example
{
        static int a = 10;
        public static void main(String[] args)
        {
                int a = 50;

                Example.a = a;

                System.out.println(a);
                System.out.println(Example.a);
        }
}
```

Local preference with parameters

Since parameters are also treated as local variables, we must refer static variable with class name if parameter is declared with same static variable name.

Find out, is static variable modified in below programs

```
class Example{
        static int a;

        static void m1(int x){
                a = x;
                System.out.println(a);
        }

        public static void main(String[] args){
                System.out.println(a);
                m1(50);
                System.out.println(a);
        }
}
```

```
class Example{
        static int a;

        static void m1(int a){
                a = a;
                System.out.println(a);
        }

        public static void main(String[] args){
                System.out.println(a);
                m1(50);
                System.out.println(a);
        }
}
```

```
class Example{
        static int a;
        static void m1(int a){
                Example.a = a;
                System.out.println(a);
        }
        public static void main(String[] args){
                System.out.println(a);
                m1(50);
                System.out.println(a);
        }
}
```

```
class Example{
        static int a;
        static void m1(int a){
                a = Example.a;
                System.out.println(a);
        }
        public static void main(String[] args){
                System.out.println(a);
                m1(50);
                System.out.println(a);
        }
}
```

What is the output from the below program?

```
class Example
{
        static int a;
        static int b;

        static void m1(){
                a = 10;
                b = 20;
        }

        static void m2(int x , int y){
                a = x;
                b = y;
        }

        static void m3(int a , int b){
                a = a;
                b = b;
        }

        static void m4(int a , int b){
                Example.a = a;
                Example.b = b;
        }

        public static void main(String[] args)
        {
                System.out.println(a+"..."+b);

                System.out.println();

                m1();
                System.out.println(a+"..."+b);

                System.out.println();

                m2(30, 40);
                System.out.println(a+"..."+b);

                System.out.println();

                m3(50, 60);
                System.out.println(a+"..."+b);

                System.out.println();

                m4(70, 80);
                System.out.println(a+"..."+b);

        }
}
```

Modularity

Modularity means dividing big task into small pieces. In Java we can achieve modularity with methods and also with classes.

In the below program we have written all arithmetic operations in single method, so it is not possible to execute exactly one of the arithmetic operations, all 4 operations will be executed, hence it is wrong design.

Dividing four tasks into four methods to execute each task individually is called modularity.

No Modularity

```
class AO
{
        static void ao(int a , int b)
        {
                //addition
                System.out.println(a + b);

                //substraction
                System.out.println(a - b);

                //multiplication
                System.out.println(a * b);

                //division
                System.out.println(a / b);
        }
}
```

With Modularity

```
class AO
{
    static void add(int a , int b)
    {
            //addition
            System.out.println(a + b);
    }
    static void sub(int a , int b)
    {
            //substraction
            System.out.println(a - b);
    }
    static void mul(int a , int b)
    {
            //multiplication
            System.out.println(a * b);
    }
    static void div(int a , int b)
    {
            //division
            System.out.println(a / b);
    }
}
```

Advantages in modularity

We can achieve *centralized code* change and *code reusability*.

For example if we have some logic to be executed in multiple methods, placing that logic in all methods is *not feasible* because for every small change in the logic we must do this modification in all places. This considered as *code redundancy* and there is no centralized code change.

So to solve above two problems that logic should not be placed in every method rather define this logic in a separate method and call that method from every method where ever we need it. So that if we need to do any changes further those changes we required to do this change only in the common method as shown below.

```
class Example
{
        static void m1()
        {
                ---------------        add();
                Sopln(10 + 20); => Sopln("The result after adding  10  and 20 is "+ 30 );
                ---------------
        }
        static void m2()
        {
                ---------------        add();
                Sopln(10 + 20); => Sopln("The result after adding  10  and 20 is "+ 30 );
                ---------------
        }
        static void m3()
        {
                ---------------        add();
                Sopln(10 + 20); => Sopln("The result after adding  10  and 20 is "+ 30 );
                ---------------
        }
        static void add()
        {
                Sopln(a + b);
                Sopln("Result: "+(a + b))
        }
}
```

Q) What is the design change we should do in the above program to use add method from multiple classes?

We should define the above add method in another separate class. Then we should access / reuse this method with that class name or with that class object. This approach is called developing *modularity at class level*.

Below program shows above said code change.

This Addition class is called ***component***, a *reusable class*.

```
//Addition.java
public class Addition {
        public static void add(int a, int b){
                System.out.println("The addition of "+ a + " and "+ b + " is: "+ (a + b) );
        }
}
```

Below classes are reusing this addition logic

```
//A.java
class A {
        public static void main(String[] args) {
                Addition.add(10, 20);
        }
}
```

```
//B.java
class B {
        public static void main(String[] args) {
                Addition.add(50, 60);
        }
}
```

Static Blocks

Static block is a class level nameless block that contains only static keyword in its prototype.

Syntax:

```
class Example
{
        static
        {
                -------------
                -------------
                -------------
                -------------
        }
}
```

Any Java Legal statement is allowed except ***return*** and ***throw*** statement

Need of static block

It is used to execute logic only at the time of class loading.

Logic like,

- Initializing static variables
- Registering native libraries
- To know classes loading order, etc...

Different ways to execute logic at the time of class loading

Actually there are two ways to execute logic at the time of class loading.

1. **Using static variable.**

```
class Example{
        static int a = m1();

        static int m1(){
                System.out.println("SV : a");

                return 10;
        }
```

```
        public static void main(String[] args)
        {
                System.out.println("main");
                m1();
        }
}
```

2 Drawbacks in this approach

a. m1() method logic can also be executed after class loading, because it can be called from main() method, that leads to execution of m1( ) after class loading.
b. the method must be a non-void method.

2. **Using static block**

It is the Solution for the above problem - write that logic in static block to execute that logic only at the time of class loading

Who will execute static block when and where?
Static blocks are executed automatically by JVM at the time of class loading in the order they defined from top to bottom by creating separate stack frame in main thread in Java stacks area.

Then what is order of execution of SB and Main method?
Static block is always executed before main method.

What is the output from the below program?

```
class Example
{
        static
        {
                System.out.println("SB");

        }
        public static void main(String[] args)
        {
                System.out.println("main");
        }
}
```

```
class Example
{
        public static void main(String[] args)
        {
                System.out.println("main");
        }
        static
        {
                System.out.println("SB");

        }
}
```

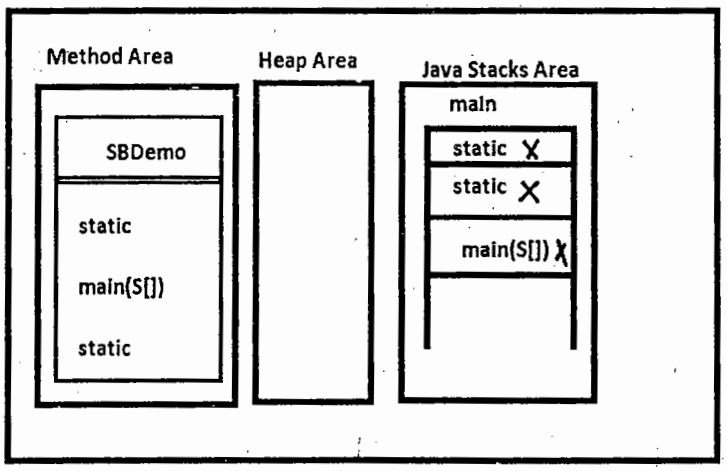
How many static blocks can be defined in a class?
We can define *more than one static block* in a class.

Can we nest static blocks *means* Can we write a static block in another static block?
No, static keyword is not allowed inside blocks or methods

What is the order of execution of all static blocks?
All static blocks are executed in the order they defined from top to bottom.

Check below program, what is the output?

```
class SBDemo
{
        static
        {
                System.out.println("SB2");
        }

        public static void main(String[] args)
        {
                System.out.println("main");
        }

        static
        {
                System.out.println("SB1");
        }
}
```



Below diagram shows Identification and execution phases of SB and main

```
class MultipleStaticBlocks
{
    static                                  (E1)
    {
        System.out.println("SB2");
    }
    public static void main(String[] args)  (E3)
    {
        System.out.println("main");
    }
    static                                  (E2)
    {
        System.out.println("SB1");
    }
}
```

```
O/P
===
SB2
SB1
main
```

Can we execute class logic without main method?

Yes, we can execute by using either static variables or static blocks, but from Java 7 main method is mandatory to execute a class.

Class execution procedure with static block and main method?

When a class is loaded, first JVM executes static variables and static block then after it searches from main method, if main method presents it execute main method else it terminates program execution with exception *java.lang.NoSuchMethodError*.

But from Java 7 onwards class execution starts only if it has main method, else program execution is terminated without executing static variables or static blocks by throwing an ***Error: main method is not available***.

What is the output from the below program?

```
class Example{
}
```

>java Example

```
class Example{
        static {
                System.out.println("SB");
        }
}
```

>java Example

```
class Example{
        static int a = m1();

        static int m1(){
                System.out.println("SV : a");
                return 10;
        }
}
```

>java Example

```
class Example{
        static int a = m1();

        static int m1(){
                System.out.println("SV : a");
                return 10;
        }

        static {
                System.out.println("SB");
        }
}
```

>java Example

```
class Example{
        static {
                System.out.println("SB");
        }

        static int a = m1();

        static int m1(){
                System.out.println("SV : a");
                return 10;
        }
        public static void main(String[] args){
                System.out.println("main");
        }
}
```

>java Example

Q) Can we execute main method at the time of class loading?

Yessssssssss, it is possible.

Call it from static block it is also executed at the time of class loading.

What is the output from the below program?

```
class Example{
        static{
                System.out.println("SB start");
                main(new String[0]);
                System.out.println("SB end");
                System.out.println();
        }
        public static void main(String[] args){
                System.out.println("main");
        }
};
```

O/P

```
SB start
main
SB end

main
```

Main method

Why main method is public?
Because it must be called by JVM from outside of our package.

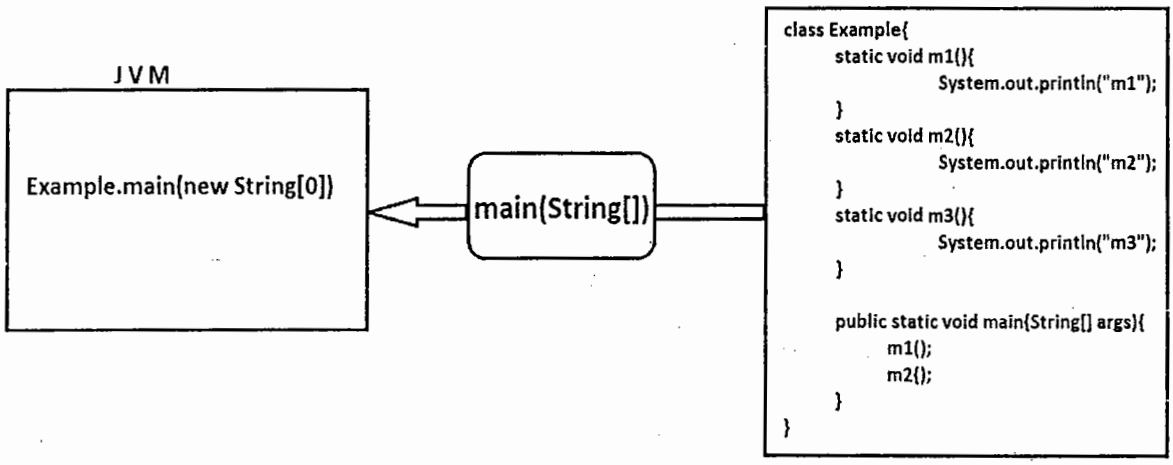
Why main method has static keyword in its definition?
main() is the initial point of class logic execution. Hence it should be identified at the time of class loading. Due to this reason it contains static keyword in its prototype. Of course it must be executed without object create.

When user defined methods should contain static keyword?
To execute the method logic without object creation, method should be defined as static method. It means if method, logic internally not using object data – non-static variables – then it is recommended to declare that method as static method.

Why JVM executes only main method why not user defined static methods? Or why main method is the initial point of class logic execution?
Main method calling statement is available in JVM software, but our static methods calling statement is not available in JVM software.

And more over JVM doesn't know your methods execution order and also it doesn't know what are the methods should be executed from your class and when.

It is the developer responsibility to inform methods and their order of execution to JVM. For this purpose there should be a common method that is known to both JVM and to developer, and that method prototype should be given by the SUN. That method is main method.



Definition of main method
main method is the mediator method between java developer and JVM to inform methods execution order.

Why main() method return type is void?
Because if we return a value, it is sent to JVM which is useless. Hence main() method return type is void.

Why its name is main?
As per the coding standards the method name should convey the operations it is doing. Since this method is intended to execute main logic of the program, that is business logic, it is named as main.

Why main() method has parameter String[]?
To read command line arguments (values) into Java application from keyboard

For example

