

# Object Oriented Programming Concepts

**Sachchida Nand Chaurasia**  
Assistant Professor

Department of Computer Science  
Banaras Hindu University  
Varanasi

Email id: [snchaurasia@bhu.ac.in](mailto:snchaurasia@bhu.ac.in), [sachchidanand.mca07@gmail.com](mailto:sachchidanand.mca07@gmail.com)



December 15, 2020

CS 204

# Object Oriented Programming

L	T	P
3	0	2

## Course Objectives

- Learn the Java programming language: its syntax, idioms, patterns, and styles
- Become comfortable with object oriented programming:  
Learn to think in objects
- Learn the essentials of the Java class library, and learn how to learn about other parts of the library when you need them
- Introduce event driven Graphical User Interface (GUI) programming

## Detailed Syllabus I

- History of Java, Java Features, The three OOP principles: Encapsulation, Inheritance, and Polymorphism.
- **Data Types, Variables, and Arrays** : Data types, variables, constants, scope and life time of variables, operators, operator hierarchy, expressions, type conversion and casting, enumerated types, control flow-block scope, conditional statements, loops, break and continue statements, simple java program, arrays, input and output, formatting output

## Detailed Syllabus II

- **Review of OOP concepts:** constructors, methods, static fields and methods, access control, this reference, overloading methods and constructors, recursion, garbage collection, building strings, exploring string class, Enumerations, Generics
- **Inheritance:** Inheritance concept, benefits of inheritance, Super classes and Sub classes, Member access rules, Inheritance hierarchies, super uses, preventing inheritance: final classes and methods, casting, polymorphism- dynamic binding, method

## Detailed Syllabus III

overriding, abstract classes and methods, the Object class and its methods.

- **Interfaces:** Interfaces vs Abstract classes, defining an interface, implementing interfaces, accessing implementations through interface references, extending interface Inner classes –Uses of inner classes, local inner classes, anonymous inner classes, static inner classes, examples
- **Packages:** Defining, Creating and Accessing a Package, Understanding CLASSPATH, importing packages

## Detailed Syllabus IV

- **Data structures creation and manipulation in java:**

Introduction to Java Collections, Overview of Java Collection frame work, Commonly used Collection classes – ArrayList, LinkedList, HashSet, HashMap, TreeMap, Collection Interfaces – Collection, Set, List, Map, Legacy Collection classes – Vector, Hashtable, Stack, Dictionary(abstract), Enumeration interface, Iteration over Collections – Iterator interface, ListIterator interface Other Utility classes– StringTokenizer, Formatter, Random, Scanner, Observable, Using javaultil

## Detailed Syllabus V

- **Files streams:** byte streams, character streams, text Input/output, binary input/output, random access file operations, File management using File class, Using javaio
- **Exception handling:** Dealing with errors, benefits of exception handling, the classification of exceptions-exception hierarchy, checked exceptions and unchecked exceptions, usage of try, catch, throw, throws and finally, re-throwing exceptions, exception specification, built in exceptions, creating own exception sub classes, Guide lines for proper use of exceptions

## Detailed Syllabus VI

- **Multithreading:** Differences between multiple processes and multiple threads, thread states, creating threads, interrupting threads, thread priorities, synchronizing threads, inter-thread communication, thread groups, daemon threads
- **GUI Programming with Java:** The AWT class hierarchy, Introduction to Swing, Swing vs AWT,MVC architecture, Hierarchy for Swing components, Containers – Top-level containers- Light weight containers – Overview of several swing components- JButton, JToggleButton, JCheckBox, JRadioButton,

## Detailed Syllabus VII

JLabel, JTextField, JTextArea, JList, JComboBox,  
JMenu, Java's Graphics capabilities – Introduction,  
Graphics contexts and Graphics objects, color control,  
Font control, Drawing lines, rectangles and ovals,  
Drawing arcs, Layout management - Layout manager  
types – border, grid, flow, box

- **Event Handling** Events, Event sources, Event classes,  
Event Listeners, Relationship between Event sources  
and Listeners, Delegation event model, Semantic and  
Low-level events, Examples: handling a button click,  
handling mouse and keyboard events, Adapter classes

## Detailed Syllabus VIII

- **Applets:** Inheritance hierarchy for applets, differences between applets and applications, life cycle of an applet
  - Four methods of an applet, Developing applets and testing, passing parameters to applets, applet security issues

## Suggested Readings I

- ① C. Thomas Wu. *An Introduction to Object-Oriented Programming with Java.*  
McGraw-Hill Education, 5 edition, 2009
- ② Elappa Balagurusamy. *Programming with Java.*  
McGraw Hill Education, 6 edition, 2019
- ③ Bruce Eckel. *Thinking in Java.*  
Pearson, 2006
- ④ Herbert Schildt. *Java: The Complete Reference.*  
McGraw Hill Education, 11 edition, 2006

## Suggested Readings II

- ⑤ Paul J. Deitel Harvey Deitel. *Java 9 for Programmers.*  
Pearson, 4 edition, 2017



# CS204: Object Oriented Programming

## Concepts

**Sachchida Nand Chaurasia**

Assistant Professor

Department of Computer Science

Banaras Hindu University

Varanasi

Email id: [snchaurasia@bhu.ac.in](mailto:snchaurasia@bhu.ac.in), [sachchidanand.mca07@gmail.com](mailto:sachchidanand.mca07@gmail.com)



## The History and Evolution of Java I

To fully understand Java, one must understand the reasons behind its creation, the forces that shaped it, and the legacy that it inherits.

Like the successful computer languages that came before, Java is a blend of the best elements of its rich heritage combined with the innovative concepts required by its unique mission.

## The History and Evolution of Java II

### The Creation of Java

Java was created by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991.

- ▶ It took 18 months to develop the first working version.
- ▶ This language was initially called “Oak”, but was renamed “Java” in 1995.

## The History and Evolution of Java III

- ▶ Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of 1995, many more people contributed to the design and evolution of the language.
- ▶ Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lindholm were key contributors to the maturing of the original prototype.

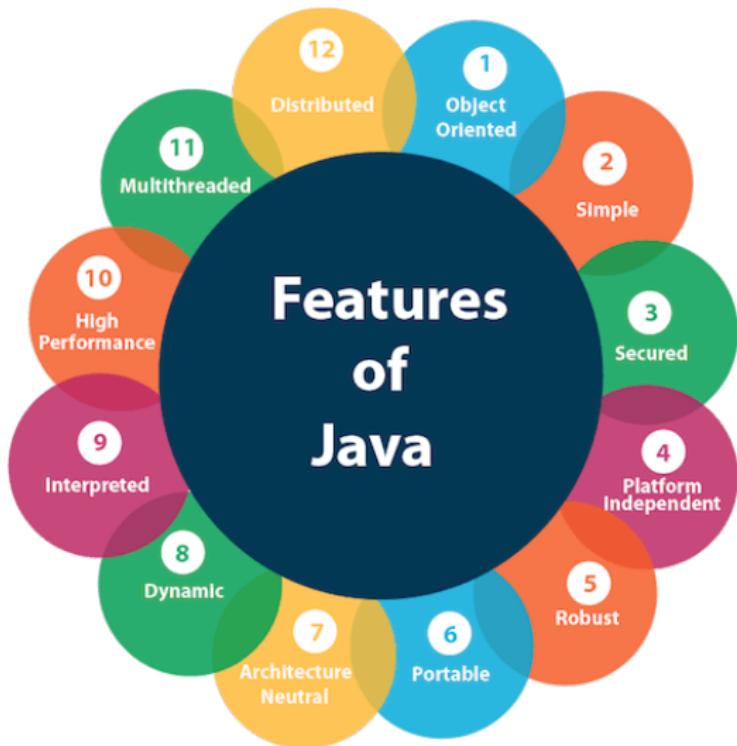
## The Java Buzzwords I

No discussion of Java's history is complete without a look at the Java buzzwords.

Although the fundamental forces that necessitated the invention of Java are **portability and security**, other factors also played an important role in molding the final form of the language.

The key considerations were summed up by the Java team in the following list of **buzzwords**:

## The Java Buzzwords II



## The Java Buzzwords III

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted and High performance

## The Java Buzzwords IV

- Distributed
- Dynamic

## The Java Buzzwords V

- **Simple** : Java was designed to be easy for the professional programmer to learn and use effectively.

## The Java Buzzwords VI

- ✓ Assuming that you have some programming experience, you will not find Java hard to master.
- ✓ If you already understand the basic concepts of object-oriented programming, learning Java will be even easier.
- ✓ Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort.

## The Java Buzzwords VII

- ✓ Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java.
- ✓ Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.

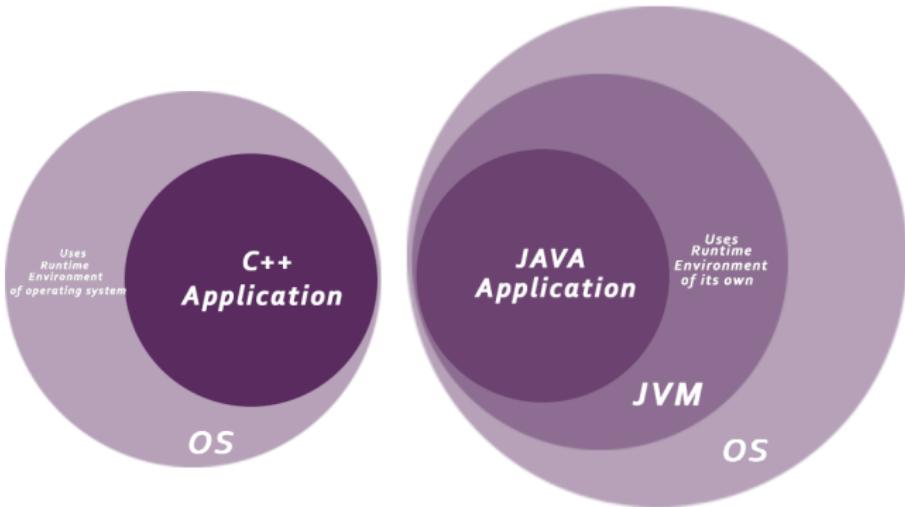
## The Java Buzzwords VIII

- ✓ There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

# The Java Buzzwords IX

## ■ **Secure :**

# The Java Buzzwords X



## The Java Buzzwords XI

- ✓ Java programs run inside a virtual machine which is known as a sandbox.
- ✓ Java does not support explicit pointer.
- ✓ Byte-code verifier checks the code fragments for illegal code that can violate access right to object.
- ✓ It provides Java.security package implements explicit security.
- ✓ It provides library level safety.

## The Java Buzzwords XII

- ✓ Run-time security check takes place when we load new code.
- ✓ Java provides some other features that make Java more secure.

## The Java Buzzwords XIII

## ■ Platform Independent :

# What is Platform?

## The Java Buzzwords XV

Any hardware or software environment in which a program runs, is known as a platform. Since Java has its own run-time environment Java Run-Time Environment (JRE) and API, it is called platform.

## The Java Buzzwords XVI

- ✓ “Platform Independence” is one of the core feature of Java.
- **Bytecode** is an intermediate code between the **source code** and **machine code**. It is a **low-level code** that is the result of the compilation of a source code which is written in a high-level language. It is processed by a virtual machine like Java Virtual Machine (JVM).

## The Java Buzzwords XVII

- Bytecode is a non-runnable code after it is translated by an interpreter into machine code then it is understandable by the machine.
- It is compiled to run on JVM, any system having JVM can run it irrespective of their operating system. That's why Java is platform-independent. Bytecode is referred to as a Portable code.

## The Java Buzzwords XVIII

```
for (int i = 2; i < 1000; i++) {  
    for (int j = 2; j < i; j++) {  
        if (i % j == 0)  
            continue outer;  
    }  
    System.out.println (i);  
}
```

## The Java Buzzwords XIX

```
0:  iconst_2
1:  istore_1
2:  iload_1
3:  sipush 1000
6:  if_icmpge    44
9:  iconst_2
10: istore_2
11: iload_2
12: iload_1
```

## The Java Buzzwords XX

```
13: if_icmpge      31
16: iload_1
17: iload_2
18: irem
19: ifne      25
22: goto      38
25: iinc      2,  1
28: goto      11
31: getstatic    //#84; Field
          java/lang/System.out:Ljava/io/PrintStream;
```

## The Java Buzzwords XXI

```
34: iload_1
35: invokevirtual //#85; Method
    java/io/PrintStream.println:(I)V
38: iinc   1, 1
41: goto   2
44: return
```

## The Java Buzzwords XXII

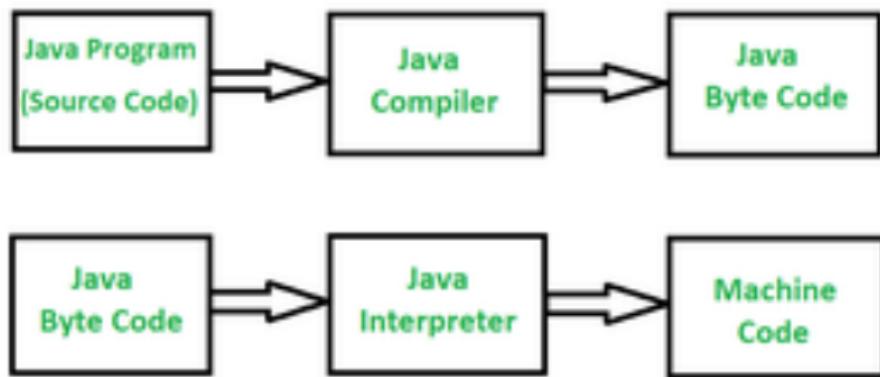
**Machine Code:** Machine code is a set of instructions that is directly machine-understandable and it is processed by the Central Processing Unit (CPU).