```
>java sample 10 20
```

Why main method parameter type is String?
Because the value passing from keyboard is sent into Java application as String type value.

Why is it String[], why not it is just String?
To read more than one value

Why main() method parameter name is args?
Again it is also because of coding standard, as we are reading arguments from keyboard the name of the parameter is "args". We can change its name, but it is not recommended.

Can we call main method explicitly?
Yes it is possible, it is also a method.
Note: main( ) method should not be called from its own block or from a method that is calling from main(), it leads to exception java.lang.StackOverflowError.

Conclusions
- Using static keyword we can provide memory location directly for variables and methods.
- If any variable or method is defined as static then we can access them using class name from outside class.
- In same class if the local variable and the static variable has the same name we must differ them using class name.
- In method block static variable can't be defined, It should be proved in class scope only.
- Using static block a class can be executed without main() method till Java 6.
- All static blocks will be executed first in their textual order.
- After execution of all static blocks JVM search for main method, if it is provided then it will be executed otherwise the runtime error (Exception) will be thrown along with the static blocks output.
- We write multiple static blocks but we can't nest them.
- These blocks will be executed only once in the life time of a class

Chapter 13

NON-Static Members

& their execution *control flow*

- In this chapter, You will learn
  - Need of non-static variable, non-static block, non-static method, and constructor
  - Storing, modifying data specific to one object
  - Initializing object common for all constructors
  - Initializing object with specific values
  - Executing logic specific to one object after its creation
  - JVM activities at the time of object creation.
  - Non-Static members execution control flow

- By the end of this chapter- you will be in a position to tell right answer yourself without using computer and Java Software.

Interview Questions

By the end of this chapter you answer all below interview questions

1. What are the members called as Non-Static members?
2. Types of non-Static members?
    a. Non-Static variables
    b. Non-Static blocks
    c. Non-Static methods
    d. Constructors
3. When do all these members get memory location and by whom?
4. What is an object, and object creation syntax?
5. Non-Static Variable
    a. When a variable can be called as Non-Static variable?
    b. Does JVM execute non-static variables automatically?
    c. When and by whom memory location is provided and where?
    d. JVM Architecture to show object structure
    e. How many objects can be created from a class?
    f. If we modify non-static variable using one object is that modification effected to another object?
    g. What are the possible values we can assign to a referenced variable?
    h. Types of referenced variables based on its value.
    i. *java.lang.NullPointerException,* how can we solve NPE?
    j. Can we print object? – introduction to toString() method
    k. Difference in static and non-static variables memory location, and their modifications?
6. Non-Static method
    a. When can we call a method is a non-static method?
    b. Does JVM execute non-static methods automatically?
    c. How non-static methods are executed and what is the order of execution?
    d. Where are they stored and executed?
    e. Can we create an object without having non-static variable in the class?
    f. Difference in memory location structure of non-static variables and non-static methods across multiple objects.
    g. How can we access non-static members from other non-static members?
    h. How Non-static variable is differentiated from multiple objects in a non-static method as it has separate copy of memory in every object?
    i. Use of *this* keyword, and its definition?
    j. How can we write code common for all objects either for setting/getting/printing object values and it should be executed with the specific values of an instance of that object?

k. Java bean design pattern?
l. What is POJO and POJI?
m. What is the difference between POJO and Java bean?
n. If we modify object in a NSM using *this* keyword is that modification effected to the original referenced variable?
o. How many referenced variables can point to a single object, and what is the effect in modifying that object using any one of those referenced varaibles?
p. Accessing NSV from local object referenced variable.
q. Can we pass object as an argument to a method?
r. In how many ways we can pass objects into a method?
s. Invoking methods by passing value and reference
t. Determine the effect upon object references and primitive values when they are passed into methods that perform assignments or other modifying operations on the parameters.
u. How Non-Static variables can be differentiated from local variables / parameters when both have same name? Local preference – shadowing
v. this keyword rules

7. Constructor
    a. Definition and need of constructor
    b. Rules in defining Constructor
    c. Rules in invoking constructor
    d. Why return type is not allowed for constructor?
    e. Can we define a method with same class name?
    f. How compiler and JVM differentiate method and constructor blocks in definition and calling if both have same name?
    g. Why constructor name should be same as its class name?
    h. Can we declare constructor as private?
    i. When constructor should be declared as private?
    j. What is singleton design pattern class?
    k. In how many ways can we develop this class
    l. Types of Constructors
        i. Default constructor
        ii. No-argument | non-parameterized | ZERO arg constructor
        iii. Parameterized constructor
    m. Why compiler given constructor is called default constructor?
    n. Why compiler defines default constructor without logic and parameters?
    o. When compiler provides constructor?
    p. When should we define constructor explicitly?
    q. When should we define parameterized constructor in a class?
    r. What are the differences between default and no-arg constructor?

s. What is the difference in creating multiple objects with above three constructors?
t. How many constructors can we define in a class?
u. Constructor overloading
v. If we invoke one constructor, will all other constructors are executed?
w. How can we execute logic common for all constructors?

8. Non-Static Blocks (NSB) / Instance Initializer Block (IIB)
    a. Definition of NSB
    b. Need of non-static block
    c. Who will execute NSB, when and where?
    d. How many non-static blocks can be defined in a class?
    e. Order of execution of all NSBs?
    f. Can we nest NSBs?
    g. Order of execution of NSB and Constructor?
    h. Non-static control flow - Order of execution of NSV, NSB, NSM and C?
    i. Order of execution of NSV and NSB (illegal forward reference)?
    j. Class execution with only NSB with and without main method?
    k. Can we start class logic execution with only non-static members?

9. Object creation process
    a. Job of "new" keyword
    b. Job of constructor

10. What is the difference between instance and object?
11. Why non-static variables are called as instance variables, and non-static methods are called as instance methods?

12. Non-static control flow – <u>Identification and execution</u>
13. Recursive method and constructor call
14. Executing non-static members from static and non-static members of same and other class members.
15. In how many ways we can load class into JVM?
16. How can you write a class, to create its object itself when it is loaded into JVM?
17. How can we pass a class's current object to another class?

Non-Static Members and their Control flow

What members are called non-static members?
Class level members which don't have static keyword in their creation statement are called non-static members.

Types of non-static members
Java supports four types of non-static members.

1. Non-static variables.
2. Non-static blocks
3. Non-static methods
4. Constructors

When do all these members get memory location and by whom?
All above members gets memory location only if object is created with new keyword and constructor of that class. JVM will not provide memory location for these members by default by itself.

What is an object and its creation syntax?
Technically speaking, object is an instance of a class that contains continuous memory location of all non-static variables of a class with specific data of this instance. Object can be created by using new keyword and constructor of that class.

The **syntax** to create an object

```
<Access Specifier> <modifier> <class name> <variable name> = new <class name>();
```

For instance, Example class object is crated as shown below

```
Example e   =      new Example();
```

Here e **is** object name **and** new Example() **is** object creation statement.

Non-static variable
The class level variable that does not have static keyword in its definition is called non-static variable.

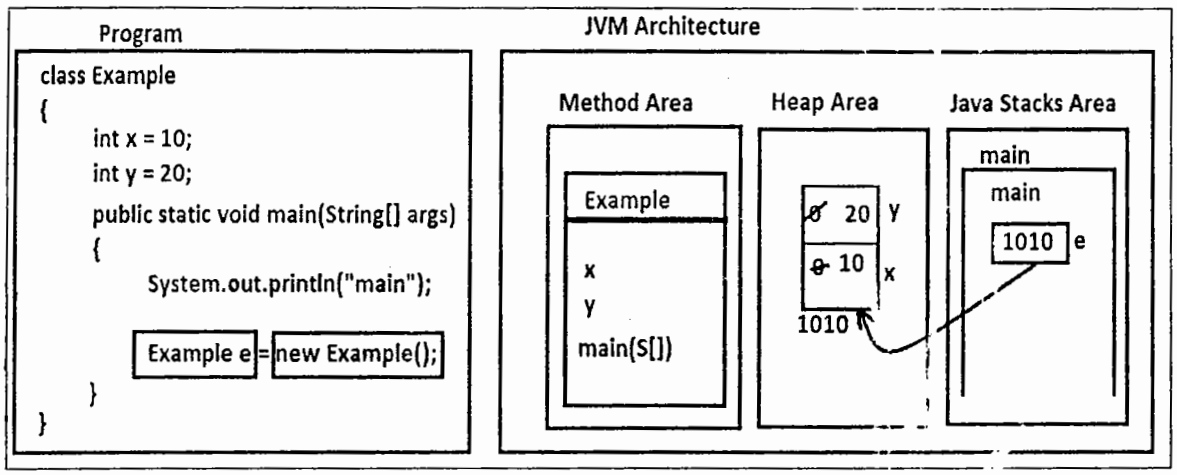
Ex:

```
class Example{
        int x =10;   } Non-static variables
        int y = 20;  }
}
```

Q) Will JVM executes non-static variables automatically?
No, JVM executes non-static variables only if object is created.

Q) When, where, how and by whom non-static variables get memory location?
Non-static variables get memory location in heap area in continuous memory locations by JVM when object is created.

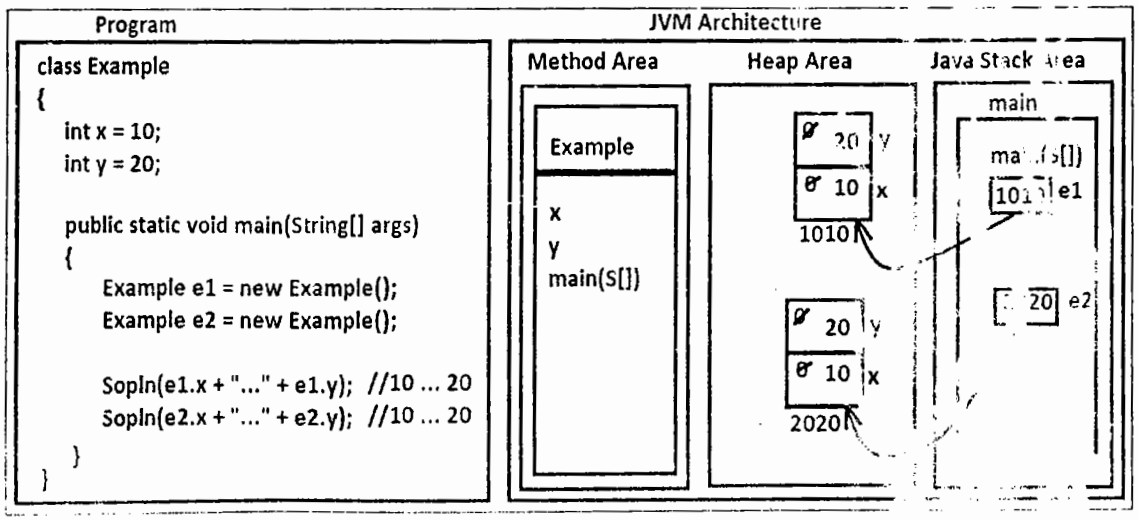
Below diagram shows defining a class with non-static members, creating object, and its JVM architecture.



I am assuming ***1010*** is the **object *reference.*** It is also called object's *base address*.

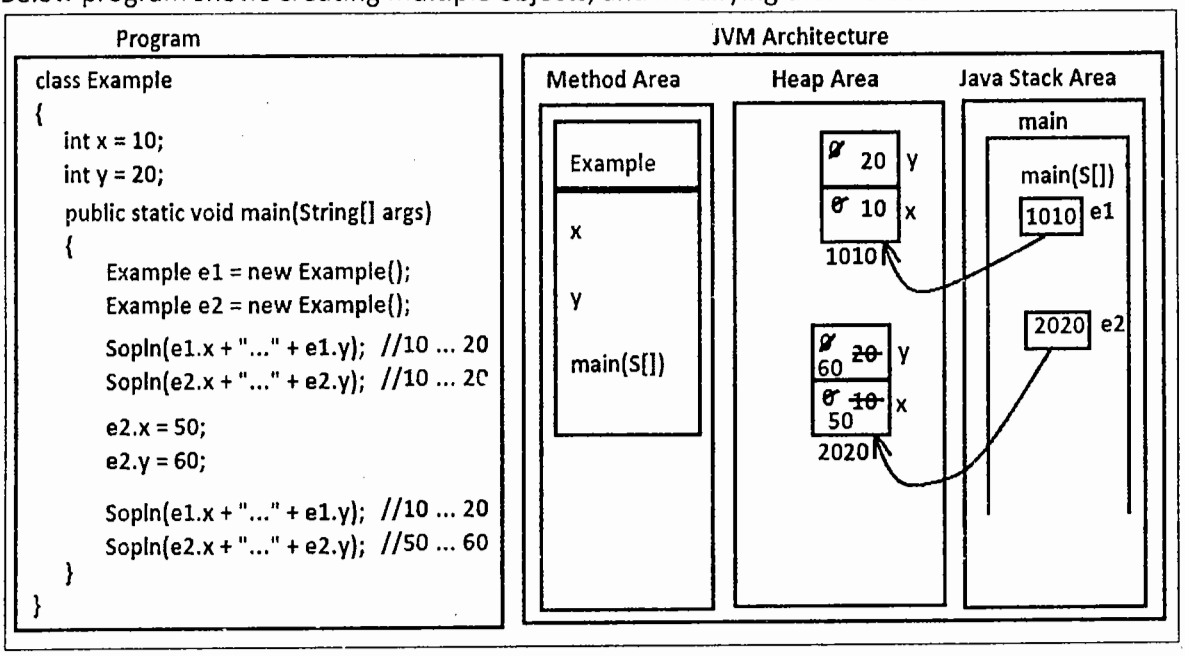
After object creation and initialization its reference / base address is stored in destination referenced variable. In the above program ***e*** is the destination referenced variable.
Note: object reference is retuned by ***new*** keyword not by constructor.

Q) How many objects can be created for a class?
A) *Multiple Objects*. We can create multiple objects for a class, but their referenced variable name must be different. When we create multiple objects, non-static variables get separate copy of memory for each object. So to access non-static variables from a particular object we must use that object's referenced variable. Below diagram shows creating multiple objects and accessing non-static variables from each object and its JVM architecture.



Q) If we modify one object data will another object data also be modified?
No, modifications done for one object will not be affected to another object, because we change object data using its referenced variable.

Below program shows creating multiple objects, and modifying their values



As you can noticed, e1 object values are not modified even though we modify e2 object values.

Types of referenced variables based on values it has
We have two types of referenced variables based the value we have stored

1. *null* referenced variables
2. *object* referenced variable

Definition
If we store ***null*** in referenced variable, it is called null referenced variable. It will not point to any *non-static variables* or *non-static methods* of the class.

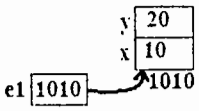
For example

Example e1 = null;

If we store ***object reference*** in referenced variable, it is called object referenced variable. It points to all non-static variables or methods of the class.

For example

Example e1 = new Example();



What is the output printed if we print referenced variable / object?
If we print null referenced variable, print() or println() methods prints => ***null***
If we print object referenced variable, print() or println() methods prints =>
classname@hashcode in hexadecimal string form at

Execute below program in your system for more clarity

```
class Employee{

        int     eid         = 7279;
        String  ename       = "HariKrishna";
        String  company     = "Naresh i Technologies";

        public static void main(String[] args){

                Employee e1 = new Employee();
                System.out.println(e1);
        }
}
```

Output on console

```
Select C:\Windows\system32\cmd.exe

D:\NareshIT\HariKrishna\CoreJava&OCJP\10NSM>javac Employee.java

D:\NareshIT\HariKrishna\CoreJava&OCJP\10NSM>java Employee
Employee@19821f

D:\NareshIT\HariKrishna\CoreJava&OCJP\10NSM>
```

What is that *19821f?*
19821f is the ***e1*** object's hash code value in hexa string format.

But when you print object, how it's hash code is printed in its hexa string format?
It is the internal implementation of *toString()* method.

A small introduction to *toString()* method
It is a predefined method available in *java.lang.Object* class to return object information in string format. Its default implementation is returning object's [*classname@hashcode in hexadecimal string format].*

Answer below question
Q) What is the method called internally by print() and println() methods to print object?
A) *toString()*

When you print object, print() and println() methods internally calls toString() method to print object's information.

Q) *Is the above printed information is meaningful?* => **No**
Q) *How can we print object's state when we print object?* => **override** *toString()* **method in Employee class** *as shown below*

Below program shows overriding toString() method to print object data

```
class Employee{

        int     eid;
        String  ename;
        String  company;

        //overriding toString() method to print object data
        public String toString(){

                return  "eid: "+ eid +"\n" +
                        "ename: "+ ename +"\n" +
                        "company: "+ company +"\n";
        }

        public static void main(String[] args){

                Employee e1 = new Employee();

                e1.eid          = 7279;
                e1.ename        = "HariKrishna";
                e1.company      = "Naresh i Technologies";

                System.out.println(e1);
        }
}
```

Note:
To know more details on toString() method and its implementation check "Chapter 21: *Fundamental Classes*"

Now execute above program, ***object's state* is printed** instead of *classname@hashcode*.

```
Select C:\Windows\system32\cmd.exe

D:\NareshIT\HariKrishna\CoreJava&OCJP\10NSM>javac Employee.java

D:\NareshIT\HariKrishna\CoreJava&OCJP\10NSM>java Employee
eid: 7279
ename: HariKrishna
company: Naresh i Technologies

D:\NareshIT\HariKrishna\CoreJava&OCJP\10NSM>_
```

Q) What is the output printed if we print null referenced variable? => null

For Example:

```
Employee e = null;
System.out.println(e); // null
```

To *print null referenced variable*, print and println() methods **does *not call toString()*** method, because it leads to *java.lang.NullPointerException*.

Q) Can we access static variables or static methods using referenced variable?
Yes, we can access, because they get memory directly at the time of class loading and more over they are part of object.

Q) Can we access non-static variable or non-static method with *null* referenced variable?
No, it leads to RE: *java.lang.NullPointerException* because non-static variable or non-static method does not have memory with respect to null referenced variable.

For Example

```
Employee e1 = null;
System.out.println(e1.eid);    RE: java.lang.NullPointerException
```

In this case we do not get compile time error, because compiler does *not check variable value*.

Q) Can we access static variable or static method with *null* referenced variable?
Yes, there is NO CE, and NO RE, program is executed successfully, because static variable or static method gets memory directly with respect to class.

The thumb rule we should follow in accessing class members is
"Where ever we use *class name* to access class members there we can use *referenced variable*, but vice versa is not possible. It means, wherever we use referenced variable to access class members there we cannot use class name". In simple terms, we can access static members with referenced variable, but we cannot access non-static members using class name, it leads CE: "*non-static member cannot be referenced from static context*".

What are the different ways to access static and non-static variables, and what is the effect in modifying static and non-static variables with an object?

<u>Accessing static and non-static variables</u>
<u>*Static variables*</u> can be accessed using

1. it's name directly
2. class name
3. null referenced variable
4. Object referenced variable name - means object

But <u>*non-static variables*</u> can only be accessed using

1. object referenced variable name (or) simply object

Note:

- If non-static variables are accessed directly by their name (or) using class name it leads to compile-time error "non-static variable cannot be referenced from static context".
- If non-static variables are accessed using null referenced variable it leads to run-time error "java.lang.NullPointerException"

Find out CE and RE in the below program, comment them, execute and print output?

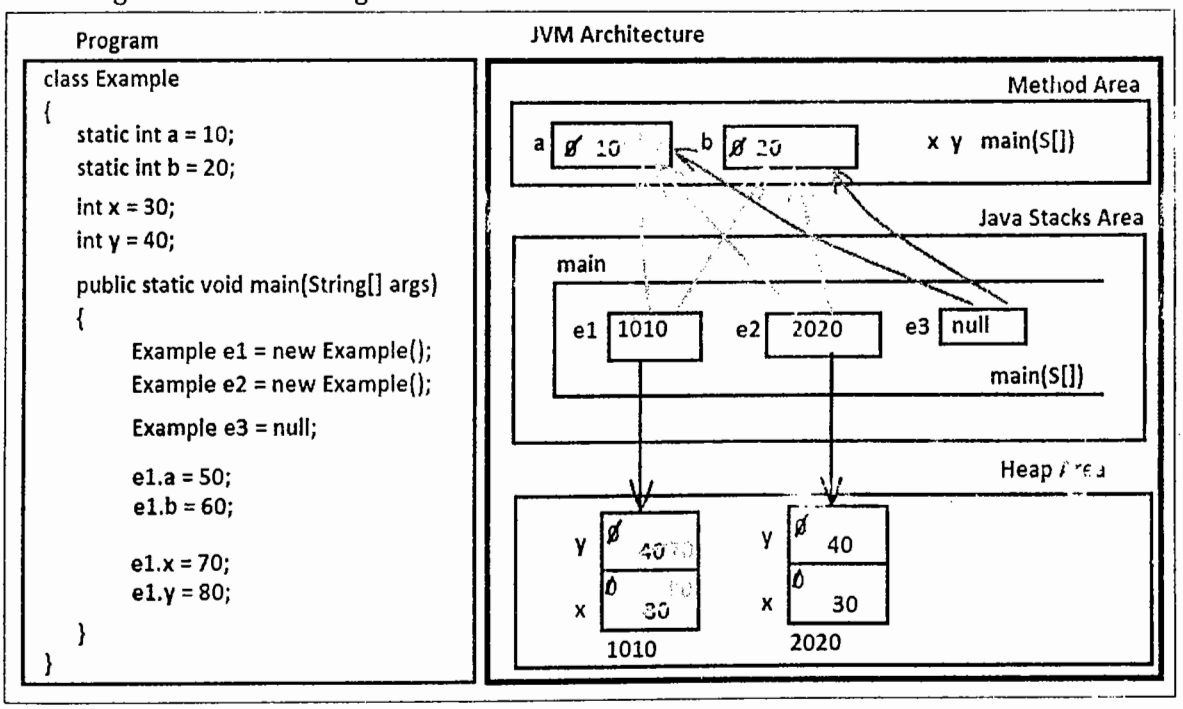
```
class Example
{
  static int a = 10;
  static int b = 20;

  int x = 30;
  int y = 40;

  public static void main(String[] args)
  {
      Sopln("a: "+a);
      Sopln("b: "+b);

      Sopln("a: "+Example.a);
      Sopln("b: "+Example.b);

      Example e = new Example();
      Sopln("x: "+e.x);
      Sopln("y: "+e.y);

      Sopln("a: "+e.a);
      Sopln("b: "+e.b);

      Sopln("x: "+Example.x);
      Sopln("y: "+Example.y);

      Example e1 = null;
      Sopln(e1.a);
      Sopln(e1.x);
  }
}
```

Modifying static and non-static variables

If we modify static variables using one object that modification is affected to all objects, because all objects share same copy of static variable's memory location.

If we modify non-static variables using one object that modification is not affected to other objects, because every object has its own copy of non-static variable's memory location.

Below diagram shows creating class with static and non-static variables and their modifications



Non-static Methods

The method which doesn't have static keyword in its prototype is called non-static method. Like non-static variables, non-static methods also should be called only by using object.

Does JVM execute non-static method automatically when object is created?

No, we should call them explicitly using object. If you call them directly from main() method it leads to CE: "non-static method cannot be referenced from static context"

What is the output from the below programs, find out if there are any CE?

```
class Example
{
    void m1()
    {
        System.out.println("In m1 method");
    }
    public static void main(String[] args)
    {
        System.out.println("In main method");
        m1();
    }
}
```

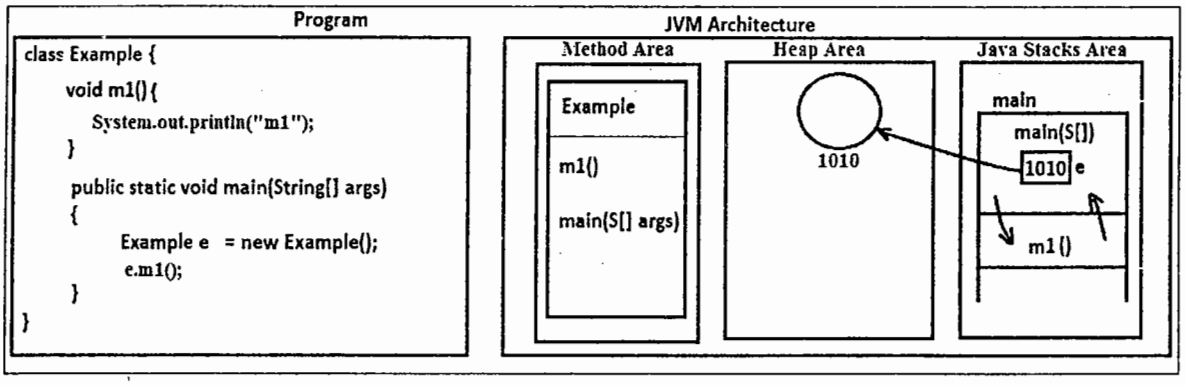
```
class Example
{
    void m1()
    {
        System.out.println("In m1 method");
    }
    public static void main(String[] args)
    {
        System.out.println("In main method");

        Example e = new Example();
        e.m1();
    }
}
```

Q) Can we create object without having non-static variable in the class?

A) Yes, it is possible. JVM creates dummy object.

Dummy object means it does not have any state, but will have behavior and hash code.

Check below diagram



What is the order of execution of non-static methods?

In the order they are called.

Where non-static methods are stored and executed?

Non-static methods logic (bytecodes) is stored in Method Area, and when it is called its logic is loaded and executed in Java Stacks Area by creating separate stack frame in main thread.

Q) How can we call non-static members from other non-static members?
A) directly by their name.

Check below diagram

```
class Example
{
        int x = 10;
        Int y = 20;

        void m1()
        {
                System.out.println("m1");
        }
        void printXY()
        {
                System.out.println(x);
                System.out.println(y);

                m1();
        }
}
```

Q) How is it possible to call non-static members directly without object?
We can call non-static members from other non-static members directly, since we can guarantee providing memory for non-static variables and non-static methods before their calling statement is executed by JVM.

How?
Well, Answer below two questions,

Q1) *Basically what do you need to call a variable or method?*
That variable or method must have memory location.

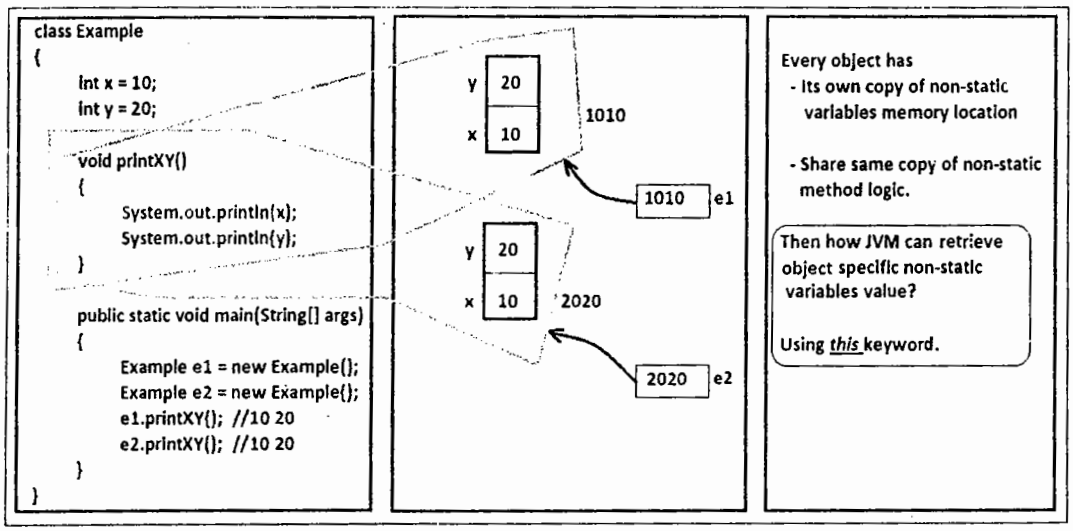
Q2) *In the above program how can you call printXY() method?*
With object right, just think when we create object, non-static variables and other non-static members are also gets memory location right.

Check below program