► Machine code is in binary (0's and 1's) format which is completely different from the byte code and source code. It is regarded as the most lowest-level representation of the source code.

## The Java Buzzwords XXIII

- Machine code is obtained after compilation or interpretation. It is also called machine language.

## The Java Buzzwords XXIV



**Figure 1:** Java source code is converted to Bytecode and then to machine code

## The Java Buzzwords XXV

### Difference between Byte Code and Machine Code:

	Bytecode	Machine Code
01.	Bytecode consisting of binary, hexadecimal, macro instructions like (new, add, swap, etc) and it is not directly understandable by the CPU. It is designed for efficient execution by software such as a virtual machine.intermediate-level	Machine code consisting of binary instructions that are directly understandable by the CPU.
02.	Bytecode is considered as the intermediate-level code.	Machine Code is considered as the low-level code.

## The Java Buzzwords XXVI

	Byte Code	Machine Code
03.	Bytecode is a non-runnable code generated after compilation of source code and it relies on an interpreter to get executed.	Machine code is a set of instructions in machine language or in binary format and it is directly executed by CPU.
04.	Bytecode is executed by the virtual machine then the Central Processing Unit.	Machine code is not executed by a virtual machine it is directly executed by CPU.
05.	Bytecode is less specific towards machine than the machine code.	Machine code is more specific towards machine than the byte code.

## The Java Buzzwords XXVII

	Byte Code	Machine Code
06.	<p>It is platform-independent as it is dependent on the virtual machine and the system having a virtual machine can be executed irrespective of the platform.</p>	<p>It is not platform independent because the object code of one platform can not be run on the same Operating System. Object varies depending upon system architecture and native instructions associated with the machine.</p>
07.	<p>All the source code need not be converted into byte code for execution by CPU. Some source code written by any specific high-level language is converted into byte code then byte code to object code for execution by CPU.</p>	<p>All the source code must be converted into machine code before it is executed by the CPU.</p>

# The Java Buzzwords XXVIII

## The Java Buzzwords XXIX

- ✓ Platform independent: Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into **platform independent bytecode**.
- ✓ This byte code is distributed over the web and interpreted by virtual Machine (JVM) on whichever platform it is being run.

## The Java Buzzwords XXX

- ✓ Platform independence in Java means you can “write once, run anywhere” which notifies that you can run Java code in any operating system that supports Java without recompilation.
  
- ✓ Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere(WORA).

## The Java Buzzwords XXXI

✓ The Java language typically compiles to a Virtual Machine a virtual CPU which runs all of the code that is written for the language. This enables the same executable binary to run on all systems that implement a Java Virtual machine (JVM). Java programs can be executed natively using a Java Processor. This isn't common and is mostly used for embedded systems.

## The Java Buzzwords XXXII

- ✓ Java code running in the JVM has access to OS-related services, like disk I/O and network access, if the appropriate privileges are granted. The JVM makes the system calls on behalf of the Java application.
- ✓ Currently, Java Standard Edition programs can run on Microsoft Windows , Mac OS X, several UNIX-Like operating systems, and several more non-UNIX-like operating systems like embedded systems. For mobile

## The Java Buzzwords XXXIII

applications, browser plugins are used for Windows and Mac based devices, and Android has built-in support for Java.

## The Java Buzzwords XXXIV

### ■ Portable :

What do you mean by portability?

## The Java Buzzwords XXXV

- ✓ Portability refers to the ability to run a program on different machines.
- ✓ Running a given program on different machines can require different amounts of work (for example, no work whatsoever, recompiling, or making small changes to the source code).

## The Java Buzzwords XXXVI

- ✓ Java is object-compatible. You compile it on one platform and the resultant class files can run on any JVM.

Note: **C is source-portable. You can take your C source code and compile it on any ISO C compiler, provided you follow the rules - that means no using undefined or implementation defined behaviour or any non-standard features.**

## The Java Buzzwords XXXVII

- Java provides three distinct types of portability:
  - ✓ **Source code portability:** A certain Java program must produce identical results, CPU, of the operating system or the underlying Java compiler.
  - ✓ **CPU architecture portability:** Current Java compilers produce object code (called byte-code) to a CPU that does not yet exist. For each real CPU in which the Java programs must run, a Java interpreter or virtual machine, executes the Java

## The Java Buzzwords XXXVIII

code. This nonexistent CPU allows the same object code to be executed in any CPU for which there is a Java interpreter.

- ✓ **OS / GUI:** Java solves this problem by providing a set of library functions (contained in libraries provided by Java, such as awt, util and lang) that converse with an imaginary operating system and an imaginary GUI. Just as the JVM presents a virtual CPU, the Java libraries have a virtual operating system / GUI.

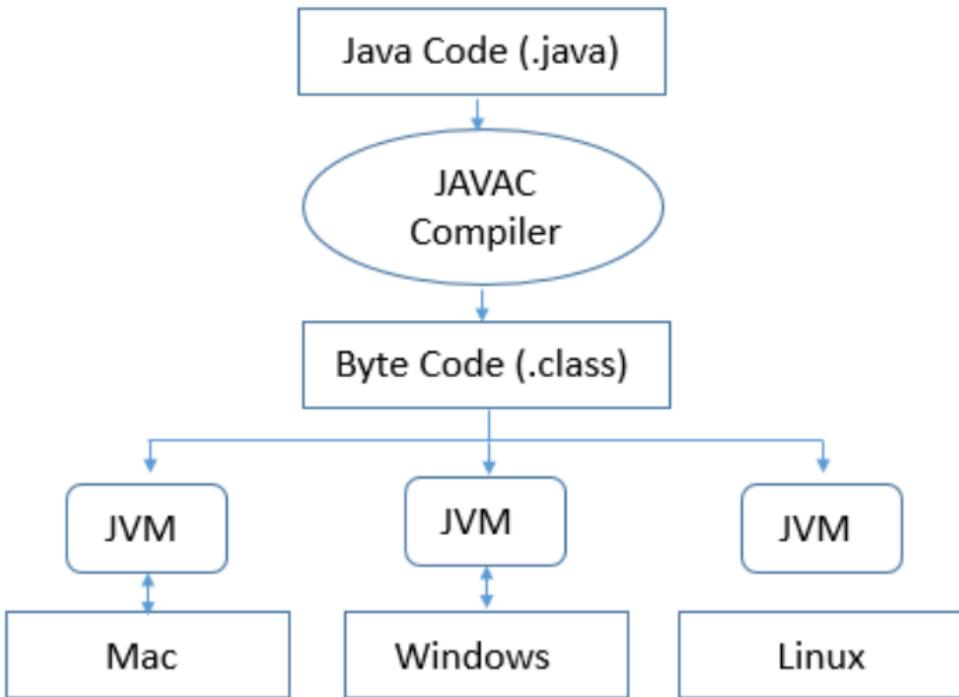
## The Java Buzzwords XXXIX

- Each Java implementation provides libraries that implement this virtual OS / GUI.
- ✓ Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it.
- ✓ If a Java program were to be run on virtually any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems.
- ✓ For example, in the case of an applet, the same applet must be able to be downloaded and executed by the wide variety of CPUs,

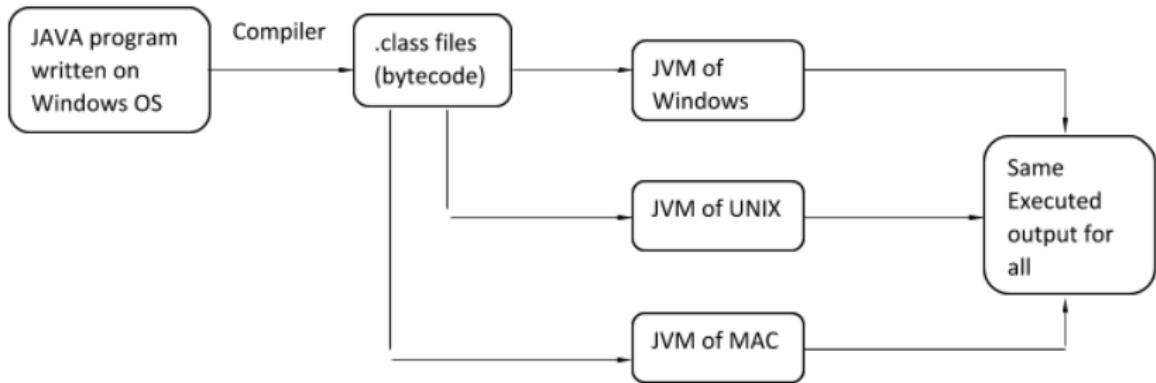
## The Java Buzzwords XL

- operating systems, and browsers connected to the Internet. ✓ It is not practical to have different versions of the applet for different computers.
- ✓ The same code must work on all computers. Therefore, some means of generating portable executable code was needed.

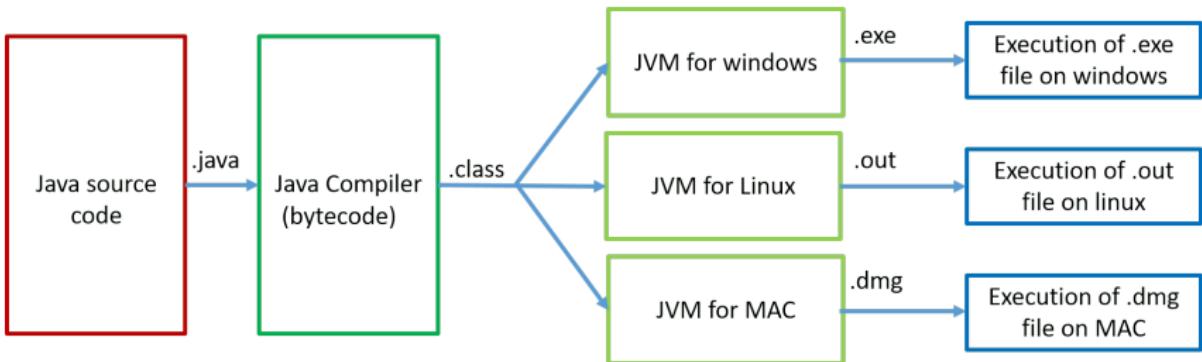
# The Java Buzzwords XLI



## The Java Buzzwords XLII



## The Java Buzzwords XLIII



**Figure 2:** Java platform neutral

### ■ Object-Oriented :

- ✓ Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language.
- ✓ This allowed the Java team the freedom to design with a blank slate.
- ✓ One outcome of this was a clean, usable, pragmatic approach to objects.

## The Java Buzzwords XLV

- ✓ Borrowing liberally from many seminal object-software environments of the last few decades, Java manages to strike a balance between the purist's "everything is an object" paradigm and the pragmatist's "stay out of my way" model.
- ✓ The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance non-objects.

## The Java Buzzwords XLVI

- ✓ Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior.
- ✓ Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules. These rules are:
  - Object
  - Class

# The Java Buzzwords XLVII

- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

## The Java Buzzwords XLVIII

### ■ Robust :

Java is Robust because it is highly supported language. It is portable across many Operating systems. Java also has feature of Automatic memory management and garbage collection. Strong type checking mechanism of Java also helps in making Java Robust.

✓ Robust programming is a style of programming that focuses on handling unexpected termination and

## The Java Buzzwords XLIX

unexpected actions. It requires code to handle these terminations and actions gracefully by displaying accurate and unambiguous error messages. These error messages allow the user to more easily debug the program.

**To better understand how Java is robust, consider two of the main reasons for program failure:** memory management mistakes and mishandled exceptional conditions (that is, run-time errors).

## The Java Buzzwords L

Memory management can be a difficult, tedious task in traditional programming environments.

For example, in C/C++, the programmer will often manually allocate and free all dynamic memory.

This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using.

Java virtually eliminates these problems by managing memory allocation and deallocation for you. (In fact, deallocation is

## The Java Buzzwords LI

completely automatic, because Java provides garbage collection for unused objects.)

Exceptional conditions in traditional environments often arise in situations such as division by zero or “file not found,” and they must be managed with clumsy and hard-to-read constructs. Java helps in this area by providing object oriented exception handling. In a

## The Java Buzzwords LII

well-written Java program, all run-time errors can—and should—be managed by your program.

```
if(var == true) {  
    ...  
} else {  
    ...  
}
```

## The Java Buzzwords LIII

```
if(var == true) {  
    ...  
} else if (var == false) {  
    ...  
}
```

## The Java Buzzwords LIV

```
if(var == true) {  
    ...  
} else if (var == false) {  
    ...  
} else {  
    ...  
}
```

- It uses strong memory management.
- There is a lack of pointers that avoids security problems.

## The Java Buzzwords LV

- There is automatic garbage collection in java which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- There are exception handling and the type checking mechanism in Java. All these points make Java robust.

## The Java Buzzwords LVI

### ■ **Multithreaded :**

- ✓ Java was designed to meet the real-world requirement of creating interactive, networked programs.
  
- ✓ To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously.

## The Java Buzzwords VII

- ✓ The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems.
- ✓ Java's easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

## The Java Buzzwords LVIII

### ■ **Architecture-neutral** :

A central issue for the Java designers was that of code longevity and portability.