```
class Example
{
        int x = 10;
        int y = 20;

        void m1(){
                System.out.println("m1");
        }
        void printXY(){
                System.out.println(x);
                System.out.println(y);

                m1();
        }
        public static void main(String[] args) {
                Example e = new Example();
                e.printXY();
        }
}
```

Q) Then why we cannot call non-static members from static members directly?
In this case we cannot guarantee providing non-static members memory location before their calling statement execution, because static members can be accessed directly without object creation. Hence compiler doesn't allow calling non-static members directly from static members.

Q) What is the object structure with non-static method? Will non-static method get separate memory for each object like variables?
No, every object doesn't have separate copy of non-static method rather <u>all objects share *same copy of non-static method logic*</u> as shown in the below diagram.



In the above program,
- In *e1.printXY()* method call, JVM reads non-static variables from e1 object.
- In *e2.printXY()* method call, JVM reads non-static variables from e2 object.

Conclusion:
In a non-static context, if non-static variables are called directly by their name, JVM reads their values from the *current object*.

What is a *current object*?
The object that is used to call a non-static method is called current object.

For Example
In the above program,
- In e1.printXY() method call, e1's object is the current object
- In e2.printXY() method call, e2's object is the current object

Q) Since method logic is shared by all objects, how can JVM retrieve object specific non-static members?
Using ***this*** keyword

Introduction to this keyword

The keyword ***this*** is a non-static final referenced variable used to store current object reference to separate non-static variables of different objects in non-static context.

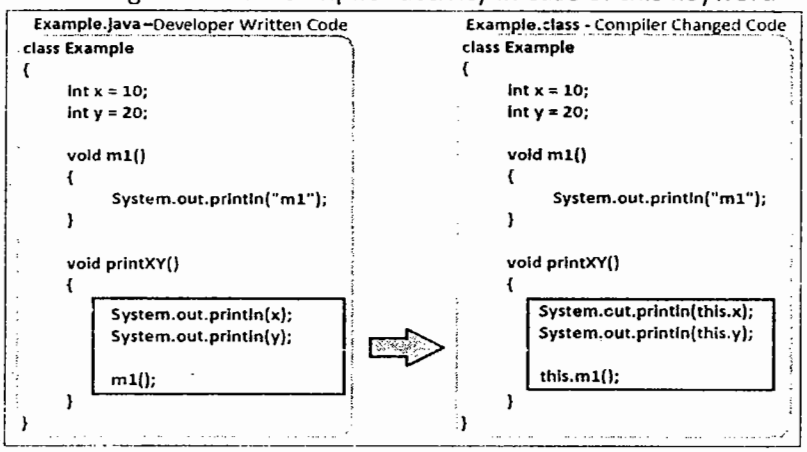
Q) Who will place *this* keyword in calling non-static members from other non-static members?

In compilation phase compiler places *this* keyword in all non-static members call if they are called directly by their name.

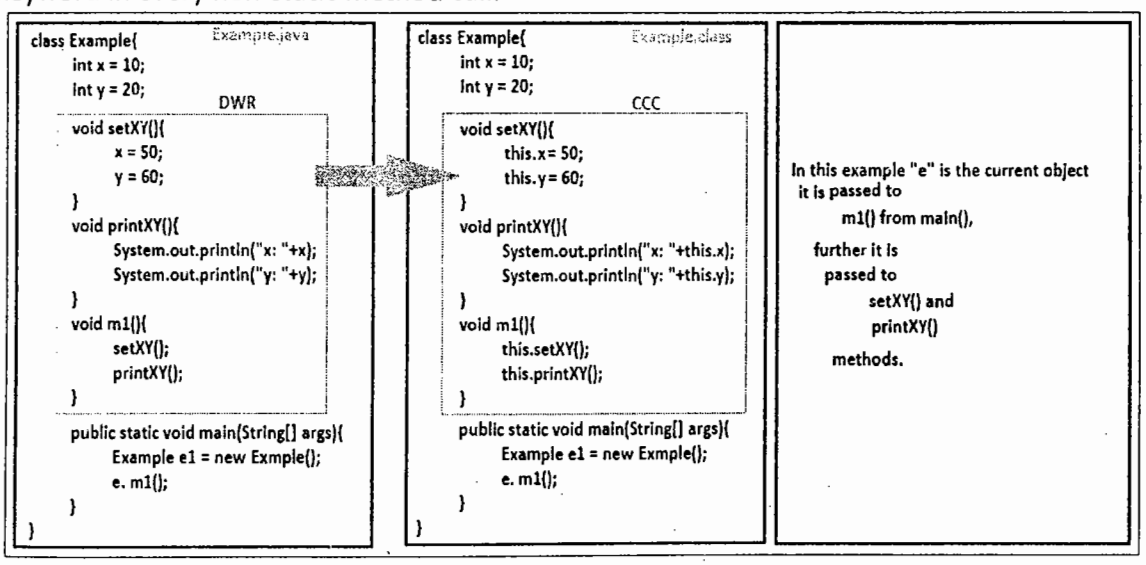
Who does fill this keyword with current object reference?

JVM stores the current object reference when a non-static member is called. The first step in non-method execution is filling *this* referenced variable with current object reference, then that method logic is executed. After that method execution completion, *this* referenced variable destroyed. Also, the associated pointer to that object is destroyed.

Below diagram shows compiler activity in case of this keyword



Q) When non-static method is shared by all objects, why its call is replaced with *this* keyword? Because current object reference should be passed in to every non-static method, since in that method also developer may uses non-static variables. So compiler places *this* keyword in every non-static method call.



Local object creation

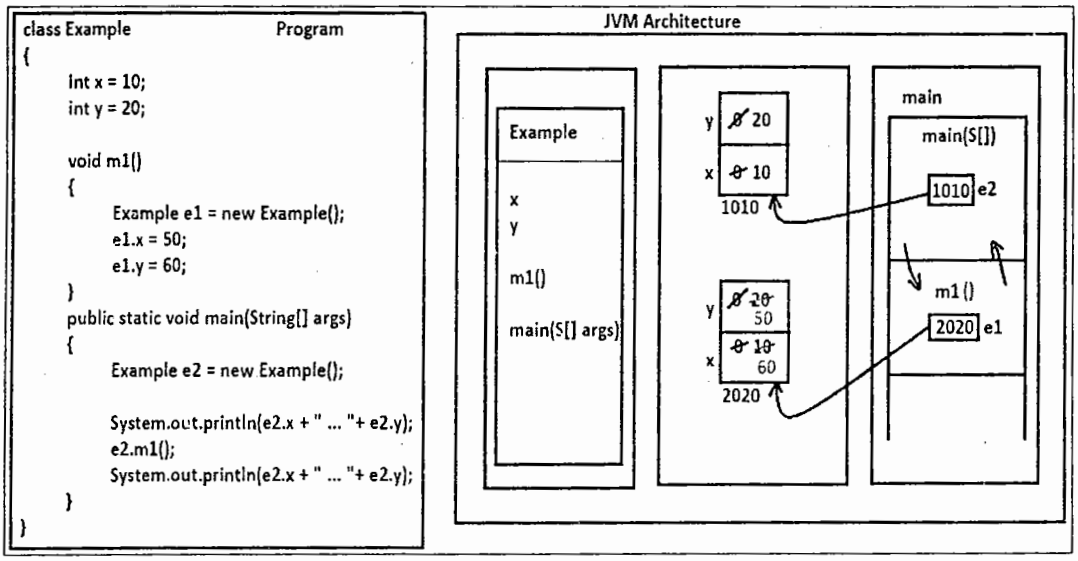
Q) Can we create object in non-static method?

Yes, we can also create object in non-static methods, this object is called local object.

For example

```
void m1(){
        Example e = new Example();
}
```

This object is accessible only within that method as it's referenced variable is local to that method. So, once method execution is completed this local object's reference variable is destroyed then object is become unreferenced. <u>If you modify non-static variables using local object's referenced variable those modifications are stored in that local object, ***not in*** current object, current object data is not be modified</u>. Check below program and JVM architecture.



In the above program the object referenced by *e1* is local to m1() method, and the object referenced by *e2* is local to main. So the modification we have done on non-static variable using *e1* in m1() method are stored in local object, and e2 object data is not modified. Hence, before m1() method call and after m1() method call you get same values *10 ... 20* from *e2* object.

Q) How many referenced variables can an object has pointing to it?

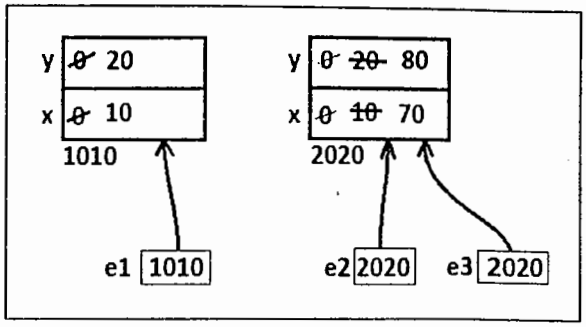
A) More than one.

If we modify that object state using one referenced variable then other referenced variable also get same modified values, because all referenced variables are pointing to same object.

Below program explains above concept.

```
class Example{
        int x = 10;
        int y = 20;

        public static void main(String[] args) {

                Example e1 = new Example();
                Example e2 = new Example();
                Example e3 = e2;

                e2.x = 70;
                e2.y = 80;

                System.out.println(e1.x + " ... " + e1.y);
                System.out.println(e2.x + " ... " + e2.y);
                System.out.println(e3.x + " ... " + e3.y);
        }
}
```

Memory location structure



Output

```
======
10 ... 20
70 ... 80
70 ... 80
```

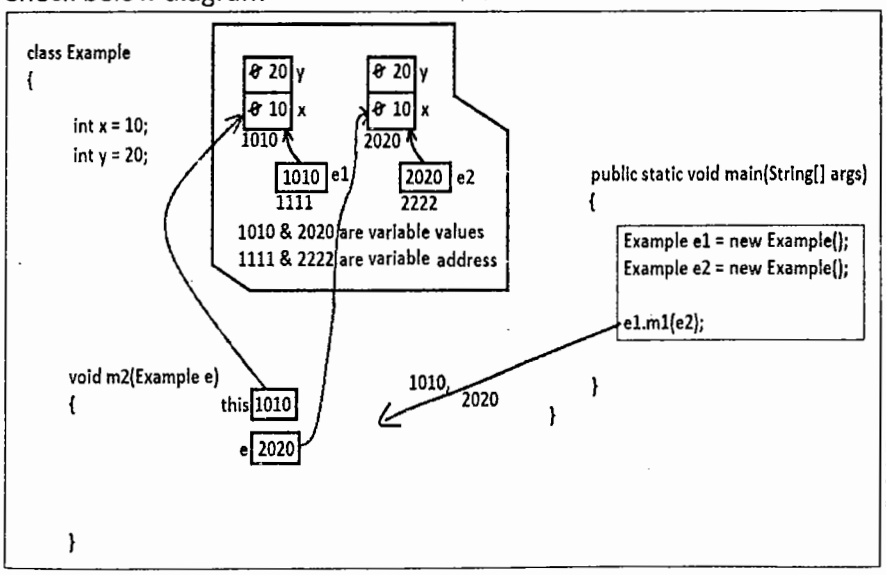
Q) <u>Can we pass object as an *argument* to a method?</u>

Yes, we can pass object as an argument to a method.

To pass object as an argument to a method, the method parameter type should be that object's class type. When we pass object as an argument to a method, its reference is passed but not object memory, that reference is stored in the parameter variable.

Then, that parameter referenced variable also pointing to the same object that is pointed by the argument referenced variable.

Check below diagram



In this diagram we called *<u>m1() method on e1 object by passing e2 object</u>*.
So, in this method call

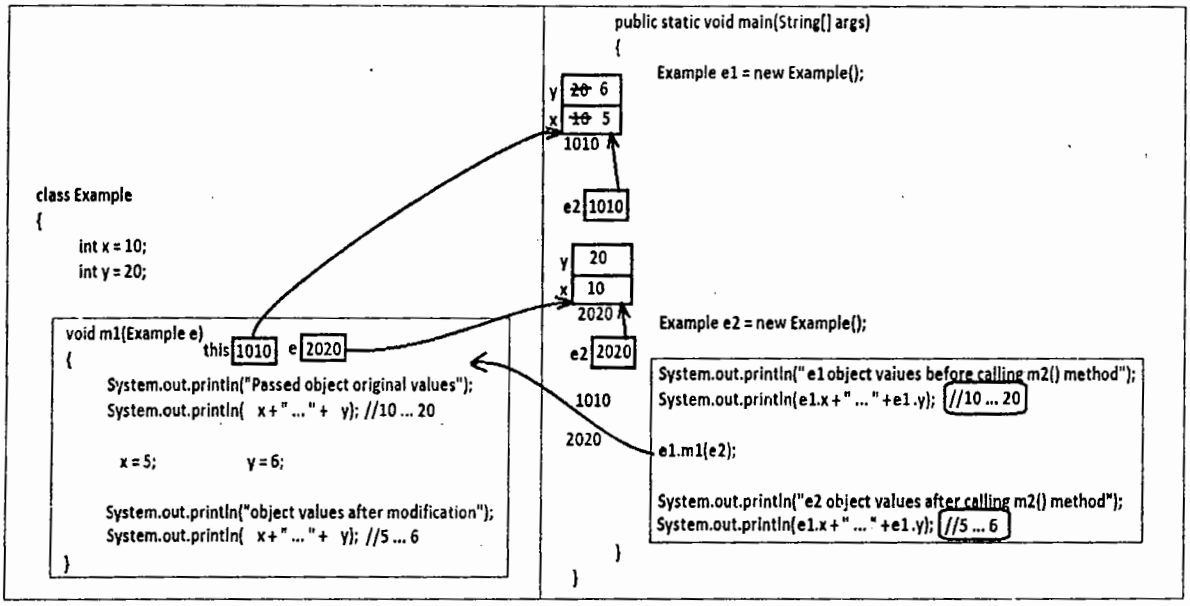
- *e1* object is called *current object*
- *e2* object is called *argument object*

Then, in this method *e1 object* is pointed by ***this*** keyword, and *e2 object* is pointed by parameter ***e***. After this method execution both *this* and *e* variables are killed.

So, till the method execution is completed *current object* and *argument objects* are pointed by two referenced variables. As we learn already, when an object is pointed by multiple referenced variables, if it is modified by one referenced variable other referenced variables are also get the same modified values.

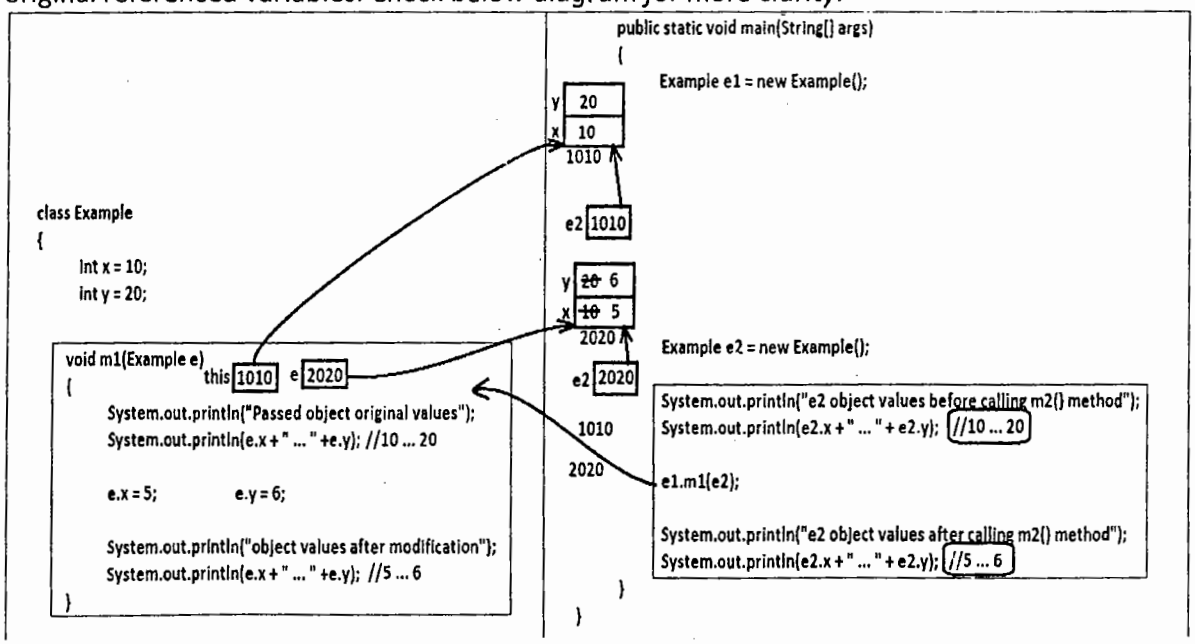
Modifying current object *using this variable*

In the above program you have learned modified argument object, in this program you will learn modifying current object also. If you modify non-static variables directly by their name or using *this* keyword, current object values are modified. *Check below diagram for more clarity.*



Modification argument object using parameter variable

If we change object values using *parameter*, the same modified values are retrieved by using original referenced variables. *Check below diagram for more clarity.*



Q) Can we call a method by passing same object as current object and argument object?
Yes, we can call method by passing same object as *current object* and also *argument object*, provided that method is defined in that current objects referenced variable's type class and with the same class type parameter.

Below program explains modifying current and argument objects in the single method.
What is the output we get from below program?

```
class Example{

        int x;
        int y;

        void m1(Example e){

//modifying current object values
        x = x + 1;
        y = y + 2;

//modifying argument object values
        e.x = e.x + 3;
        e.y = e.y + 4;

        }

    public static void main(String[] args) {
        Example e1 = new Example();
        Example e2 = new Example();
        e1.m1(e2);
        System.out.println(e1.x + " ... " + e1.y);
        System.out.println(e2.x + " ... " + e2.y);
        e2.m1(e1);
        System.out.println(e1.x + " ... " + e1.y);
        System.out.println(e2.x + " ... " + e2.y);
        e1.m1(e1);
        System.out.println(e1.x + " ... " + e1.y);
        System.out.println(e2.x + " ... " + e2.y);
        e2.m1(e2);
        System.out.println(e1.x + " ... " + e1.y);
        System.out.println(e2.x + " ... " + e2.y);
    }
}
```

Strict rule: Solve above program only by drawing objects memory, else definitely you will get wrong results.

Pass by *value* and Pass by *reference* or *address*

In programming languages we can call methods either by passing variable value (or) variable address. Calling a method by passing *variable value* is called "*pass by value*" and calling a method by passing *variable address* is called "*pass by address*" and there is no concept called "*pass by reference*" as per language designers.

Does Java support pass by address?

No, Java doesn't support pass by address, because it doesn't support pointers.

Controversy point

Can we consider invoking a method by *passing object* as an argument is *pass by reference*?

No. Many of the new learners are considering it as pass by reference, because when you pass object as an argument, its reference is passed into that method.

Really speaking ***it is false***, it is not a *pass by reference*. It is also treated as pass by value.

It is clearly written in SUN specification and tutorial, ***Java does not support pass by reference***. Calling a method by passing object directly or via a referenced variable is also considered as "*pass by value*", below is the document copied from the Java tutorial.

Passing Primitive Data Type Arguments

Primitive arguments, such as an `int` or a `double`, are passed into methods *by value*. This means that any changes to the values of the parameters exist only within the scope of the method. When the method returns, the parameters are gone and any changes to them are lost. Here is an example:

```
class Example
{
    void m1(int p)
    {
        p = 60;
    }

    public static void main(String[] args)
    {
        Example e = new Example();

        int x = 50;
        e.m1(x);

        System.out.println(x);
    }
}
```


As you can observe here, in m1() method call, variable value 50 is passed but not variable address 1111, this is the reason this called "call by value".

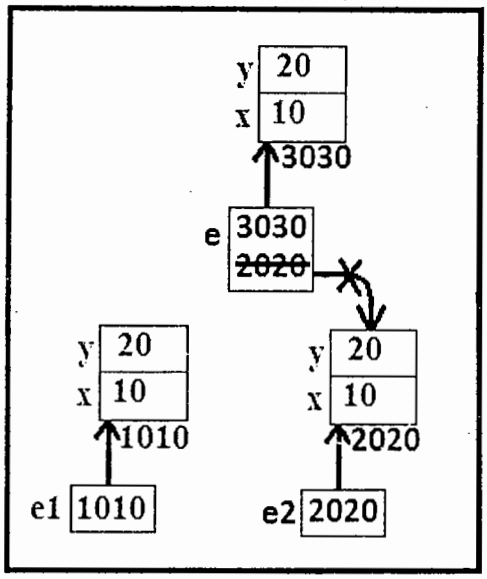
As you can observe, even if we change parameter value, argument variable ***x*** value is not changed. After the above program execution, still you will get **x = 50**.

Passing Reference Data Type Arguments

Reference data type parameters, such as objects, are also passed into methods *by value*. This means that when the method returns, the passed-in reference still references the same object as before.

Below diagram shows calling method **by passing object reference**

```
class Example{
        int x = 10, y = 20;
        void m1(Example e){
                System.out.println("\te: "+e);
                e = new Example();
                System.out.println("\te: "+e);
        }
        public static void main(String[] args){
                Example e1 = new Example();
                Example e2 = new Example();
                System.out.println("e2: "+e2);
                e1.m1(e2);
                System.out.println("e2: "+e2);
        }
}
```



Below is the output from the above program

```
Select C:\Windows\system32\cmd.exe

D:\NareshIT\HariKrishna\CoreJava&OCJP\10NSM>javac Example.java

D:\NareshIT\HariKrishna\CoreJava&OCJP\10NSM>java Example
        e2: Example@19821f

                e: Example@19821f
                e: Example@addbf1

        e2: Example@19821f

D:\NareshIT\HariKrishna\CoreJava&OCJP\10NSM>_
```

In the above program we have assigned new object to parameter variable, after that method execution, in main method e2 is still pointing to its own object.

As shown in the above memory location diagram and output, e2 object reference is not changed even if we changed parameter value.

Since argument variable value is not changed when we changed parameter value, so, this point proves that passing objects as argument is also comes under *pass by value*.

However, the values of the object's non-static variables *can* be changed in the method and those object modifications are effected to passed-in referenced variable, because both passed-in referenced variable and method parameter pointing to same object.

Conclusions from all above points

As you noticed in the above two diagrams, the difference in passing primitive value and object reference as argument is –

- in case of invoking method by passing primitive value - the method parameter variable is not pointing to argument's variable but
- in case of invoking method by passing object reference the parameter variable is pointing to the same object that is pointed by argument referenced variable.

Determine the effect upon object references and primitive values when they are passed into methods that perform assignments (or) other modifying operations on the parameters.

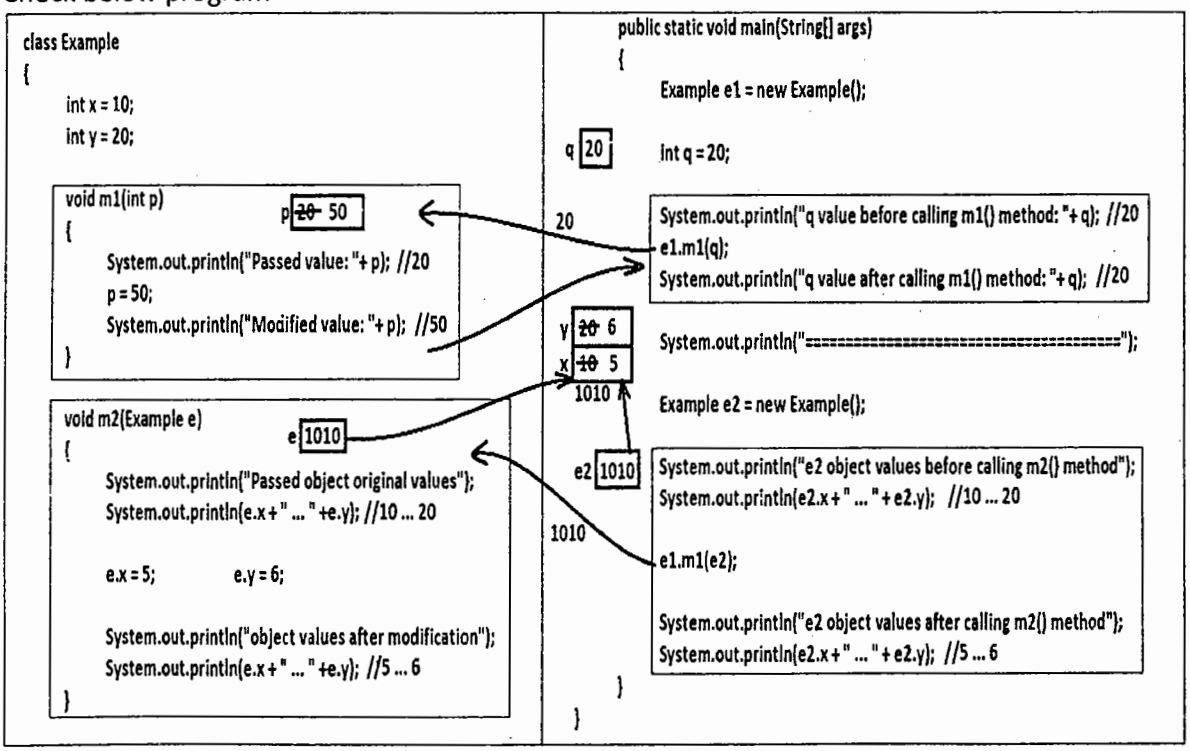
Passing Primitive value as argument

When we pass primitive values as an argument to a method and if we modify those values using method parameter, those **modifications are not affected** to original variable. Because parameter variable and argument variable does not have link.

Passing object reference as argument

When we pass object reference as an argument to a method and if we modify that object using method parameter, those **modifications are effected** to original reference variable, because parameter variable and argument variable are both pointing to same object.

Check below program



Answer below questions

In how many ways can we send a primitive value from one method to another method?
only as argument.

In how many ways can we pass object reference from one method to another method?
We can pass object reference in two ways to a non-static method

1) As **Current object**
2) As **Argument Object**

In that called method

current object members are accessed using ***this*** keyword, and
argument object members are accessed using **parameter name.**

Note: But we cannot pass object as current object to a *static method,* we can only pass object *as argument*, because we cannot use *this* keyword in static context.

Shadowing - Local preference in non-static members
Creating a local variable with the same non-static variable name is called shadowing.
In a non-static method if there is any local variable or parameter defined with same non-static variable name always local variable value is retrieved.

What is the output from the below program?

DWC

```
class Example
{
    int x = 10;
    int y = 20;

    void m1()
    {
        System.out.println( x );
        System.out.println( y );

        int x = 50;
        System.out.println( x );
        System.out.println( y );
    }
    public static void main(String[] args)
    {
        Example e1 = new Exmple();
        e. m1();
    }
}
```

CCC

```
class Example
{
    int x = 10;
    int y = 20;

    void m1()
    {
        System.out.println(this.x);
        System.out.println(this.y);

        int x = 50;
        System.out.println( x );
        System.out.println(this.y);
    }
    public static void main(String[] args)
    {
        Example e1 = new Exmple();
        e. m1();
    }
}
```

O/P

Q) Then how can we access non-static variable from a non-static method in presence of local variable if both have same name? Or how can we differentiate non-static variable from local variable if both have same name?

Using ***this*** keyword we can differentiate non-static variable from local variable. In presence of local variable we must use *this* keyword explicitly to access non-static variable, as shown below

What is the output from the below program?

DWC

```
class Example
{
    int x = 10;
    int y = 20;

    void m1()
    {
        System.out.println( x );
        System.out.println( y );

        int x = 50;

        System.out.println( x );
        System.out.println(this.x);
        System.out.println( y );
    }
    public static void main(String[] args)
    {
        Example e1 = new Exmple();
        e. m1();
    }
}
```

CCC

```
class Example
{
    int x = 10;
    int y = 20;

    void m1()
    {
        System.out.println(this.x);
        System.out.println(this.y);

        int x = 50;

        System.out.println( x );
        System.out.println(this.x);
        System.out.println(this.y);
    }
    public static void main(String[] args)
    {
        Example e1 = new Exmple();
        e. m1();
    }
}
```

O/P

Q) When does compiler place *this* keyword in accessing non-static variables (or) when developer should place *this* keyword explicitly in accessing non-static variable?

If there is any local variable or parameter defined in the current method with the same non-static variable name compiler does not place *this* keyword. In this case developer must use this keyword explicitly to access non-static variable as shown in the above diagram.

What is the output of the below program?