- ✓ At the time of Java's creation, one of the main problems facing programmers was that no guarantee existed that if you wrote a program today, it would run tomorrow—even on the same machine.

## The Java Buzzwords LIX

- ✓ Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction.
- ✓ The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was “write once; run anywhere, any time, forever.” To a great extent, this goal was accomplished.

## The Java Buzzwords LX

- ✓ Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.
- ✓ **In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture.**

## The Java Buzzwords LXI

✓ However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

## The Java Buzzwords LXII

### ■ **Interpreted and High Performance:**

- ✓ Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode.
- ✓ This code can be executed on any system that implements the Java Virtual Machine. Most previous attempts at cross platform solutions have done so at the expense of performance.

## The Java Buzzwords LXIII

- ✓ The Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler.
- ✓ Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

## The Java Buzzwords LXIV

- ✓ Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++).
- ✓ Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

### ■ **Distributed :**

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file.

► Java also supports Remote Method Invocation (RMI). This feature enables a program to invoke methods across a network.

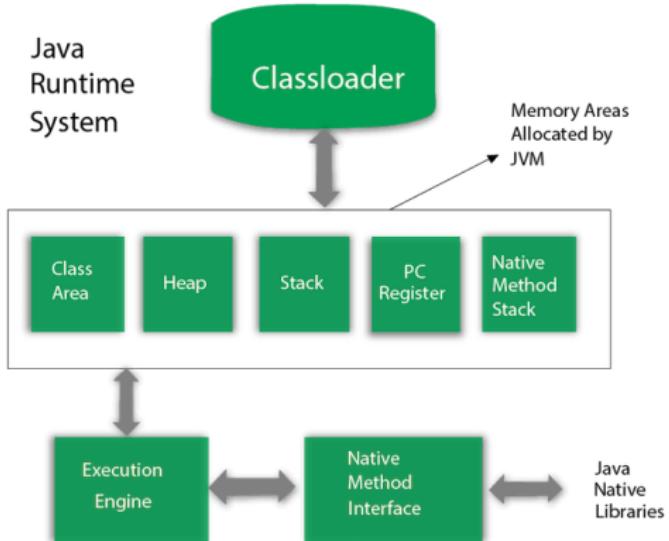
## The Java Buzzwords LXVI

■ **Dynamic:** Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

## Difference between JDK, JRE, and JVM I

# Java Virtual Machine

## Difference between JDK, JRE, and JVM II



**Figure 3:** JVM architecture

## Difference between JDK, JRE, and JVM III

- ✓ JVM (Java Virtual Machine) is an abstract machine.
- ✓ It is called a virtual machine because it doesn't physically exist.
- ✓ It is a specification that provides a runtime environment in which Java bytecode can be executed.
- ✓ It can also run those programs which are written in other languages and compiled to Java bytecode.

## Difference between JDK, JRE, and JVM IV

- ✓ JVMs are available for many hardware and software platforms.
- ✓ JVM, JRE, and JDK are **platform dependent** because the configuration of each OS is different from each other.  
However, Java is platform independent.
- ✓ There are three notions of the JVM: specification, implementation, and instance.

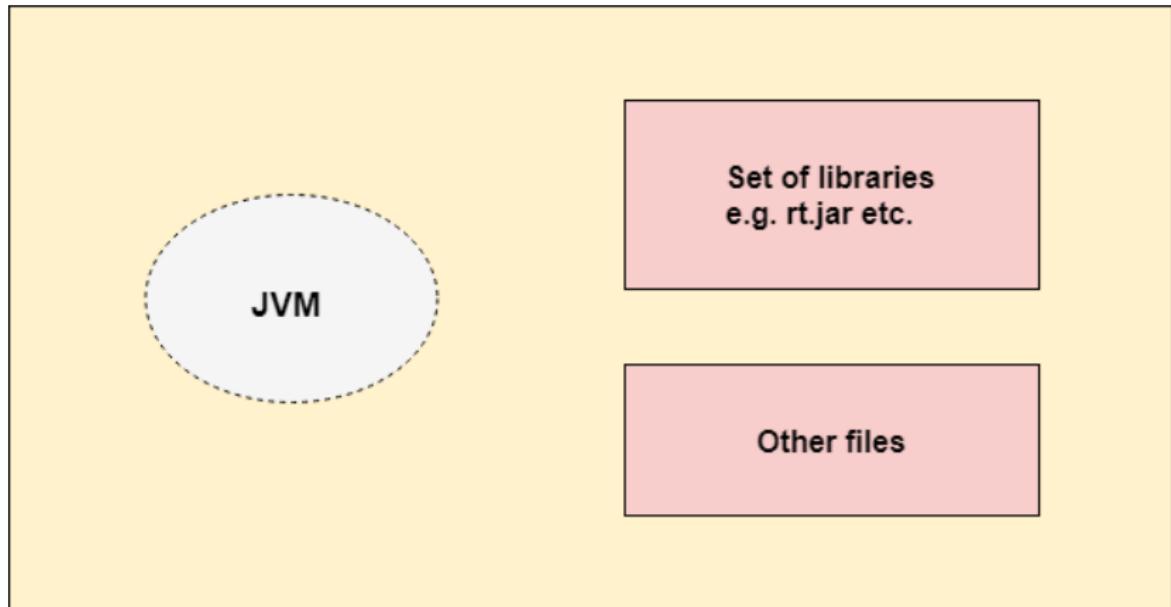
## Difference between JDK, JRE, and JVM V

The JVM performs the following main tasks:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

## Difference between JDK, JRE, and JVM VI

# Java Runtime Environment



JRE

## Difference between JDK, JRE, and JVM VII

- ✓ The Java Runtime Environment is a set of software tools which are used for developing Java applications.
- ✓ It is used to provide the runtime environment.
- ✓ It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

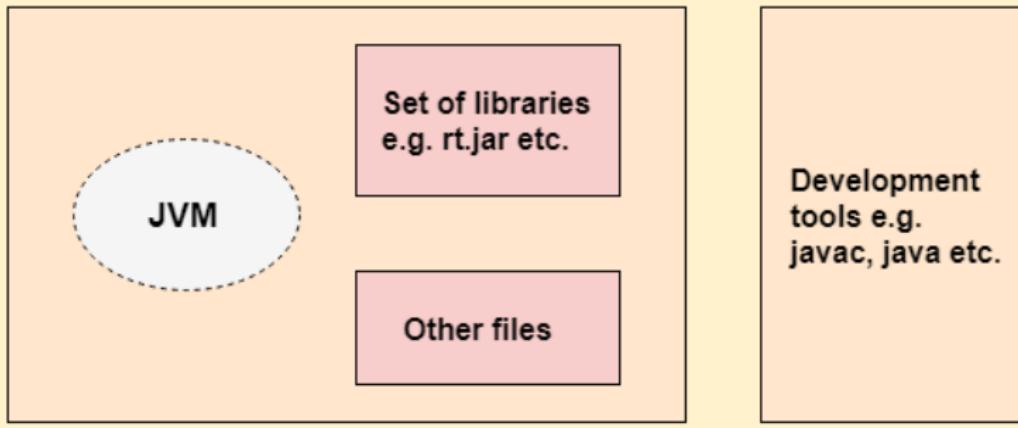
## **Difference between JDK, JRE, and JVM VIII**

- ✓ The implementation of JVM is also actively released by other companies besides Sun Micro Systems.

Difference between JDK, JRE, and JVM IX

# Java Development Kit

# Difference between JDK, JRE, and JVM X



JDK

## Difference between JDK, JRE, and JVM XI

- ✓ The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets.
- ✓ It physically exists. It contains JRE + development tools.
- ✓ JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:
  - Standard Edition Java Platform

## Difference between JDK, JRE, and JVM XII

- Enterprise Edition Java Platform
- Micro Edition Java Platform

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.

<b>Comparison Index</b>	<b>C++</b>	<b>Java</b>
Platform-independent	C++ is platform-dependent.	Java is platform-independent.
Mainly used for	C++ is mainly used for system programming.	Java is mainly used for application programming. It is widely used in window, web-based, enterprise and mobile applications.
Design Goal	C++ was designed for systems and applications programming. It was an extension of C programming language.	Java was designed and created as an interpreter for printing systems but later extended as a support network computing. It was designed with a goal of being easy to use and accessible to a broader audience.
Goto	C++ supports the goto statement.	Java doesn't support the goto statement.
Multiple inheritance	C++ supports multiple inheritance.	Java doesn't support multiple inheritance through class. It can be achieved by interfaces in Java.
Operator Overloading	C++ supports operator overloading.	Java doesn't support operator overloading.
Pointers	C++ supports pointers. You can write pointer program in C++.	Java supports pointer internally. However, you can't write the pointer program in Java. It means Java has restricted pointer support in Java.

Compiler and Interpreter	C++ uses compiler only. C++ is compiled and run using the compiler which converts source code into machine code so, C++ is platform dependent.	Java uses compiler and interpreter both. Java source code is converted into bytecode at compilation time. The interpreter executes this bytecode at runtime and produces output. Java is interpreted that is why it is platform independent.
Call by Value and Call by reference	C++ supports both call by value and call by reference.	Java supports call by value only. There is no call by reference in Java.
Structure and Union	C++ supports structures and unions.	Java doesn't support structures and unions.
Thread Support	C++ doesn't have built-in support for threads. It relies on third-party libraries for thread support.	Java has built-in thread support.
Documentation comment	C++ doesn't support documentation comment.	Java supports documentation comment (/** ... */) to create documentation for Java source code.
Virtual Keyword	C++ supports virtual keyword so that we can decide whether or not override a function.	Java has no virtual keyword. We can override all non-static methods by default. In other words, non-static methods are virtual by default.
unsigned right shift >>>	C++ doesn't support >>>operator.	Java supports unsigned right shift >>>operator that fills zero at the top for the negative numbers. For positive numbers, it works same like >>operator.

Inheritance Tree	C++ creates a new inheritance tree always.	Java uses a single inheritance tree always because all classes are the child of Object class in Java. The object class is the root of the inheritance tree in Java.
Hardware	C++ is nearer to hardware.	Java is not so interactive with hardware.
Object-oriented	C++ is an object-oriented language. However, in C language, single root hierarchy is not possible.	Java is also an object-oriented language. However, everything (except fundamental types) is an object in Java. It is a single root hierarchy as everything gets derived from Java.lang.Object.



## Package I

- A package is a **namespace** that organizes a set of related classes and interfaces.

**A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it. Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries.**

## Package II

- Conceptually you can think of packages as being similar to different folders on your computer.
- You might keep HTML pages in one folder, images in another, and scripts or applications in yet another. Because software written in the Java programming language can be composed of hundreds or thousands of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages.

## Package III

- The Java platform provides an enormous class library (a set of packages) suitable for use in your own applications. This library is known as the "Application Programming Interface", or "API" for short.
- Its packages represent the tasks most commonly associated with general-purpose programming. For example, a String object contains state and behavior for character strings; a File object allows a programmer to easily create, delete, inspect,

## Package IV

compare, or modify a file on the filesystem; a Socket object allows for the creation and use of network sockets; various GUI objects control buttons and checkboxes and anything else related to graphical user interfaces.

► There are literally thousands of classes to choose from. This allows you, the programmer, to focus on the design of your particular application, rather than the infrastructure required to make it work.

## Package V

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

# CS204: Object Oriented Programming Concepts

**Sachchida Nand Chaurasia**  
Assistant Professor

Department of Computer Science  
Banaras Hindu University  
Varanasi

Email id: [snchaurasia@bhu.ac.in](mailto:snchaurasia@bhu.ac.in), [sachchidanand.mca07@gmail.com](mailto:sachchidanand.mca07@gmail.com)



January 19, 2021

## Object I

## What is an OBJECT?

# Object II

## Objects: Real World Examples

Pencil



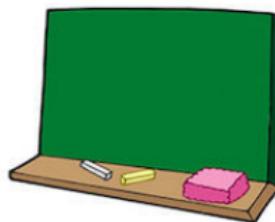
Apple



Book



Bag



Board

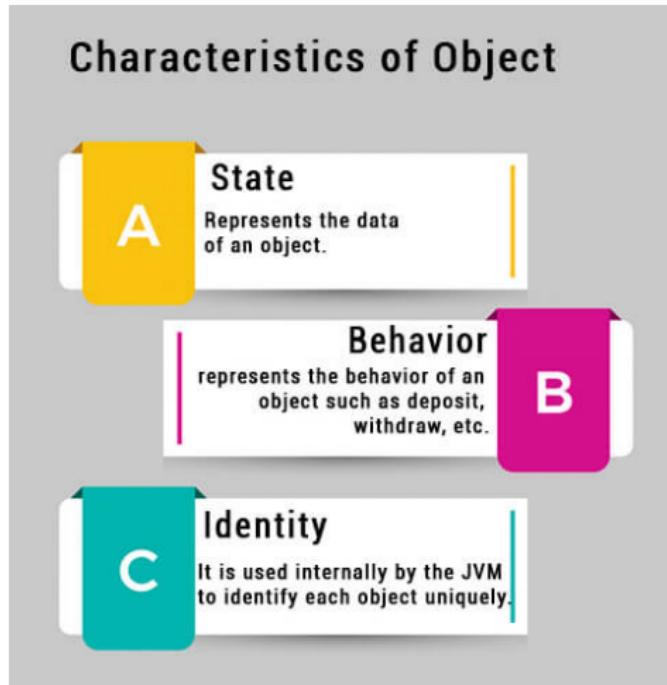
**Figure 1:** Objects

## Object III

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

# Object IV



**Figure 2:** Objects

## Object V

For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

## Object in the context of Java

### What Is an Object in the context of Java?

- ▶ Objects are key to understanding object-oriented technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.
- ▶ Real-world objects share two characteristics: They all have state and behavior. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

## Object in the context of Java

What Is an Object in the context of Java?

► Objects are key to understanding object-oriented technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.

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

## Object in the context of Java

What Is an Object in the context of Java?

- Objects are key to understanding object-oriented technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.
- Real-world objects share two characteristics: They all have state and behavior. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

## Object Oriented Programming Concepts

- Take a minute right now to observe the real-world objects that are in your immediate area. For each object that you see, ask yourself two questions: "What possible states can this object be in?" and "What possible behavior can this object perform?".

For example: your desktop lamp may have only two possible states (on and off) and two possible behaviors (turn on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and behavior (turn on, turn off, increase volume, decrease volume, seek, scan, and tune). These real-world observations all translate into the world of object-oriented programming.

## Object Oriented Programming Concepts

- Take a minute right now to observe the real-world objects that are in your immediate area. For each object that you see, ask yourself two questions: "What possible states can this object be in?" and "What possible behavior can this object perform?".
- For Example: your desktop lamp may have only two possible states (on and off) and two possible behaviors (turn on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and behavior (turn on, turn off, increase volume, decrease volume, seek, scan, and tune). These real-world observations all translate into the world of object-oriented programming.

# Object Oriented Programming Concepts I

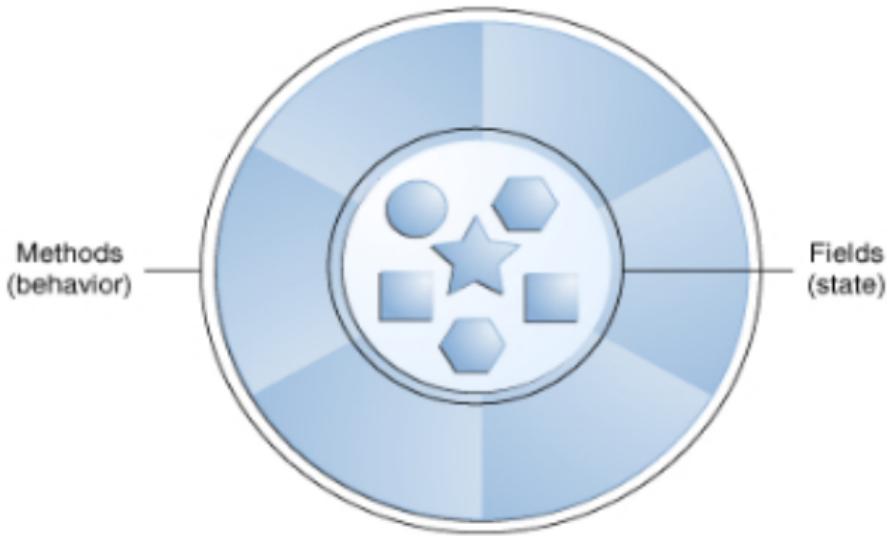
**Object**- Unique programming entity that has *methods* has *attributes* and can react to events.

**Method** - Things which an object can do; the "verbs" of objects. In code, usually can be identified by an "action" word.

**Attribute**- Things which describe an object; the "adjectives" of objects. In code, usually can be identified by a "descriptive" word- *enable*, *BackColor*.

**Events**- Forces external to an object to which that object can react. In code, usually attached to an event procedure.

## Object Oriented Programming Concepts II

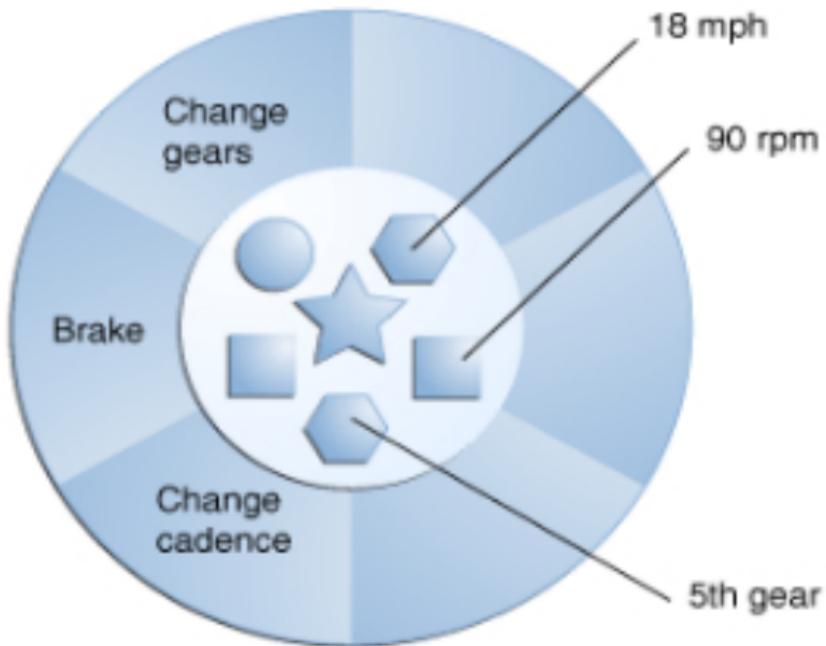


**Figure 3:** A software object

## Object Oriented Programming Concepts III

- Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in fields (variables in some programming languages) and exposes its behavior through methods (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication. Hiding internal state and requiring all interaction to be performed through an object's methods is known as data encapsulation — a fundamental principle of object-oriented programming.  
Consider a bicycle, for example:

## Object Oriented Programming Concepts IV



**Figure 4:** A bicycle modeled as a software object

## Object Oriented Programming Concepts V

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

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

- ① **Modularity:** The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.

## Object Oriented Programming Concepts VI

- ② Information-hiding: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
- ③ Code re-use: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.

## Object Oriented Programming Concepts VII

- ④ Pluggability and debugging ease: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace it, not the entire machine.

## Class I

In the real world, you'll often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components.

- In object-oriented terms, we say that your bicycle is an *instance* of the *class of objects* known as bicycles.
- A class is the blueprint from which individual objects are created.

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

## Class II

```
class Bicycle {  
    int cadence = 0;  
    int speed = 0;  
    int gear = 1;  
    void changeCadence(int newValue) {  
        cadence = newValue;  
    }  
    void changeGear(int newValue) {  
        gear = newValue;  
    }  
    void speedUp(int increment) {  
        speed = speed + increment;  
    }  
}
```

## Class III

```
void applyBrakes(int decrement) {  
    speed = speed - decrement;  
}  
void printStates() {  
    System.out.println("cadence:" +  
cadence + " speed:" +  
speed + " gear:" + gear);  
}  
}
```

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

```
class BicycleDemo
```

## Class IV

```
public static void main(String[] args) {  
  
    // Create two different  
    // Bicycle objects  
    Bicycle bike1 = new Bicycle();  
    Bicycle bike2 = new Bicycle();  
  
    // Invoke methods on  
    // those objects  
    bike1.changeCadence(50);  
    bike1.speedUp(10);  
    bike1.changeGear(2);  
    bike1.printStates();
```

## Class V

```
bike2.changeCadence(50);
bike2.speedUp(10);
bike2.changeGear(2);
bike2.changeCadence(40);
bike2.speedUp(10);
bike2.changeGear(3);
bike2.printStates();
}
}
```

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

## Class VI

cadence:50 speed:10 gear:2

cadence:40 speed:20 gear:3

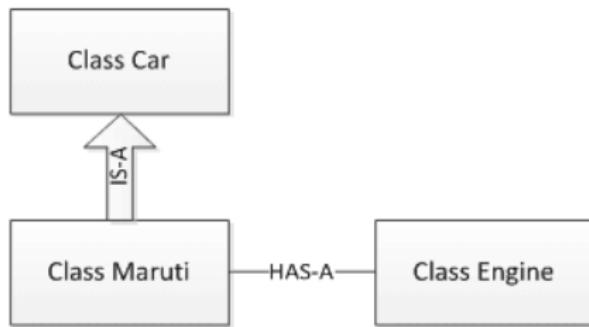
## Inheritance I

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.
- Inheritance represents the IS-A relationship which is also known as a parent-child relationship.

## Inheritance II

In object-oriented programming, the concept of IS-A is a totally based on Inheritance. It is just like saying "A is a B type of thing". For example, Apple is a Fruit, Car is a Vehicle etc. Inheritance is uni-directional. For example, House is a Building. But Building is not a House.



**Figure 5:** IS-A relationship

## Inheritance III

### Why use inheritance in Java:

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

### Terms used in Inheritance:

- ① Class: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- ② Sub Class/Child Class: Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.

## Inheritance IV

- ③ Super Class/Parent Class: Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- ④ Reusability: As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

## Inheritance V

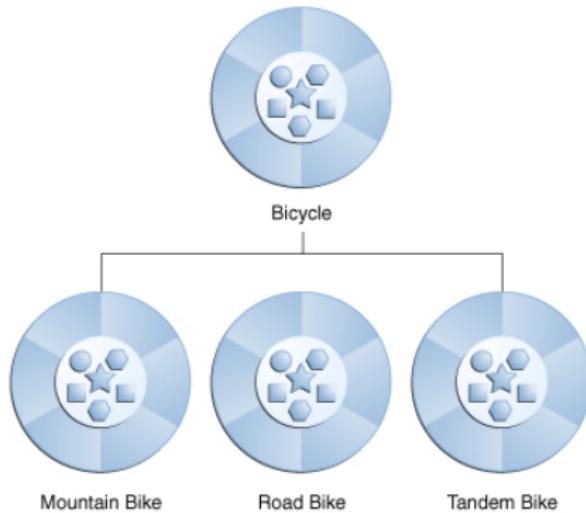
- Different kinds of objects often have a certain amount in common with each other. For example: Mountain bikes, road bikes, and tandem bikes, all share the characteristics of bicycles (current speed, current pedal cadence, current gear).
- Yet each also defines additional features that make them different: tandem bicycles have two seats and two sets of handlebars; road bikes have drop handlebars; some mountain bikes have an additional chain ring, giving them a lower gear ratio.
- Object-oriented programming allows classes to inherit commonly used state and behavior from other classes. In this

## Inheritance VI

example, *Bicycle* now becomes the superclass of *MountainBike*, *RoadBike*, and *TandemBike*.

- In the Java programming language, **each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of *subclasses*.**

## Inheritance VII



**Figure 6:** A hierarchy of bicycle classes

## Inheritance VIII

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

```
class MountainBike extends Bicycle {  
  
    // new fields and methods defining  
    // a mountain bike would go here  
  
}
```

## Inheritance IX

This gives MountainBike all the same fields and methods as Bicycle, **yet allows its code to focus exclusively on the features that make it unique**. This makes code for your subclasses easy to read.

- However, you must take care to properly document the state and behavior that each superclass defines, since that code will not appear in the source file of each subclass.

## Inheritance X

```
public class Bicycle {  
    // the Bicycle class has three fields  
    public int cadence;  
    public int gear;  
    public int speed;  
    // the Bicycle class has one constructor  
    public Bicycle(int startCadence, int startSpeed,  
        int startGear) {  
        gear = startGear;  
        cadence = startCadence;  
        speed = startSpeed;  
    }  
    // the Bicycle class has four methods
```

## Inheritance XI

```
public void setCadence(int newValue) {  
    cadence = newValue;  
}  
public void setGear(int newValue) {  
    gear = newValue;  
}  
public void applyBrake(int decrement) {  
    speed -= decrement;  
}  
public void speedUp(int increment) {  
    speed += increment;  
}  
}
```

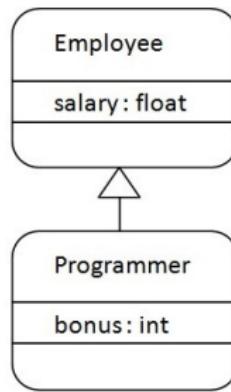
## Inheritance XII

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

```
public class MountainBike extends Bicycle {  
    // the MountainBike subclass adds one field  
    public int seatHeight;  
    // the MountainBike subclass adds one method  
    public void setHeight(int newValue) {  
        seatHeight = newValue;  
    }  
}
```

## Inheritance XIII

- MountainBike inherits all the fields and methods of Bicycle and adds the field seatHeight and a method to set it.
- Another example:**



**Figure 7:** Inheritance

## Inheritance XIV

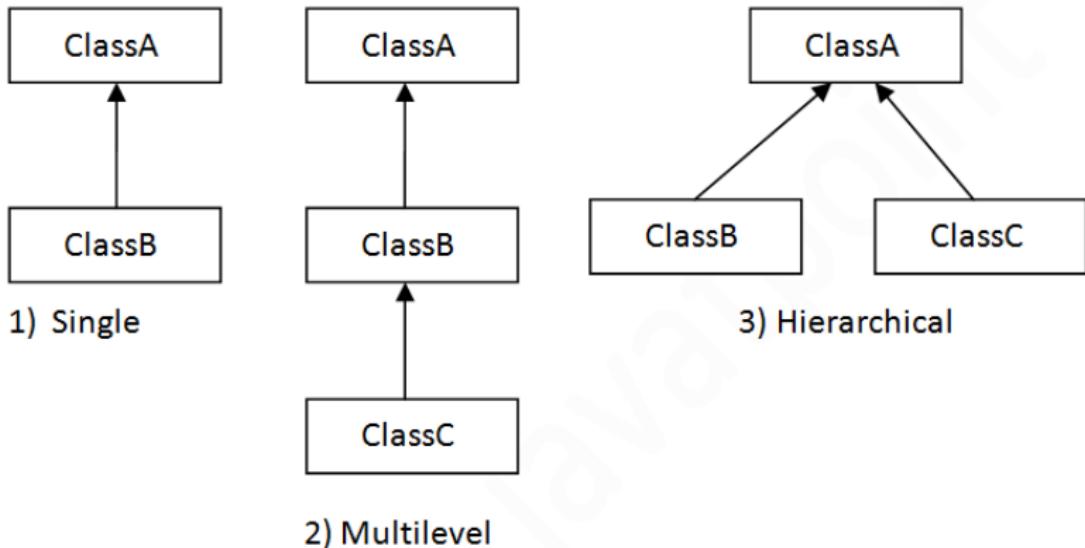
```
class Employee{  
    float salary=40000;  
}  
class Programmer extends Employee{  
    int bonus=10000;  
    public static void main(String args[]){  
        Programmer p=new Programmer();  
        Programmer p1=new Programmer();  
        p1.bonus=12000;  
        System.out.println("Programmer salary is:"+p.salary);  
        System.out.println("Bonus of Programmer is:"+p.bonus);  
  
        System.out.println("Programmer salary is:"+p1.salary);  
    }  
}
```

## Inheritance XV

```
System.out.println("Bonus of Programmer is:"+p1.bonus);  
}  
}
```

Output: ????