```
class Example
{
    int x = 10;
    int y = 20;

    void m1(int a, int b)
    {
        x = a;
        y = b;
    }

    void m2(int x, int y)
    {
        x = x;
        y = y;
    }

    void m3(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public static void main(String[] args)
    {
        Example e1 = new Example();

        System.out.println(e1.x + " ... "+ e1.y);

        e1.m1(30, 40);
        System.out.println(e1.x + " ... "+ e1.y);

        e1.m2(50, 60);
        System.out.println(e1.x + " ... "+ e1.y);

        e1.m3(70, 80);
        System.out.println(e1.x + " ... "+ e1.y);
    }
}
```

Note: We can also access static variables using *this* keyword but only in non-static members.

Rules on this Keyword

1) We cannot use *this* keyword in static members it leads to compile time error, Since it is non-static variable. Check below diagram

```
class Example
{
        int x = 10;
        int y = 20;

        void m1()
        {
                System.out.println(this.x);   ✓
                System.out.println(this.y);   ✓
        }
        public static void main(String[] args)
        {
                System.out.println(this.x);   ✗ CE: non-static variable this cannot be refereced from static context
                System.out.println(this.y);   ✗ CE: non-static variable this cannot be refereced from static context
        }
}
```

2) We cannot explicitly initialize it with either null (or) with object references since it is a final variable violation leads to compile time error.

```
class ThisRule2
{
        void m1()
        {
                ThisRule2 tr = new ThisRule2();

                this = tr;  ✗ CE: cannot assign a value to final variable this

                this = null;  ✗ CE: cannot assign a value to final variable this
        }
}
```

Q) What is the type of *this* keyword?

A) ***current class*** type is its type, because it should store current class object reference. So to prove this point assign another class object reference in current class you will get compiler time error **incompatible types,** as shown below.

```
class ThisRule3
{
        void m1()
        {
                Example e = new Example();

                this = e;  ✗ CE: incompatible types
                                found: Example
                                required: ThisRule3
        }
}
```

Constructor

Definition

Constructor is a special method given in OOP language for creating and initializing object. In Java, constructor role is only initializing object, and new keyword role is creating object. In C++, constructor alone creates and initializes object.

Rules in Defining Constructor

1) Constructor name should be same as class name.
2) It should not contain return type.
3) It should not contain modifiers
4) In its logic return statement with value is not allowed

Note:

5) It can have all four accessibility modifiers.
6) It can have parameters.
7) It can have throws clause - it means we can throw exception from constructor.
8) It can have logic, as part of logic it can have all Java legal statement except return statement with value.
9) We can place *return;* in constructor.

Constructor creation **syntax**

```
class Example
{
    Example()
    {
        ---------
        ---------
        ---------
    }
}
```

All Java legal statements are allowed except <u>return statement with value</u>.

Below application shows defining a class with constructor and its execution

```
class Example {
    Example()
    {
        System.out.println("constructor");
    }
    public static void main(String[] args)
    {
        System.out.println("main");
        Example e = new Example();
    }
}
```

O/P:

```
main
constructor
```

Rules in calling constructor

Constructor must be called along with *"new"* keyword; else it leads to compile time error.

Check below program

```
class Example
{
        Example()
        {
                System.out.println("constructor");
        }
        public static void main(String[] args)
        {
                System.out.println("main");

                Example e = new Example();

                Example();
        }
}
```

CE: cannot find symbol
symbol: method Example()

Q) Can we define a method with same class name?

Yes, it is allowed to define a method with same class name. No compile time error and no runtime error is raised, but it not recommended as per coding standards.

Q) If we place return type in constructor prototype will leads to CE?

No, because compiler and JVM consider it as method.

Below program shows defining a method with the same class name

```
public class Example
{
    Example()
    {
        System.out.println("In Constructor");
    }

    void Example()
    {
        System.out.println("In method");
    }

    public static void main(String[] args)
    {
        Example e = new Example();   <- Constructor call

        e.Example();   <- method call
    }
}
```

O/P

In constructor
In method

Q) How compiler and JVM can differentiate constructor and method definitions if both have same class name?
By using *return type*, if there is a return type it is considered as method else it is considered as constructor.

Q) How compiler and JVM can differentiate constructor and method invocations if both have same class name?
By using *new* keyword, if new keyword is used in calling then constructor is executed else method is executed.

Q) Why return type is not allowed for constructor?
As there is a possibility to define method with same class name, return type is not allowed to constructor to differentiate constructor block from method block.

Q) Why constructor name should be same as class name?
Every class object is created by using the same new keyword, so it must have information about the class to which it must create object. For this reason constructor name should be same as class name.

Q) Can we declare constructor as private?
Yes, we can declare constructor as private. All four accessibility modifiers are allowed to constructor. We should declare constructor as private for not to allow user to create object from outside of our class. Basically we will declare constructor as private to implement singleton design pattern.

Q) Is constructor definition mandatory in class?
No, it is optional. If we do not define constructor compiler define constructor.

Check below diagram

Example.java Developer written code

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

As you can observe in the above diagram, constructor is automatically placed by compiler in Example.class.

Example.class Compiler changed code

```
class Example
{
        Example()
        {
                super();
        }

        public static void main(String[] args)
        {
                System.out.println("main");
        }
}
```

It is placed by ***compiler*** automatically

Q) Can we define empty class? If so, is it really an empty class?
Yes we can define empty class, but after compilation it has constructor definition that is placed by compiler. So really it is not an empty class.

Check below diagram

Example.java Developer written code

```
class Example
{
}
```

Example.class Compiler changed code

```
class Example extends java.lang.Object
{
    Example()
    {
        super();
    }
}
```

It is placed by ***compiler*** automatically

Q) How can we know that compiler places constructor in class file?
We should use "**javap**" tool. It is a java binary file available in "jdk\bin" folder. It is used to decompile class file to check that class members.

After compilation use ***javap*** tool as shown below

```
C:\Windows\system32\cmd.exe
D:\NareshIT\JavaHari\NonStaticMembers:javac Example.java

D:\NareshIT\JavaHari\NonStaticMembers:javap Example
Compiled from "Example.java"
class Example extends java.lang.Object{
    Example();
}

D:\NareshIT\JavaHari\NonStaticMembers:_
```

Note: javap tool only displays class member's prototype.
it does not show body and logic.

Types of constructors
Java supports 3 types of constructors

1. Default constructor
2. No-argument | non-parameterized constructor
3. Parameterized constructor

Definitions
+ The compiler given constructor is called default constructor. It does not parameters and logic except *super()* call.

+ The developer given constructor without parameters is called no-arg | non-parameterized constructor.
+ The developer given constructor with parameters is called parameterized constructor.
+ In developer given constructor we can have logic.

Rule: *super()* call must also be placed as a first statement in developer given constructors. If developer does not place it, compiler places this statement.

Q) Why compiler given constructor is called default constructor?
Because it obtain all its default properties from its class
They are:

1. Its accessibility modifier is same as its class accessibility modifier
2. Its name is same as its class name
3. It does not have parameters and logic

Q) What is the default accessibility modifier of a default constructor?
It is assigned from its class. So, it may be default or public. Check below programs

Case #1: class is created with default accessibility modifier, so constructor is also created with default accessibility modifier

```
class Example
{
}
```

```
D:\NareshIT\JavaHari\NonStaticMembers:javac Example.java

D:\NareshIT\JavaHari\NonStaticMembers:javap Example
Compiled from "Example.java"
class Example extends java.lang.Object{
    Example();
}

D:\NareshIT\JavaHari\NonStaticMembers:_
```

Case #2: class is created with public accessibility modifier, so constructor is also created with public accessibility modifier

```
public class Example
{
}
```

```
D:\NareshIT\JavaHari\NonStaticMembers:javac Example.java

D:\NareshIT\JavaHari\NonStaticMembers:javap Example
Compiled from "Example.java"
public class Example extends java.lang.Object{
    public Example();
}

D:\NareshIT\JavaHari\NonStaticMembers:
```

Q) Why compiler defines default constructor without logic and parameters?
Because compiler does not know logic required for your class. The object's initialization logic is different from one class to another class.

But it places super() method call in all constructors because it is generic logic required for every class for calling super class constructor in order to initialize super class non-static variables when subclass object is created.

Since there is no logic in default constructor compiler defines constructor without parameters.

Q) When does compiler provide default constructor?,
Only if there is no explicit constructor defined by developer.

Q) When developer must provide constructor explicitly?
If we what to execute some logic at the time of object creation, that logic may be object initialization logic or some other useful logic.

Find out if there are any compile time errors in the below cases. If there are no compile time errors identify what is the constructor used in object creation.

Before answering below cases, first compile it yourself and add missing code to your program that is placed by complier, after that try executing the program yourself to get output. Habituate this procedure always to get right output.

Case #1:

```
class Example{
        public static void main(String[] args) {
                System.out.println("In main");
                Example e = new Example();
        }
}
```

Case #2:

```
class Example{
        Example(){
                System.out.println("In no-arg constructor");
        }
        public static void main(String[] args) {
                System.out.println("In main");
                Example e = new Example();
        }
}
```

Case #3:

```
class Example{
        Example(){
                System.out.println("In no-arg constructor");
        }
        Example(int x){
                System.out.println("In int-arg constructor");
        }

        public static void main(String[] args) {
                System.out.println("In main");
                Example e = new Example();
        }
}
```

Case #4:

```
class Example{
        Example(int a){
                System.out.println("In int-arg constructor");
        }
        public static void main(String[] args) {
                System.out.println("In main");
                Example e = new Example();
        }
}
```

Case #5:

```
class Example{
        Example(int a){
                System.out.println("In int-arg constructor");
        }
        public static void main(String[] args) {
                System.out.println("In main");
                Example e = new Example(50);
        }
}
```

Answers

Case #1:

NO CE, NO RE,
output:
In main
Object is created using default constructor.

Case #2:

NO CE, NO RE,
output:
In main
In no-arg constructor
Object is created using no-arg constructor.

Case #3:

NO CE, NO RE,
output:
In main
In no-arg constructor
Object is created using no-arg constructor.

Case #4:

This case compilation leads to CE:
cannot find symbol
Symbol: constructor Example()

In this class we do not have no-arg constructor and also default constructor. Compiler will not provide default constructor because this class has excplit constructor.

Solution: object should be created only by using parameterized constructor as shown below in

Case 5

Parameterized constructor is called by passing its parameter type argument. So, NO CE, NO RE

Q) If class has explicit constructor, does it has default constructor?
No. Compiler places default constructor only if there is no explicit developer given constructor.

Conclusion: Q) **How can we create object of a class?**
We must create object of a class by using new keyword and available constructor.

Q) Can we consider both default and no-arg constructors are same?
No, both are different. They are seems to be same, but really not same.

Q) What are the differences between no-argument and default constructor?

Default constructor	No-arg constructor
1. It is given by compiler	1. It is given by developer
2. It accessibility modifier is same as class accessibility modifier. So the only allowed accessibility modifiers are *default* or *public*.	2. It can have all four accessibility modifiers as it is defined by developer. So the allowed accessibility modifiers are *private, default, protected, public*
3. It does not have logic except super() call.	3. It can have logic including super() call.

Note: The common point between these two constructors is "both constructors do not have parameters.

Q) Can we define a class with public accessibility modifier and a constructor with other accessibility modifier?
A) Yes we can define. Compiler does not change developer given constructor accessibility modifier to class accessibility modifier.

Q) When should we define parameterized constructor in a class?
To initialize object dynamically with user given values then we should defined parameterized constructor.

What is the difference we can find in objects creation by using all three constructors?
Case 1: object creation with default constructor
The objects created by using default constructor will have same default assigned values.

```
class Example
{
        int x = 10, y = 20;
        public static void main(String[] args){
                Example e1 = new Example();
                Example e2 = new Example();
        }
}
```

In the above case both objects have same default assigned values 10, 20.

Case 2: object creation with no-arg constructor
The objects created by using no-arg constructor will have same values initialized in constructor.

```
class Example{
        int x = 10, y = 20;
        Example(){
                x = 50;
                y = 60;
        }
        public static void main(String[] args){
                Example e3 = new Example();
                Example e4 = new Example();
        }
}
```

In the above case both objects has same values 50, 60.

Case 3: object creation with parameterized constructor
The objects created by using parameterized constructor will have user given values. Those values may be same or different.

```
class Example{
        int x = 10, y = 20;
        Example(int x, int y){
                this.x = x;
                this.y = y;
        }
        public static void main(String[] args){
                Example e5 = new Example(5, 6);
                Example e6 = new Example(5, 6);
                Example e7 = new Example(7, 8);
        }
}
```

In the above case e5 and e6 objects has same user given values "5, 6" and e7 object will have "7, 8".

Q) How many constructors can be defined in a class?
In a class we can define multiple constructors, but every constructor must have different parameters type (or) parameters order. So in a class we can define one no-argument constructor + 'n' number of parameterized constructors.

Q) What is a constructor overloading?
Defining multiple constructors with different parameter types | order | list is called constructor overloading.

Below program shows implementing constructor overloading

```
class Example
{
        Example(){
                Sopln("no-arg constructor");
        }
        Example(int x){
                Sopln("int-arg constructor");
        }

        Example(String s){
                Sopln("String-arg constructor");
        }
```

```
    public static void main(String[] args){
        System.out.println("main");

        Example e1 = new Example();

        Example e2 = new Example(10);

        Example e3 = new Example("abc");
    }
}
```

O/P

```
main
no-arg constructor
int-arg constructor
String-arg constructor
```

Q) In an object creation apart from invoked constructor do other constructors are also executed?

No, only invoked constructor is executed. Other constructors will not be executed.

Q) Then how can we execute logic when an object is created using any of the constructors?

Solution #1

Write that logic in all constructors

Problem: It is not recommended because we lose code reusability and centralized code change. Since code is redundant we must perform code change in every constructor, it leads to lot of maintenance cost because after code change every constructor's logic should check again.

Solution #2

As per modularity write that logic in a non-static method and call that method in all constructors.

Problem: It is also not recommended because there is a chance of missing calling that method in one of the constructors and also that method can be called after object creation.

Solution #3: The best solution for this requirement is Non-Static Block (NSB).

Introduction to Non Static Block

Definition of NSB

A class level block which doesn't have prototype (head) is called non-static block.

Below is the syntax to create NSB

```
class Example
{
    {
        ---------
        ---------
        ---------
    }
}
```

All Java legal statements are allowed except ***return*** statement either with or without value and ***throw*** statement.

Need of NSB

We should define non-static block to execute some logic only at the time of object creation irrespective of the constructor used in object creation.

Q) Who does execute NSB when and where?

NSB is executed automatically by JVM for every object creation in Java stacks area by creating separate stack frame in main thread.

What is the output from the below program?