# Inheritance XVI



**Figure 8:** Types of inheritance in Java

## Inheritance XVII

### ► Single inheritance Example:

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
}
```

## Inheritance XVIII

Output: ????

## Inheritance XIX

### ► Multilevel Inheritance Example:

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
}
```

## Inheritance XX

```
d.bark();  
d.eat();  
}}
```

Output: ????

## Inheritance XXI

### ► Hierarchical Inheritance Example:

```
class Animal{
void eat(){
System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
```

## Inheritance XXII

```
c.meow();  
c.eat();  
//c.bark();//C.T.Error  
}}
```

Output: ????

## Inheritance

### Q) Why multiple inheritance is not supported in Java?

> To reduce the complexity and simplify the language, multiple inheritance is not supported in Java.

The main reason behind this is that it creates ambiguity in the language. If you inherit from two classes, then which method should be used? In other words, if there are two methods with same name and same parameters in both the classes, then which one should be used? This is called diamond problem.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

## Inheritance

### Q) Why multiple inheritance is not supported in Java?

- To reduce the complexity and simplify the language, multiple inheritance is not supported in Java.

Java does not support multiple inheritance because it can lead to complex class hierarchies and ambiguous method resolution. In multiple inheritance, a class can inherit from two or more classes, which can result in conflicts if the parent classes have methods with the same name or parameters. This can make it difficult to predict the behavior of the code at runtime, leading to bugs and errors.

Java's design philosophy emphasizes simplicity and readability. By not supporting multiple inheritance, Java makes it easier for developers to write clean and maintainable code. Instead, Java provides other mechanisms like interfaces and composition to achieve similar functionality in a more controlled and predictable way.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

## Inheritance

### Q) Why multiple inheritance is not supported in Java?

- To reduce the complexity and simplify the language, multiple inheritance is not supported in Java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

For example, if you inherit both A and B classes, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

### Q) Why multiple inheritance is not supported in Java?

► To reduce the complexity and simplify the language, multiple inheritance is not supported in Java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were
public static void main(String args[]){
    C obj=new C();
    obj.msg(); //Now which msg() method would be invoked?
}
}
```

## Polymorphism I

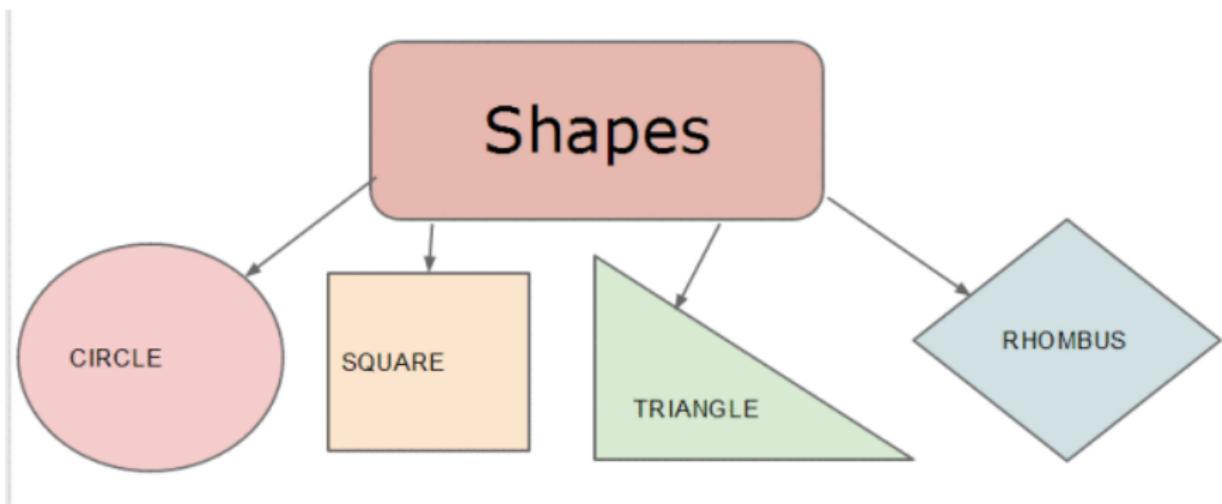
Polymorphism is considered one of the important features of Object-Oriented Programming.

- Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations.
- The word “poly” means many and “morphs” means forms, So it means many forms.

**More generally, the concept of polymorphism is often expressed by the phrase “one interface, multiple methods.”**

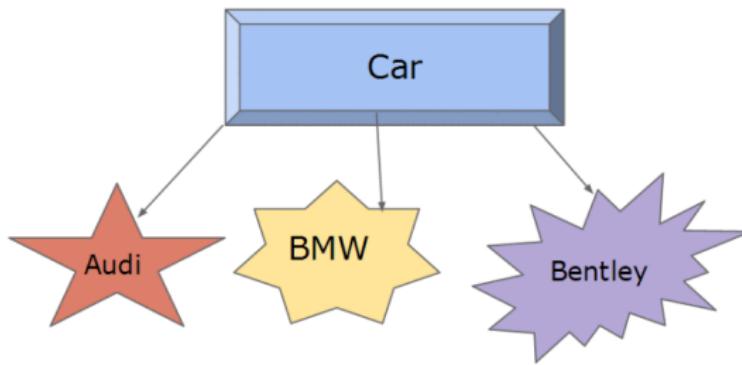
## Polymorphism II

For example, Shapes is a class, and square, circles are all shapes which they can acquire. So a method written in shapes can be used by its forms like this:



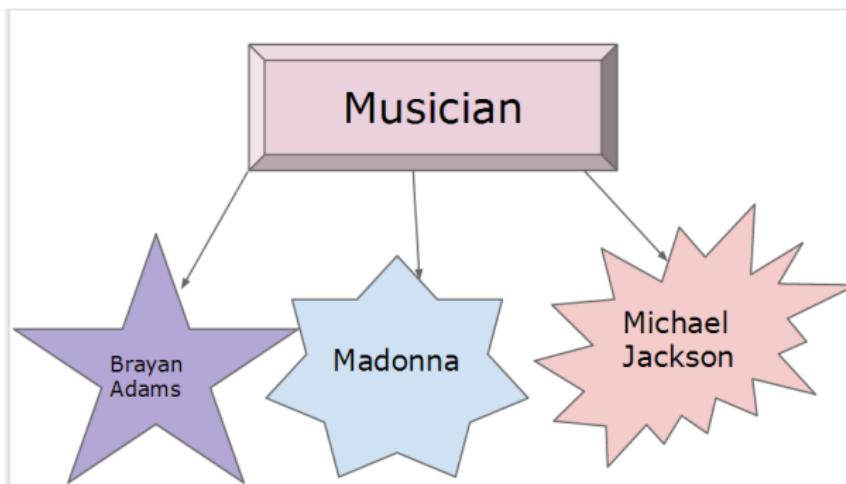
## Polymorphism III

Cars can be of different types- Racing cars, 2 seater cars, 6 seater cars, 4 seater cars. Also can be classified on the basis of doors like cars having 2 doors, 4 doors, 6 doors, etc. It can also be categorized on the basis of brands like BMW, Audi, Merc, Maruti, etc.



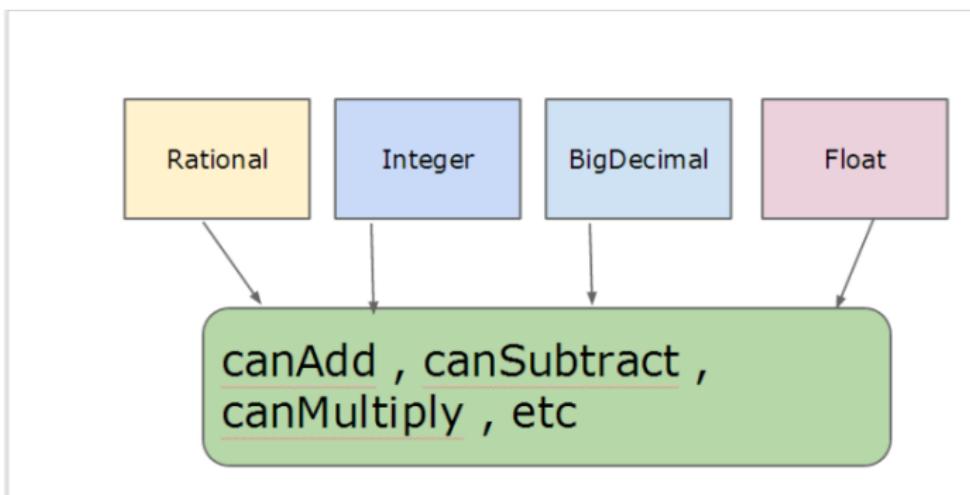
## Polymorphism IV

Musician – Different forms of musicians are there who sings Hollywood songs, Metal, Blues, Soft music, Rock Music, etc. So we can divide on the basis of singers-genres.



## Polymorphism V

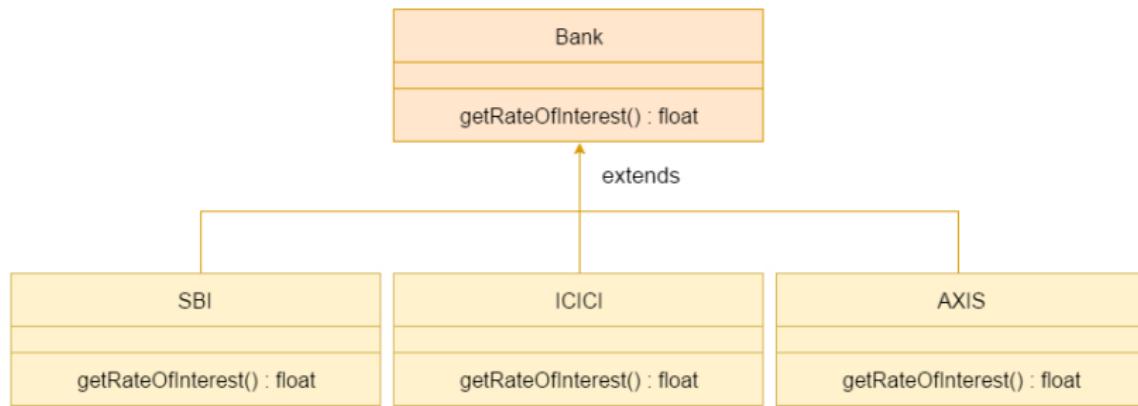
Numbers- Mathematics numbers are of different types like Rational, Integer, Big decimal, Float, etc. And they further can be used to apply other mathematical operations.



## Polymorphism VI

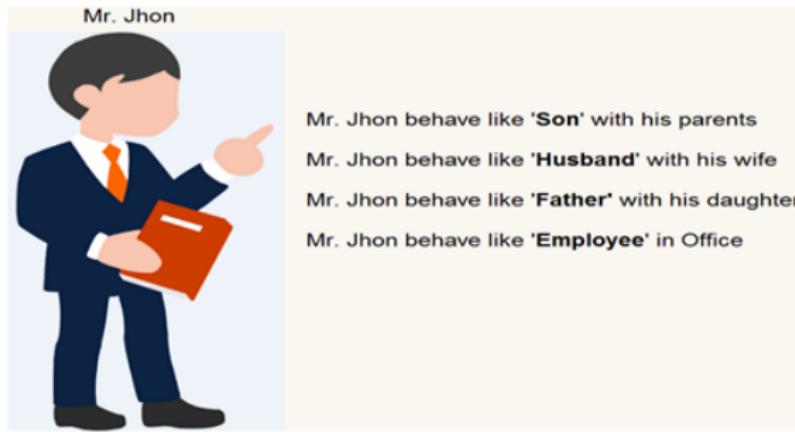
Consider a scenario where Bank is a class that provides a method to get the rate of interest. However, the rate of interest may differ according to banks.

For example, SBI, ICICI, and AXIS banks are providing 8.4%, 7.3%, and 9.7% rate of interest.



## Polymorphism VII

Another Real life example of polymorphism: A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism.



## Polymorphism VIII

Polymorphism example in the context of Java:

```
// Java program for Method overloading
class MultiplyFun {
    // Method with 2 parameter
    static int Multiply(int a, int b){
        return a * b;
    }
    // Method with the same name but 3 parameter
    static int Multiply(int a, int b, int c){
        return a * b * c;
    }
}
class Main {
    public static void main(String[] args)
```

## Polymorphism IX

```
{  
    System.out.println(MultiplyFun.Multiply(2, 4));  
    System.out.println(MultiplyFun.Multiply(2, 7, 3));  
}  
}
```

## Polymorphism X

### Advantage of polymorphism:

- ✓ It helps the programmer to reuse the codes, i.e., classes once written, tested and implemented can be reused as required.
- ✓ Saves a lot of time.
- ✓ Single variable can be used to store multiple data types.
- ✓ Easy to debug the codes.

## Data Abstraction I

- Abstraction is a process of hiding the implementation details and showing only functionality to the user.
- Another way, it shows only essential things to the user and hides the internal details.
- For example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

## Data Abstraction II

A car is viewed as a car rather than its individual components.



Registration
<ul style="list-style-type: none"><li>• Vehicle Identification Number</li><li>• License plate</li><li>• Current Owner</li><li>• Tax due, date</li></ul>

Owner
<ul style="list-style-type: none"><li>• Car Description</li><li>• Service History</li><li>• Petrol Mileage History</li></ul>

Garage
<ul style="list-style-type: none"><li>• License plate</li><li>• Work Description</li><li>• Billing Info</li><li>• Owner</li></ul>

## Data Abstraction III

- In the figure, you can see that an Owner is interested in details like Car description, service history, etc.
- Garage Personnel are interested in details like License, work description, bill, owner, etc.
- Registration Office interested in details like vehicle identification number, current owner, license plate, etc. It means each application identifies the details that are important to it.
- Abstraction can be seen as the technique of filtering out the unnecessary details of an object so that there remain only the useful characteristics that define it. Abstraction focuses on

## Data Abstraction IV

the perceived behavior of the entity. It provides an external view of the entity.

► Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing the accelerator the speed is actually increasing, he does not know about the inner

## Data Abstraction V

mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

In java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.

### ► **Abstract classes and Abstract methods :**

- ① An abstract class is a class that is declared with abstract keyword.
- ② An abstract method is a method that is declared without an implementation.
- ③ An abstract class may or may not have all abstract methods. Some of them can be concrete methods

## Data Abstraction VI

- ④ A method defined abstract must always be redefined in the subclass, thus making overriding compulsory OR either make subclass itself abstract.
- ⑤ Any class that contains one or more abstract methods must also be declared with abstract keyword.
- ⑥ There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the new operator.
- ⑦ An abstract class can have parametrized constructors and default constructor is always present in an abstract class.

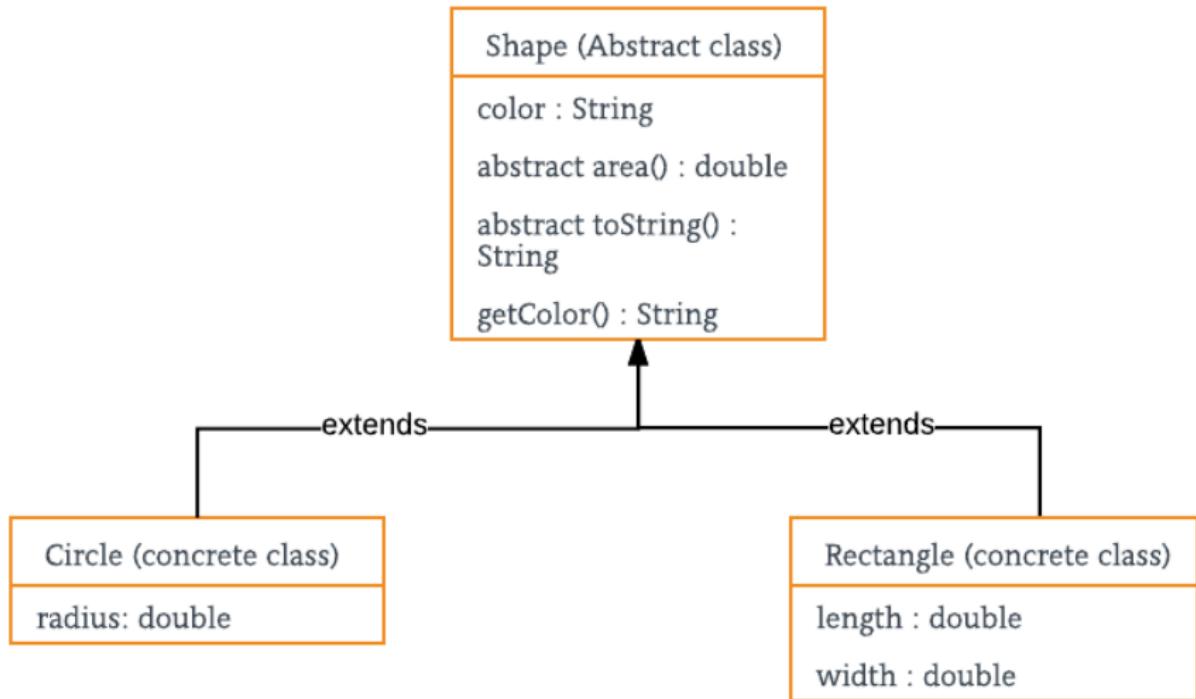
### ► When to use abstract classes and abstract methods

There are situations in which we will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes we will want to create a superclass that only defines a generalization form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.

## Data Abstraction VIII

Consider a classic “shape” example, perhaps used in a computer-aided design system or game simulation. The base type is “shape” and each shape has a color, size and so on. From this, specific types of shapes are derived(inherited)-circle, square, triangle and so on – each of which may have additional characteristics and behaviors. For example, certain shapes can be flipped. Some behaviors may be different, such as when you want to calculate the area of a shape. The type hierarchy embodies both the similarities and differences between the shapes.

# Data Abstraction IX



**Figure 9:** Abstract Class and Methods

## Data Abstraction X

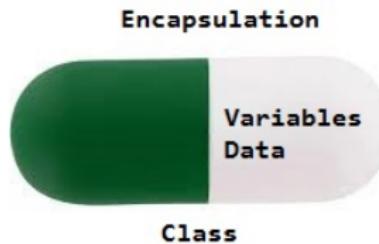
### ► Advantages of Abstraction:

- It reduces the complexity of viewing the things.
- Avoids code duplication and increases reusability.
- Helps to increase security of an application or program as only important details are provided to the user.

## Encapsulation I

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates.

► Other way to think about encapsulation is, it is a protective shield that prevents the data from being accessed by the code outside this shield.



## Encapsulation II

- Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared.
- As in encapsulation, the data in a class is hidden from other classes using the data hiding concept which is achieved by making the members or methods of class as private and the class is exposed to the end user or the world without providing any details behind implementation using the abstraction concept, so it is also known as combination of data-hiding and abstraction.

## Encapsulation III

- Encapsulation can be achieved by: Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.

### Advantages of Encapsulation:

- Data Hiding: The user will have no idea about the inner implementation of the class. It will not be visible to the user that how the class is storing values in the variables. He only knows that we are passing the values to a setter method and variables are getting initialized with that value.

## Encapsulation IV

- Increased Flexibility: We can make the variables of the class as read-only or write-only depending on our requirement. If we wish to make the variables as read-only then we have to omit the setter methods. If we wish to make the variables as write-only then we have to omit the get methods.
- Reusability: Encapsulation also improves the re-usability and easy to change with new requirements. Testing code is easy: Encapsulated code is easy to test for unit testing.

## Encapsulation V

### Example of encapsulation:

```
class Employee{  
    private int Emp_id; //Data hiding  
    public void SetEmpId(int Emp_id1){  
        Emp_id = Emp_id1;  
    }  
    public int GetEmpId(){  
        return Emp_id;  
    }  
}
```

## Encapsulation VI

```
/* File name : EncapTest.java */
public class EncapTest {
    private String name;
    private String idNum;
    private int age;
    public int getAge() {
        return age;
    }
    public String getName() {
        return name;
    }
    public String getIdNum() {
        return idNum;
    }
}
```

## Encapsulation VII

```
}

public void setAge( int newAge) {
    age = newAge;
}

public void setName(String newName) {
    name = newName;
}

public void setIdNum( String newId) {
    idNum = newId;
}

/*
File name : RunEncap.java */
public class RunEncap {
```

## Encapsulation VIII

```
public static void main(String args[]) {  
    EncapTest encap = new EncapTest();  
    encap.setName("James");  
    encap.setAge(20);  
    encap.setIdNum("12343ms");  
    System.out.print("Name : " + encap.getName() + "  
                    Age : " + encap.getAge());  
}  
}
```

Name : James Age : 20

## Encapsulation vs Data Abstraction I

S.NO	ABSTRACTION	ENCAPSULATION
1.	Abstraction is the process or method of gaining the information.	While encapsulation is the process or method to contain the information.
2.	In abstraction, problems are solved at the design or interface level.	While in encapsulation, problems are solved at the implementation level.

## Encapsulation vs Data Abstraction II

3.	<p>Abstraction is the method of hiding the unwanted information.</p>	<p>Whereas encapsulation is a method to hide the data in a single entity or unit along with a method to protect information from outside.</p>
4.	<p>We can implement abstraction using abstract class and interfaces.</p>	<p>Whereas encapsulation can be implemented using access modifier i.e. private, protected and public.</p>

## Encapsulation vs Data Abstraction III

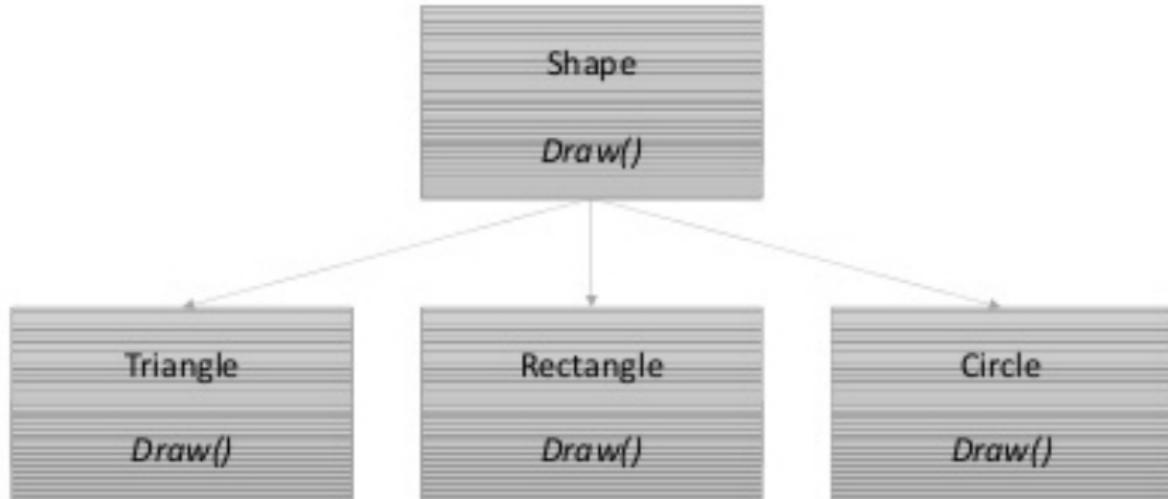
5.	In abstraction, implementation complexities are hidden using abstract classes and interfaces.	While in encapsulation, the data is hidden using methods of getters and setters.
6.	The objects that help to perform abstraction are encapsulated.	Whereas the objects that result in encapsulation need not be abstracted.

# Encapsulation vs Data Abstraction IV

## Polymorphism, Encapsulation, and Inheritance Work Together I

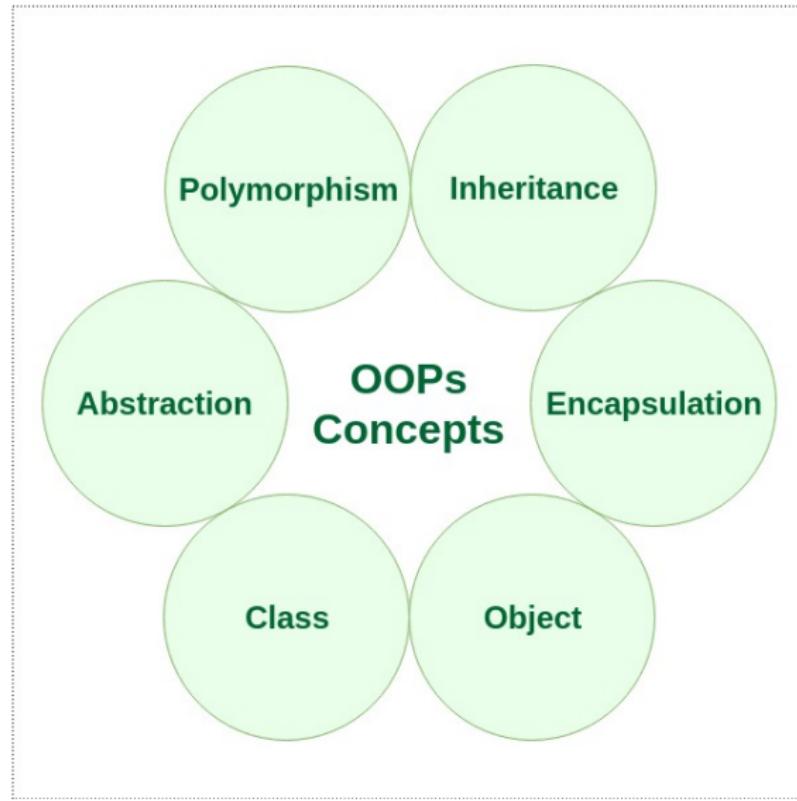
- When properly applied, polymorphism, encapsulation, and inheritance combine to produce a programming environment that supports the development of far more robust and scaleable programs than does the process-oriented model.
- A well-designed hierarchy of classes is the basis for reusing the code in which you have invested time and effort developing and testing.
- Encapsulation allows you to migrate your implementations over time without breaking the code that depends on the public interface of your classes.
- Polymorphism allows you to create clean, sensible, readable, and resilient code.

## Polymorphism, Encapsulation, and Inheritance Work Together II



**Figure 10:** Polymorphism, Encapsulation, and Inheritance work together

# Polymorphism, Encapsulation, and Inheritance Work Together III



**Figure 11:** Object-Oriented-Programming-Concepts

# CS204:Object Oriented Programming

## Concepts

**Sachchida Nand Chaurasia**  
Assistant Professor

Department of Computer Science  
Banaras Hindu University  
Varanasi

Email id: [snchaurasia@bhu.ac.in](mailto:snchaurasia@bhu.ac.in), [sachchidanand.mca07@gmail.com](mailto:sachchidanand.mca07@gmail.com)



January 27, 2021

# Object Oriented Programming Concepts I

## large First Simple program in Java