```
class Example
{
    {
        Sopln("NSB");
    }
    Example(){
        Sopln("No-arg constructor");
    }

    Example(String s){
        Sopln("String-arg constructor");
    }
```

```
public static void main(String[] args){
    System.out.println("main");

    Example e2 = new Example();

    Example e3 = new Example("abc");
  }
}
```

O/P

main

NSB

No-arg constructor

NSB

String-arg constructor

Q) From this output can you conclude **order of execution** of **constructor and non-static block**?
NSB always executed before constructor.

Q) How many non-static blocks can be defined in a class?
We can define multiple non-static blocks.

Q) What is the order of execution of all NSBs?
All non-static blocks are executed by default by JVM in the order they defined from top to bottom before the invoked constructor.

What is the output from below program?

```
class Example
{
        {
                Sopln("NSB1");
        }

        Example(){
                Sopln("No-arg constructor");
        }

        {
                Sopln("NSB2");
        }

public static void main(String[] args){
        System.out.println("main");

        Example e2 = new Example();

    }
}
```

O/P
main
NSB1
NSB2
No-arg constructor

What is the order of execution of all non-static members?
When an object is created first all non-static variables and non-static blocks are executed in the order they are defined from top to bottom, then the invoked constructor logic is executed. Non-static methods are executed only if they are invoked from any of the static or non-static members.

What is the output from below program?

```
class Example{
        int x = m1();
        int m1(){
                Sopln("NSV x");
                return 10;
        }

        {
                Sopln("NSB1");
        }

        Example(){
                Sopln("No-arg constructor");
        }

    public static void main(String[] args){
            System.out.println("main");
            Example e = new Example();
    }

    {
            Sopln("NSB2");
    }

    int y = m2();
    int m2(){
            Sopln("NSV y");
            return 20;
    }
}
```

```
Output
main

NSV x
NSB1
NSB2
NSV y
No-arg constructor
```

What JVM does internally when an object is created?
object creation process
When we create object - first control is sent to invoked constructor, but its logic will not be executed, instead non-static variables and non-static blocks are executed first in the order they are defined from top to bottom, then after constructor logic is executed.

If all non-static variables and non-static blocks execution is completed, we can say object creation is completed

If constructor execution is completed, we can say object initialization is completed.

Who will initialize default and assigned values to non-static variables?

A) new keyword.

Constructor initializes object only with developer or user given values.

Responsibilities of new keyword and constructor in creating object
The responsibility of new keyword is

- In identification phase it creates non-static variables with default values in heap area in continuous memory locations and also identifies non-static blocks

- In execution phase it executes variables and non-static blocks in the order they defined, means it initialized non-static variables with assigned values and executes non-static block logic in JSA by creating separate stack frame.

- Then after execution of non-static variables and non-static blocks, new keyword sends control into the invoked constructor with this object reference to complete object initialization with user given values.

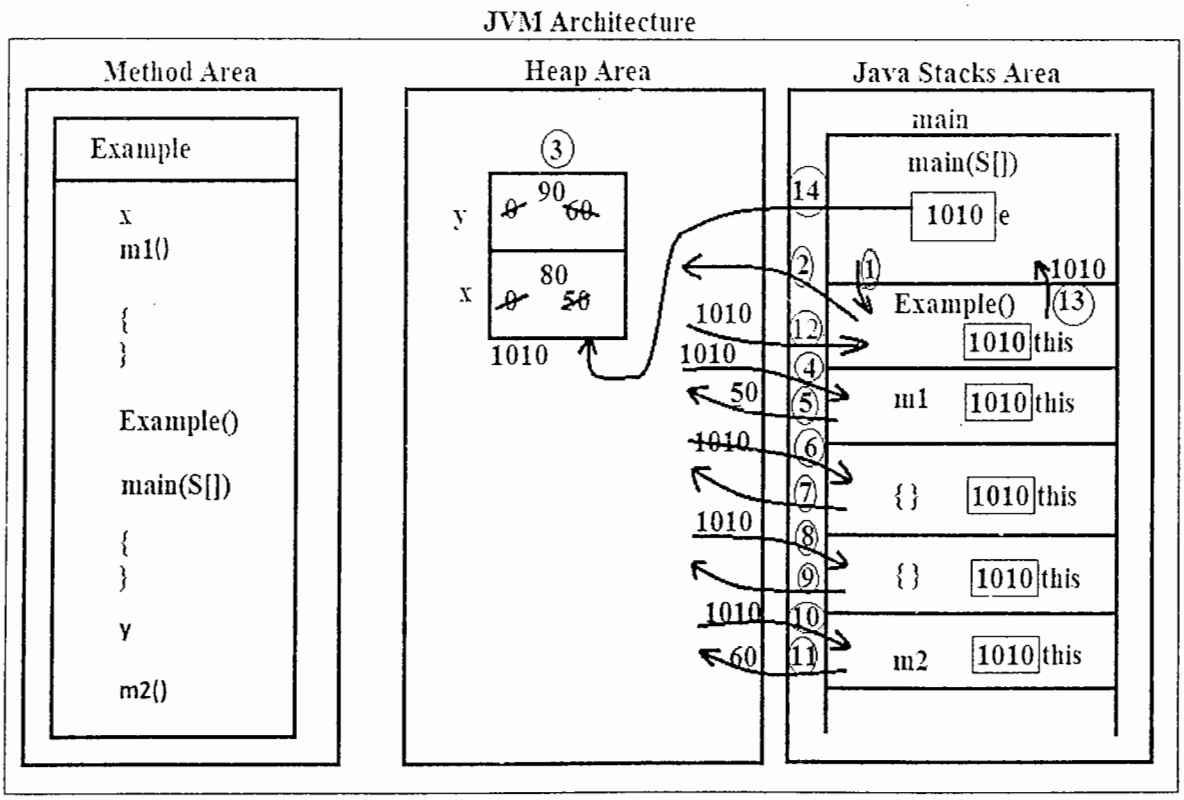
The responsibility of constructor is initializing non-static variables means object with developer or user given values as per its logic.

Then finally new keyword returns object reference to calling area, then it is stored in destination referenced variable.

Below diagrams shows the program with all four non-static members and their execution with JVM Architecture.

```
class Example{
        int x = m1();
        int m1(){
                Sopln("NSV x");
                return 50;
        }

        {
                Sopln("NSB1");
        }

        Example(){
                Sopln("No-arg constructor");

                x = 80;
                y = 90;
        }
```

```
    public static void main(String[] args){
            System.out.println("main");
            Example e = new Example();
    }

    {
            Sopln("NSB2");
    }
    int y = m2();
    int m2(){
            Sopln("NSV y");
            return 60;
    }
}
```



Compile time error: Illegal forward reference

A class level variable should not be accessed *directly* before its creation statement either from other variables, or from blocks. It leads to CE: *illegal forward reference*. But it is possible to access those variables from methods and constructors, because they are always executed after variables execution.

Check below programs

Static variable and block

```
class Example{
        static int a = 10;
        static {
                Sopln("a: "+a);
                Sopln("b: "+b);
        }
        public static void main(String[] args){
                Sopln("a: "+a);
                Sopln("b: "+b);
        }
        static int b = 20;
}
```

Non-static variable and block

```
class Example{
        int x = 10;
        {
                Sopln("x: "+x);
                Sopln("y: "+y);
        }
        public static void main(String[] args){
                Example e = new Example();
                Sopln("x: "+e.x);
                Sopln("y: "+e.y);
        }
        int y = 20;
}
```

Solutions

There are two ways to call the above two variables "b" and "x"

Solution #1: call static variable with "class name" and non-static variable with "this"

```
class Example{
        static int a = 10;
        static {
                Sopln("a: "+ a);
                Sopln("b: "+Example.b);
        }
        public static void main(String[] args){
                Sopln("a: "+a);
                Sopln("b: "+b);
        }
        static int b = 20;
}
```

```
class Example{
        int x = 10;
        {
                Sopln("x: "+x);
                Sopln("y: "+this.y);
        }
        public static void main(String[] args){
                Example e = new Example();
                Sopln("x: "+e.x);
                Sopln("y: "+e.y);
        }
        int y = 20;
}
```

Output

From SB	From main
a: 10	**a: 10**
b: 0	**b: 20**

Output

From NSB	From main
x: 10	**x: 10**
y: 0	**y: 20**

Solution #2: call variables via static and non-static methods.

```
class Example{
        static int a = 10;
        static {
                Sopln("a: "+ a);
                Sopln("b: "+getB());
        }
        public static void main(String[] args){
                Sopln("a: "+a);
                Sopln("b: "+b);
        }
        static int getB(){
                return b;
        }
        static int b = 20;
}
```

```
class Example{
        int x = 10;
        {
                Sopln("x: "+x);
                Sopln("y: "+getY());
        }
        public static void main(String[] args){
                Example e = new Example();
                Sopln("x: "+e.x);
                Sopln("y: "+e.y);
        }
        int getY(){
                return y;
        }
        int y = 20;
}
```

Output

From SB	From main
a: 10	**a: 10**
b: 0	**b: 20**

Output

From NSB	From main
x: 10	**x: 10**
y: 0	**y: 20**

Explanation

In the above two cases variable's default value is printed in both static and non-static blocks. Because by the time blocks are being executed variables execution is not yet completed, they have been just created with default value.

By using above programs we can prove that variables are initialized with default values even though we initialized them with explicit values.

Recursive method and constructor call- RE: *java.lang.StackOverflowError*
Calling a method or constructor from its own block is called recursive method or constructor call. In both cases program execution is terminated with *java.lang.StackOverflowError*.

Because due to recursive method or constructor call stack frames are created continuously without *destroying* previous stack frames. Due to this reason at some point of time there will not be memory in thread to create new stack frame. Hence JVM terminates execution by throwing above exception.

Check below programs – write down output with thread architecture

Recursive method call

```
class A{

        static void m1(){
                System.out.println("m1");
                m1();
        }
        public static void main(String[] args){
                System.out.println("main");
                m1();
        }
}
```

O/P:
main
m1
m1
m1
m1
m1
m1
.
.
.
.
java.lang.StackOverflowError

Recursive constructor call

```
class A{

        A(){
            System.out.println("constructor");
            A a = new A();
        }
        public static void main(String[] args){
                System.out.println("main");
                A a = new A();
        }
}
```

O/P
main
constructor
constructor
constructor
constructor
constructor
constructor
.
.
.
.
java.lang.StackOverflowError

What is the output from below program - write down output with thread architecture

Recursive method call

```
class Example{
        void m1 {
                Sopln("m1 start");
                m1();
                Sopln("end of m1");
        }
        public static void main(String[] args){
                Sopln("main start");
                Example e = new Example();
                Sopln("object is created");
                e.m1();
                Sopln("main end");
        }
        int x = m2();
        int m2(){
                Sopln("NSV x");
                return 50;
        }
        Example(){
                Sopln("constructor start");
                Sopln("end of constructor");
        }
}
```

Recursive constructor call

```
class Example{
        public static void main(String[] args){
                Sopln("main start");
                Example e = new Example();
                Sopln("object is created");
                Sopln("end of main");
        }
        int x = m2();
        int m2(){
                Sopln("NSV x");
                Sopln("end of m2");
                return 10;
        }
        {
                Sopln("NSB start");
                Sopln("end of NSB");
        }
        Example(){
                Sopln("constructor start");
                Example e = new Example();
                Sopln("end of constructor");
        }
}
```

Can we create object in static and non-static blocks?

Yes, we can create object in both static and non-static blocks. But due to the object creation in non-static block it leads to *java.lang.StackOverflowError*, because constructor and non-static blocks are called recursively.

No Exception

```
class A{
        static {
            System.out.println("SB");
            A a = new A();
        }
        A(){
            System.out.println("constructor");
        }
        public static void main(String[] args){
            System.out.println("main");
            A a = new A();
        }
}
```

Recursive constructor call

```
class A{
        {
            System.out.println("NSB");
            A a = new A();
        }
        A(){
            System.out.println("constructor");
        }
        public static void main(String[] args) {
                System.out.println("main");
                A a = new A();
        }
}
```

Can we create object at class level using static and non-static referenced variables?
Yes, we can create object at class level using static and non-static referenced variables, but the object creation using non-static referenced variable also leads to *java.lang.StackOverflowError*.
It is possible to create object at class level using static referenced variable.

No Exception

```
class A{

        static A a = new A();

        A(){
            System.out.println("constructor");
        }
        public static void main(String[] args){
            System.out.println("main");
            A a = new A();
        }

}
```

Recursive constructor call

```
class A{

        A a = new A();

        A(){
            System.out.println("constructor");
        }
        public static void main(String[] args) {
                System.out.println("main");
                A a = new A();
        }

}
```

Conclusion
Object of same class should not be created using non-static referenced variable (or) in non-static block or in same constructor it leads to exception java.lang.*StackOverflowError*.

What is the output from below programs? Give complete answer with thread architecture.

Case #1: Objects creation in static and non-static blocks

```
class Example{
        static {
                Sopln("SB start");
                Example e = new Example();
                Sopln("end of SB");
        }
        {
                Sopln("NSB start");
                Sopln("end of NSB");
        }
        Example()
        {
                Sopln("constructor start");
                Sopln("end of constructor");
        }
        public static void main(String[] args){
                Sopln("main start");
                Sopln("end of main");
        }
}
```

⇨ **No SOE**

```
class Example{
        int x = m1();
        int m1() {
                Sopln("\nNSV x");
                Sopln("end of m1");
                return 10;
        }
        {
                Sopln("NSB start");
                Example e = new Example();
                Sopln("end of NSB");
        }
        Example(){
                Sopln("constructor start");
                Sopln("end of constructor");
        }
        public static void main(String[] args){
                Sopln("main start");
                Example e = new Example();
                Sopln("end of main");

        }
}
```

⇨ **It leads SOE**

Case #2: Objects creation using static and non-static variables

```
class Example{
        static Example e = new Example();
        static {
                Sopln("SB start");
                Sopln("end of SB");
        }
        {
                Sopln("NSB start");
                Sopln("end of NSB");
        }
        Example()
        {
                Sopln("constructor start");
                Sopln("end of constructor");
        }
        public static void main(String[] args){
                Sopln("main start");
                Sopln("end of main");
        }
}
```

⇨ **No SOE**

```
class Example{
        Example e = new Example();
        static {
                Sopln("SB start");
                Sopln("end of SB");
        }
        int x = m1();
        int m1() {
                Sopln("\nNSV x");
                Sopln("end of m1");
                return 10;
        }
        {
                Sopln("NSB start");
                Sopln("end of NSB");
        }
        Example(){
                Sopln("constructor start");
                Sopln("end of constructor");
        }
        public static void main(String[] args){
                Sopln("main start");
                Example e = new Example();
                Sopln("end of main");
        }
}
```

⇨ **It leads to SOE**

For this above case, also write down output for the below sub cases:
Sub case #1: e variable creation statement is placed after x variable creation statement
Sub case #2: e variable creation statement is placed after NSB

What is the out from the below program?