```
1 public class Simple
2 {
3     public static void main (String args [])
4     {
5         System.out.println("Hello Java");
6     }
7 }
```

To setup Javac path:

<https://www.Javatpoint.com/how-to-set-path-in-Java>

## Object Oriented Programming Concepts II

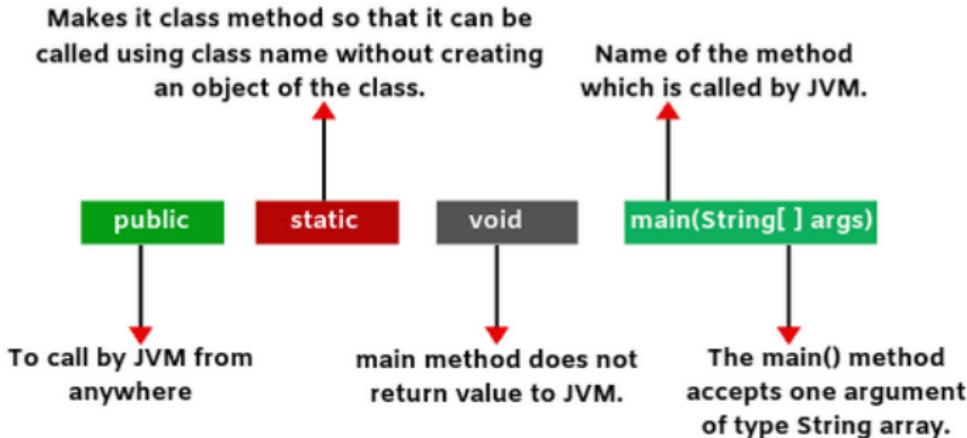


Fig: Java main method

## Object Oriented Programming Concepts III

Public Static void main

Accessible outside class

No need to create its  
object

Does not return  
anything

Execution of  
program starts  
from here

\* The order in which they occur is not important

## Object Oriented Programming Concepts IV

### large Parameters used in First Java Program:

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

- ✓ **class** keyword is used to declare a class in Java.
- ✓ **public** keyword is an access modifier which represents visibility. It means it is visible to all.
- ✓ **static** is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method.
- ✓ The **main** method is executed by the JVM, so it doesn' t require to create an object to invoke the main method. So it saves memory.
- ✓ **void** is the return type of the method. It means it doesn 't return any value.

## Object Oriented Programming Concepts V

- ✓ `main` represents the starting point of the program.
- ✓ `String[] args` is used for command line argument.

Compare with the arguments of main in C programming language: **int main(int argc, char \*argv[])**

- ✓ `System.out.println()` is used to print statement. Here, System is a class, out is the object of PrintStream class, `println()` is the method of PrintStream class.

## Object Oriented Programming Concepts VI

- Java file name be the same as the public class name that contains main method. Java file name must be "Simple.Java".

```
1 public class Simple
2 {
3     public static void main(String args[])
4     {
5         System.out.println("Hello Java");
6     }
7 }
```

- main method name must be within the public class.

## Object Oriented Programming Concepts VII

- One Java file can consist of multiple classes with the restriction that only one of them can be public.

```
1 public class Class1
2 {
3
4 }
5 class Class2
6 {
7
8 }
9 class Class3
10 {
11
12 }
```

## Object Oriented Programming Concepts VIII

- Compiler generates separate .class file of all classes after compilation of Java file.

```
1 public class Class1
2 {
3     public static void main(String args[])
4     {
5         System.out.println("Hello Java");
6     }
7 }
8 class Class2
9 {
10 }
11 }
12 class Class3
13 {
14 }
```

After compilation of Class1.Java, it generates three class files, also called bytecode, Class1.class, Class2.class and Class3.class.

## Object Oriented Programming Concepts IX

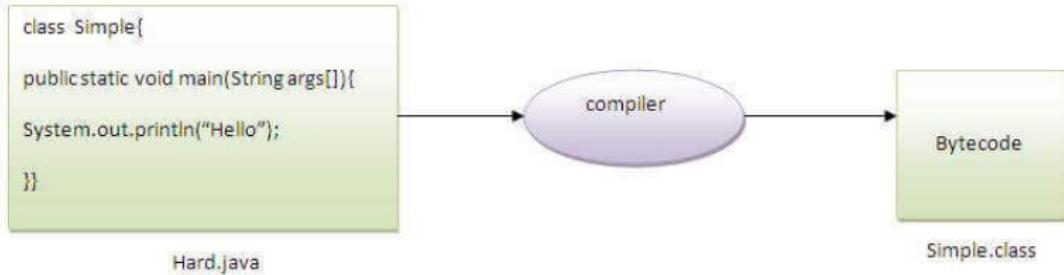
- A class file that contains the main method can be used for execution. In the above example, only Class1.class can be the executable file.

## Object Oriented Programming Concepts X

**Can you save a Java source file by other name than the class name?**

## Object Oriented Programming Concepts XI

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

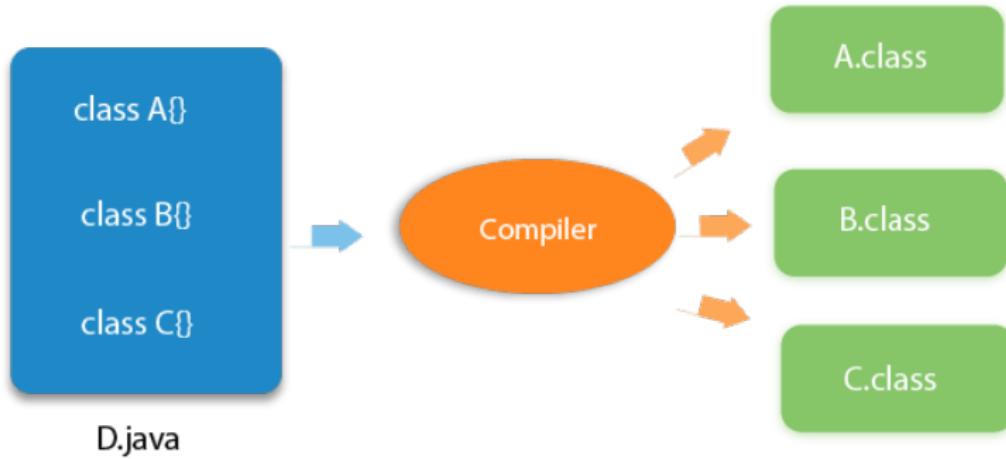


## Object Oriented Programming Concepts XII

**Can you have multiple classes in a Java source file?**

## Object Oriented Programming Concepts XIII

Yes, like the figure given below illustrates:



## Object Oriented Programming Concepts XIV

### How many ways can we write a Java program:

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

- ① By changing the sequence of the modifiers, method prototype is not changed in Java.

Let's see the simple code of the main method.

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

- ② The subscript notation in Java array can be used after type, before the variable or after the variable. Let's see the different codes to write the main method.

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

## Object Oriented Programming Concepts XV

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

Let's see the simple code of using var - args in the main method. We will learn about var - args later in Java New Features chapter.

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

4. Having a semicolon at the end of class is optional in Java. Let's see the simple code.

```
1 class A
2 {
3     static public void main(String... args)
4     {
5         System.out.println("Hello Java4");
6     }
7 }
8 ;
```

# Object Oriented Programming Concepts XVI

## Valid Java main method signature:

- 1 public static void main(String[] args)
- 2 public static void main(String []args)
- 3 public static void main(String args[])
- 4 public static void main(String... args)
- 5 static public void main(String[] args)
- 6 public static final void main(String[] args)
- 7 final public static void main(String[] args)
- 8 final strictfp public static void main(String[] args)

# Object Oriented Programming Concepts XVII

## Invalid Java main method signature:

- 1 `public void main(String[] args)`
- 2 `static void main(String[] args)`
- 3 `public void static main(String[] args)`
- 4 `abstract public static void main(String[] args)`

## Java Naming conventions I

- ✓ Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method, etc.
- ✓ But, it is not forced to follow. So, it is known as convention not rule. These conventions are suggested by several Java communities such as Sun Microsystems and Netscape.
- ✓ All the classes, interfaces, packages, methods and fields of Java programming language are given according to the Java naming convention. If you fail to follow these conventions, it may generate confusion or erroneous code.

### ► **Advantage of naming conventions in Java:**

## Java Naming conventions II

✓ By using standard Java naming conventions, you make your code easier to read for yourself and other programmers. Readability of Java program is very important.

✓ It indicates that less time is spent to figure out what the code does.

The following are the key rules that must be followed by every identifier:

- The name must not contain any white spaces.
- The name should not start with special characters like & (ampersand), \$ (dollar), \_ (underscore).
- Example: 4count, high-temp, Not/ok are invalid identifiers.

## Java Naming conventions III

Let's see some other rules that should be followed by identifiers.

### Class

- It should start with the uppercase letter.
- It should be a noun such as Color, Button, System, Thread, etc.
- Use appropriate words, instead of acronyms.

Example:-

```
1 public class Employee  
2 {  
3     //code snippet  
4 }
```

# Interface

- It should start with the uppercase letter.
- It should be an adjective such as Runnable, Remote, ActionListener.
- Use appropriate words, instead of acronyms.

Example:-

```
1 interface Printable
2 {
3     //code snippet
4 }
```

# Method

- It should start with lowercase letter.
- It should be a verb such as main(), print(), println().
- If the name contains multiple words, start it with a lowercase letter followed by an uppercase letter such as actionPerformed().

### Example:-

```
1 public class Employee
2 {
3     //method void draw()
4     {
5         //code snippet
6     }
7 }
```

# Variable

- It should start with a lowercase letter such as id, name.
- It should not start with the special characters like &(ampersand), \$(dollar), \_(underscore).
- If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as firstName, lastName.
- Avoid using one - character variables such as x, y, z.

Example:-

```
1 class Employee
2 {
3     //variable
4     int
5     id;
6     //code snippet
7 }
```

## Java Naming conventions VII

### Package

- It should be a lowercase letter such as Java, lang.
- If the name contains multiple words, it should be separated by dots(.) such as Java.util, Java.lang.

Example:-

```
1 package com.Javatpoint; //package
2 class Employee
3 {
4     //code snippet
5 }
```

# Constant

- It should be in uppercase letters such as RED, YELLOW.
- If the name contains multiple words, it should be separated by an underscore (\_)such as MAX\_PRIORITY.
- It may contain digits but not as the first letter.

Example: -