```
class Example{
    int x = 5, y=10;

     public static Example e = new Example();
    {
       System.out.println("initializer block is called...");
       System.out.println(e.x);
       System.out.println(e.y);
    }
    public static void main(String[] args){ }
  }
```

Different ways to execute static & non-static members from other class members

Static members can be accessed in the below 3 ways

1. Using its class name
   Example.m1()
2. Using null referenced variable
   Example e = null;
   e.m1();
3. Using object or object name or object referenced variable
   Example e = new Example();
   e.m1();

Non-static members can be accessed in 1 way

1. Using object or object name or object referenced variable
   Example e = new Example();
   e.m2();

Note: If we access non-static members using null referenced variable no CE but leads to exception NPE

```
Example e = null;
e.m2();
```

Check blow program and give output, find out CE and RE.

Example.java

```
class Example{
        static int a = 10;
        static int b = 20;

        int x = 30;
        int y = 40;

        static void m1()
        {
                Sopln("m1");
        }
        void m2()
        {
                Sopln("NSB start");
                Sopln("end of NSB");
        }
}
```

Sample.java

```
class Sample{
        public static void main(String[] args){
                Sopln("main start");

                Sopln("a: "+a);

                Sopln("a: "+Example.a);

                Example e1 = null;
                Sopln("b: "+e1.b);

                Example e2 = new Example();
                Sopln("b: "+e2.b);
                Example.m1()

                Sopln(Example.x);
                Sopln(e1.x);
                Sopln(e2.x);
                Sopln(e2.y);
                e2.m2();
        }
}
```

In the above cases what is the order of loading Sample and Example classes?
First Sample class is loaded and then Example class is loaded while executing Sample class main method. JVM loads class when it encounters calling any one of the member of that class.

How do we know whether class is loaded or not?
In that class define static block with debug statement Sopln().

What is the output from below program?

Example.java

```
class Example{
        static{
                Sopln("Example class is loaded");
        }
        static void m1(){
                Sopln("Example m1");
        }
}
```

Sample.java

```
class Sample{
        static{
                Sopln("Sample class is loaded");
        }
        public static void main(String[] args){
                Sopln("Sample main start");
                Example.m1();
                Sopln("end of Sample main");
        }
}
```

Output

```
D:\NareshIT\HariKrishna\OCJP\NSM>javac Sample.java
D:\NareshIT\HariKrishna\OCJP\NSM>java Sample
Sample class is loaded
Sample main start
Example class is loaded
Example m1
end of Sample main
```

In how many ways we can load class into JVM?
We have 4 ways to load class into JVM

1. **Using "java" command**
   >java Example
2. **From another class**, by <u>calling static member</u>
3. **From another class**, <u>by creating object</u>
   ```
   class Sample{
           public static void main(String[] args){
                   Example.m1(); //or
                   Example e = new Example();
           }
   }
   ```
4. **Using Reflection API**, Class.forName("<classname>");
   ```
   class Sample{
           public static void main(String[] args) throws ClassNotFoundException{
                   Class.forName("Example");
           }
   }
   ```

In Second, third and fourth cases Example class is loaded due to Sample class execution, and Sample class is loaded using "java" command.

How many times a class is loaded into JVM?
ClassLoader loads class into JVM ***ONLY ONCE*** when it found any one of its member is accessed at first time. Means using either of the above three cases class is loaded only once.

While loading class what are the members executed by JVM in all 4 approaches?
JVM executes static variables and blocks in all 4 approaches, but main method is executed only if class is loaded using "java" command.

Consider below program.

```
class Example{
        static int a = m1();
        static int m1(){
                System.out.println("Example SV a – variable is created");
                return 50;
        }
        static{
                System.out.println("Example SB – class is loaded");
        }
        public static void main(String[] args) {
                System.out.println("Example main – execution started");
        }
        static void m2(){
                System.out.println("Example m2");
        }
        Example(){
                System.out.println("Example constructor – object is created");
        }
}
```

Consider all above points while solving these bits

Case #1: What is the output printed on console if we execute Example class using "java" command?

Output – class is loaded SV, SB executed and then main method is executed

```
D:\NareshIT\HariKrishna\OCJP\NSM>javac Example.java
D:\NareshIT\HariKrishna\OCJP\NSM>java Example
Example SV a - variable is created
Example SB – class is loaded
Example main – execution started
```

Case #2: What is the output after executing below class Sample?
Example class Static member is called

```
class Sample{
        static{
                System.out.println("Sample SB- class is loaded");
        }
        public static void main(String[] args) {
                System.out.println("Sample main – execution is started");
                Example.m1();
        }
}
```

Output – class is loaded SV and SB only executed, then m1 method executed

```
D:\NareshIT\HariKrishna\OCJP\NSM>javac Sample.java
D:\NareshIT\HariKrishna\OCJP\NSM>java Sample
Sample SB – class is loaded
Sample main – execution is started
Example SV a – variable is created
Example SB – class is loaded
Example m1
```

Case #3: What is the output after executing below class Sample?
Example class object is created

```
class Sample{
        static{
                System.out.println("Sample SB- class is loaded");
        }
        public static void main(String[] args) {
                System.out.println("Sample main – execution is started");
                Example e = new Example();
        }
}
```

Output – class is loaded SV and SB only executed, then m1 method executed

```
D:\NareshIT\HariKrishna\OCJP\NSM>javac Sample.java
D:\NareshIT\HariKrishna\OCJP\NSM>java Sample
Sample SB – class is loaded
Sample main – execution is started
Example SV a – variable is created
Example SB – class is loaded
Example constructor – object is created
```

Case #4: What is the output after executing below class Sample?
Example class is loaded using Class.forName()

```
class Sample{
        static{
                System.out.println("Sample SB- class is loaded");
        }
        public static void main(String[] args) throws ClassNotFoundException{
                System.out.println("Sample main – execution is started");
                Class.forName("Example");
        }
}
```

Output – class is loaded SV and SB only executed

```
D:\NareshIT\HariKrishna\OCJP\NSM>javac Sample.java
D:\NareshIT\HariKrishna\OCJP\NSM>java Sample
Sample SB – class is loaded
Sample main – execution is started
Example SV a – variable is created
Example SB – class is loaded
```

From this output we can say "Class.forName()" method only loads class bytecodes. It does not create object.

Case #5: What is the output after executing below class Sample?
Example class is loaded using Class.forName() also object is created.

```
class Sample{
        static{
                System.out.println("Sample SB- class is loaded");
        }
        public static void main(String[] args) throws ClassNotFoundException{
                System.out.println("Sample main – execution is started");
                Class.forName("Example");
                Example e = new Example();
        }
}
```

Output – class is loaded SV and SB only executed from Class.forName() statement. SV and SB will not be executed again when object is created because class is already loaded into JVM

```
D:\NareshIT\HariKrishna\OCJP\NSM>javac Sample.java
D:\NareshIT\HariKrishna\OCJP\NSM>java Sample
Sample SB – class is loaded
Sample main – execution is started
Example SV a – variable is created
Example SB – class is loaded
Example constructor – class object is created
```

Case #6: What is the output after executing below class Sample?
Example class object is created then trying to load class using Class.forName().

```
class Sample{
        static{
                System.out.println("Sample SB- class is loaded");
        }
        public static void main(String[] args) throws ClassNotFoundException{
                System.out.println("Sample main – execution is started");
                Example e = new Example();
                Class.forName("Example");
        }
}
```

Output – There is no change in output, the same above output will come, as shown below.
In this case class is loaded since we are using constructor. Class *will not* be loaded again due to Class.forName(), because class is loaded only once.

```
D:\NareshIT\HariKrishna\OCJP\NSM>javac Sample.java
D:\NareshIT\HariKrishna\OCJP\NSM>java Sample
Sample SB – class is loaded
Sample main – execution is started
Example SV a – variable is created
Example SB – class is loaded
Example constructor – class object is created
```

Case #7: What is the output after executing below class Sample?
Example class null referenced variable is created.

```
class Sample{
        static{
                System.out.println("Sample SB- class is loaded");
        }
        public static void main(String[] args) {
                System.out.println("Sample main – execution is started");
                Example e = null;
        }
}
```

Output – Example class will not be loaded, because no member is called from that class.

```
D:\NareshIT\HariKrishna\OCJP\NSM>javac Sample.java
D:\NareshIT\HariKrishna\OCJP\NSM>java Sample
Sample SB – class is loaded
Sample main – execution is started
```

Case #8: What is the output after executing below class Sample?
Example class null referenced variable is created, and its static method is called using that null referenced variable.

```
class Sample{
        static{
                System.out.println("Sample SB- class is loaded");
        }
        public static void main(String[] args) {
                System.out.println("Sample main – execution is started");

                System.out.println("Before null referenced variable creation");

                Example e = null;

                System.out.println("After  null referenced variable creation");

                System.out.println("Before m2() method is called");

                e.m2();

                System.out.println("After  m2 method is called");
        }
}
```

Output – Example class is loaded when m2() method is called, but not at the time null referenced variable creation. Because JVM thinks, if class is loaded here, if no member is calling from this class it is a waste of memory.

```
D:\NareshIT\HariKrishna\OCJP\NSM>javac Sample.java
D:\NareshIT\HariKrishna\OCJP\NSM>java Sample
Sample SB – class is loaded
Sample main – execution is started

Before null referenced variable creation
After null referenced variable creation

Before m2() method is called
Example SV a – variable is created
Example SB – class is loaded
Example m2
After m2() method is called
```

Write a program to create a class object automatically when it is loaded into JVM.
Create class object in its static block as shown below

```
class Example{
        static{
                System.out.println("Example SB – class is loaded");
                Example e = new Example();
        }
        Example(){
                System.out.println("Example constructor – object is created");
        }
}
```

Case #9: What is the output after executing below class Sample?
Example class is loaded using Class.forName()

```
class Sample{
        static{
                System.out.println("Sample SB- class is loaded");
        }
        public static void main(String[] args) throws ClassNotFoundException{
                System.out.println("Sample main – execution is started");
                Class.forName("Example");
        }
}
```

Output – class is loaded, SV and SB are executed from Class.forName() statement. Also constructor is executed because of object creation in SB.

```
D:\NareshIT\HariKrishna\OCJP\NSM>javac Sample.java
D:\NareshIT\HariKrishna\OCJP\NSM>java Sample
Sample SB – class is loaded
Sample main – execution is started
Example SB – class is loaded
Example constructor – class object is created
```

Write a program to create object at the time of class loading and send that object to another class, for example create Example class object and sent it to Sample class when it is loaded.
Procedure

1. Write a Sample class with Example parameterized method as static method, say m1().
2. In Example class static block create its object and send that object to Sample class by calling that parameterized method m1()
3. Finally write Test class and load Example class by using any of the above three approaches, best suitable approach for this case is Class.forName()

Check below three programs

Test.java

```
class Test{
        public Static void main(String[] args) throws ClassNotFoundException{
                Class.forName("Example");
                Sample.m2("HariKrishna, NareshTechnologies");
        }
}
```

Example.java

```
class Example{
        static{
                System.out.println("Example SB – class is loaded");
                Example e = new Example();
                Sample.m1(e);
        }
        Example(){
                System.out.println("Example constructor -- object is created");
        }
        void print(String msg){
                System.out.println(msg);
        }
}
```

Sample.java

```
class Sample{
        static Example e1;
        static void m1(Example e){
                e1 = e;
        }
        static void m2(String msg){
                e1.print(msg);
        }
}
```

Output
Example SB – class is loaded
Example constructor – object is created
HariKrishna, Naresh *i* Technologies

Basically in the above program we have loaded Example class, we have created its object, and we *registered* that object to Sample class.

This code is very import to understand JDBC internal flow. In JDBC, Class.forName() is used to load Driver subclass and is register it with DriverManager class automatically. The driver class is registering with DriverManager class not because of Class.forName(), it is because of static block written inside Driver subclass as shown in Example class.

Rule: if we want to load class using Class.forName(), its .class file must be available, because in this case compiler will not compile Example.java file because compiler treat it as string not as class name. It is only known to JVM, if that class's .class file is not found by JVM, it throws RE: java.lang.ClassNotFoundExcepton.

Run below program by deleting Example.class, you will experience CNFE.

```
class Test{
        public Static void main(String[] args) throws ClassNotFoundException{
                Class.forName("Example");
                Sample.m2("HariKrishna, Naresh i Technologies");
        }
}
```

Write program to pass a class's current object to another class

Passing A class object from its non-static method to B class's non-static method to call A class non-static method.

```
//A.java
class A{
        void m1(){
                B b = new B();
                b.m2(this);
        }

        void print(String msg){
                Sopln(msg);
        }
}
```

```
//B.java
class B{

        void m2(A a){
                a.print(msg);

        }

}
```

```
//Test.java
class Test{
    public static
    void main(String[] args){
        A a = new A();
        a.m1();

    }
}
```

What is the output from the below program? This application throws NPE.

```
//A.java
class A{
        void m1(){
                B b = new B();
                b.m2(this);
        }

        void print(String msg){
                Sopln(msg);
        }
}
```

```
//B.java
class B{
        A a;
        void m2(A a){
                this.a = a;
        }

        void m3(String msg){
                a.print(msg);
        }
}
```

```
//Test.java
class Test{
    public static
    void main(String[] args){
        A a = new A();
        a.m1();

        B b = new B();
        b.m3("Hari Krishna ");
    }
}
```