```
1 class Employee
2 {
3     //constant
4     static final int MIN_AGE = 18;
5     //code snippet
6 }
```

## Java Naming conventions IX

CamelCase in Java naming conventions:

- ✓ Java follows camel - case syntax for naming the class, interface, method, and variable.
- ✓ If the name is combined with two words, the second word will start with uppercase letter always such as actionPerformed (), firstName, ActionEvent, ActionListener, etc.

## Java Keywords I

- **large Separators:** In Java, there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon, and it is used to terminate statements.

Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a for statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.
::	Colons	Used to create a method or constructor reference. (Added by JDK 8.)

## Java Keywords II

**The Java Keywords:** There are 50 keywords currently defined in the Java language. These keywords, combined with the syntax of the operators and separators, form the foundation of the Java language.

- ✓ These keywords cannot be used as identifiers. Thus, they cannot be used as names for a variable, class, or method.
- ✓ The keywords const and goto are reserved but not used.
- ✓ In addition to the keywords, Java reserves the following: **true, false, and null.** These are values defined by Java. You may not use these words for the names of variables, classes, and so on.

## Java Keywords III

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

## Java Keywords IV

1	abstract:	Java abstract keyword is used to declare abstract class Abstract class can provide the implementation of interface It can have abstract and non - abstract methods
2	assert:	Assertion is a statement in Java It can be used to test your assumptions about the program While executing assertion, it is believed to be true If it fails, JVM will throw an error named AssertionError It is mainly used for testing purpose It provides an effective way to detect and correct programming errors
3	boolean:	Java boolean keyword is used to declare a variable as a boolean type It can hold True and False values only
4	break:	Java break keyword is used to break loop or switch statement It breaks the current flow of the program at specified condition

## Java Keywords V

5	byte:	Java byte keyword is used to declare a variable that can hold an 8 - bit data values
6	case:	Java case keyword is used to with the switch statements to mark blocks of text
7	catch:	Java catch keyword is used to catch the exceptions generated by try statements It must be used after the try block only
8	char:	Java char keyword is used to declare a variable that can hold unsigned 16 - bit Unicode characters
9	class:	Java class keyword is used to declare a class
10	continue:	Java continue keyword is used to continue the loop It continues the current flow of the program and skips the remaining code at the specified condition

## Java Keywords VI

11	default:	Java default keyword is used to specify the default block of code in a switch statement
12	do:	Java do keyword is used in control statement to declare a loop It can iterate a part of the program several times
13	double:	Java double keyword is used to declare a variable that can hold a 64 - bit floating - point numbers
14	else:	Java else keyword is used to indicate the alternative branches in an if statement
15	enum:	Java enum keyword is used to define a fixed set of constants Enum constructors are always private or default
16	extends:	Java extends keyword is used to indicate that a class is derived from another class or interface

## Java Keywords VII

17	final:	Java final keyword is used to indicate that a variable holds a constant value It is applied with a variable It is used to restrict the user
18	finally:	Java finally keyword indicates a block of code in a try - catch structure This block is always executed whether exception is handled or not
19	float:	Java float keyword is used to declare a variable that can hold a 32 - bit floating - point number
20	for:	Java for keyword is used to start a for loop It is used to execute a set of instructions / functions repeatedly when some conditions become true If the number of iteration is fixed, it is recommended to use for loop
21	if:	Java if keyword tests the condition It executes the if block if condition is true

## Java Keywords VIII

22	implements:	Java implements keyword is used to implement an interface
23	import:	Java import keyword makes classes and interfaces available and accessible to the current source code
24	instanceof:	Java instanceof keyword is used to test whether the object is an instance of the specified class or implements an interface
25	int:	Java int keyword is used to declare a variable that can hold a 32 - bit signed integer
26	interface:	Java interface keyword is used to declare an interface It can have only abstract methods
27	long:	Java long keyword is used to declare a variable that can hold a 64 - bit integer

## Java Keywords IX

28	native:	Java native keyword is used to specify that a method is implemented in native code using JNI (Java Native Interface)
29	new:	Java new keyword is used to create new objects
30	null:	Java null keyword is used to indicate that a reference does not refer to anything It removes the garbage value
31	package:	Java package keyword is used to declare a Java package that includes the classes
32	private:	Java private keyword is an access modifier It is used to indicate that a method or variable may be accessed only in the class in which it is declared
33	protected:	Java protected keyword is an access modifier It can be accessible within package and outside the package but through inheritance only It can 't be applied on the class

## Java Keywords X

34	public:	Java public keyword is an access modifier It is used to indicate that an item is accessible anywhere It has the widest scope among all other modifiers
35	return:	Java return keyword is used to return from a method when its execution is complete
36	short:	Java short keyword is used to declare a variable that can hold a 16-bit integer
37	static:	Java static keyword is used to indicate that a variable or method is a class method The static keyword in Java is used for memory management mainly
38	strictfp:	Java strictfp is used to restrict the floating-point calculations to ensure portability

## Java Keywords XI

39	super:	Java super keyword is a reference variable that is used to refer parent class object It can be used to invoke immediate parent class method
40	switch:	The Java switch keyword contains a switch statement that executes code based on test value The switch statement tests the equality of a variable against multiple values
41	synchronized:	Java synchronized keyword is used to specify the critical sections or methods in multithreaded code
42	this:	Java this keyword can be used to refer the current object in a method or constructor
43	throw:	The Java throw keyword is used to explicitly throw an exception The throw keyword is mainly used to throw custom exception It is followed by an instance

## Java Keywords XII

44	throws:	The Java throws keyword is used to declare an exception Checked exception can be propagated with throws
45	transient:	Java transient keyword is used in serialization If you define any data member as transient, it will not be serialized
46	try:	Java try keyword is used to start a block of code that will be tested for exceptions The try block must be followed by either catch or finally block
47	void:	Java void keyword is used to specify that a method does not have a return value
48	volatile:	Java volatile keyword is used to indicate that a variable may change asynchronously

## Java Keywords XIII

49	while:	Java while keyword is used to start a while loop This loop iterates a part of the program several times If the number of iteration is not fixed, it is recommended to use while loop
----	--------	--

## Java Keywords XIV

### Java Data Types

- ✓ Java Is a Strongly Typed Language
- ✓ Indeed, part of Java's safety and robustness comes from this fact.
- ✓ Every variable has a type, every expression has a type, and every type is strictly defined.
- ✓ All assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.
- ✓ There are no automatic coercions or conversions of conflicting types as in some languages.
- ✓ The Java compiler checks all expressions and parameters to ensure that the types are compatible.
- ✓ Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.

## Data Types, Variables, and Arrays II

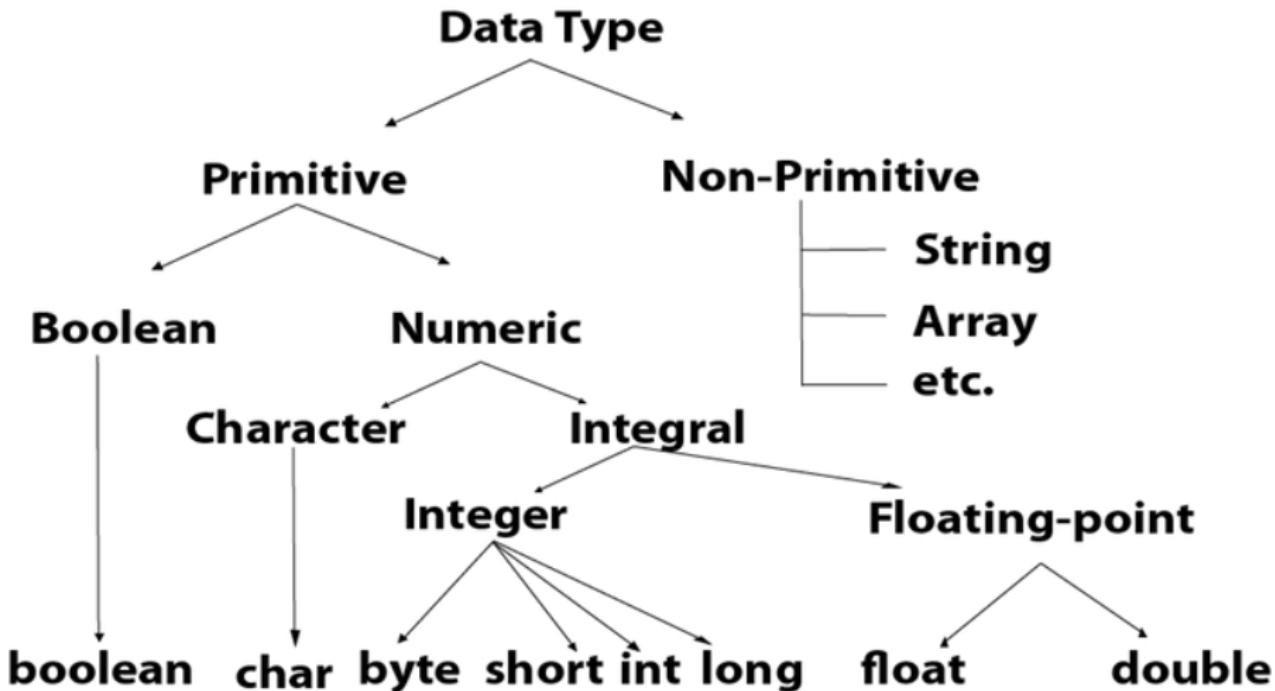


Figure 1: Java Data type

## Data Types, Variables, and Arrays III

Primitive Type	Size	Minimum Value	Maximum Value	Wrapper Type
<b>char</b>	16-bit	Unicode 0	Unicode $2^{16}-1$	Character
<b>byte</b>	8-bit	-128	+127	Byte
<b>short</b>	16-bit	$-2^{15}$ (-32,768)	$+2^{15}-1$ (32,767)	Short
<b>int</b>	32-bit	$-2^{31}$ (-2,147,483,648)	$+2^{31}-1$ (2,147,483,647)	Integer
<b>long</b>	64-bit	$-2^{63}$ (-9,223,372,036,854,775,808)	$+2^{63}-1$ (9,223,372,036,854,775,807)	Long
<b>float</b>	32-bit	32-bit IEEE 754 floating-point numbers		
<b>double</b>	64-bit	64-bit IEEE 754 floating-point numbers		
<b>boolean</b>	1-bit	<code>true</code> or <code>false</code>		
<b>void</b>	-----	-----	-----	Void

**Figure 2:** Range of data type

## Data Types, Variables, and Arrays IV

```
1 // Compute distance light travels using long variables.
2 public class Light
3 {
4     public static void main(String args[])
5     {
6         int lightspeed;
7         long days;
8         long seconds;
9         long distance;
10        // approximate speed of light in miles per second
11        lightspeed = 186000;
12        days = 1000; // specify number of days here
13        seconds = days * 24 * 60 * 60; // convert to seconds
14        distance = lightspeed * seconds; // compute distance
15        System.out.print("In " + days);
16        System.out.print(" days light will travel about ");
17        System.out.println(distance + " miles.");
18        double pi, r, a;
19        r = 10.8; // radius of circle
20        pi = 3.1416; // pi, approximately
21        a = pi * r * r; // compute area
22        System.out.println("Area of circle is " + a);
23        char ch1, ch2;
24        ch1 = 88; // code for X
25        ch2 = 'Y';
```

## Data Types, Variables, and Arrays V

```
26     System.out.print("ch1 and ch2: ");
27     System.out.println(ch1 + " " + ch2);
28     boolean b;
29     b = false;
30     System.out.println("b is " + b);
31     b = true;
32     System.out.println("b is " + b);
33     // a boolean value can control the if statement
34     if(b) System.out.println("This is executed.");
35     b = false;
36     if(b) System.out.println("This is not executed.");
37     // outcome of a relational operator is a boolean value
38     System.out.println("10 > 9 is " + (10 > 9));
39 }
40 }
```

## Data Types, Variables, and Arrays VI

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	\u0000
String (or any object)	null
boolean	false

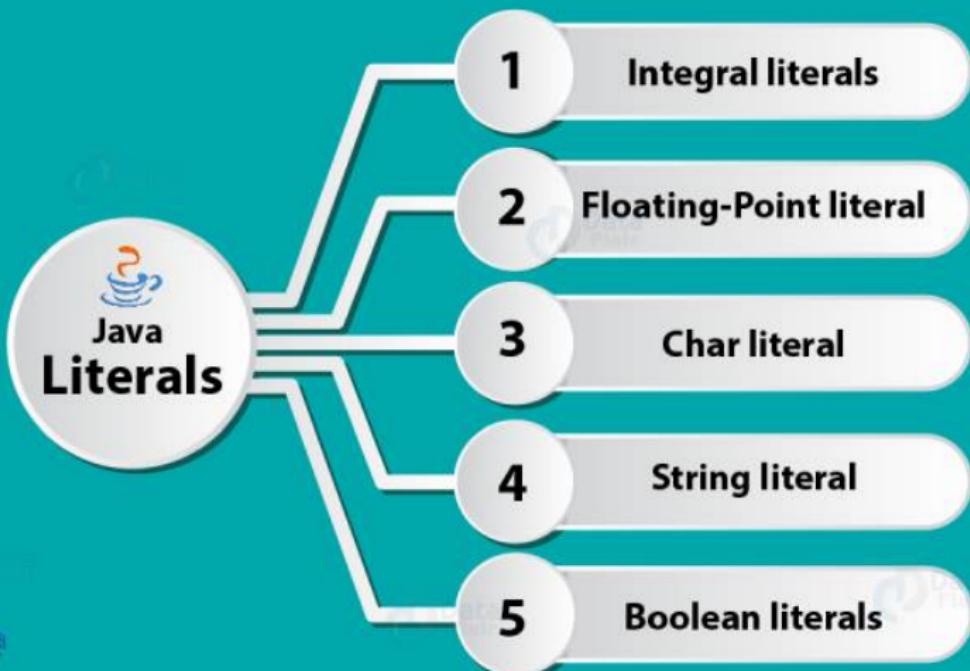
**Table 2:** Default values for the data types

## Data Types, Variables, and Arrays VII

### Java Literals

Java Literals are syntactic representations of boolean, character, numeric, or string data. Literals provide a means of expressing specific values in your program.

## Data Types, Variables, and Arrays VIII



## Data Types, Variables, and Arrays IX

- ✓ Any constant value which can be assigned to the variable is called as literal/constant.

For example, in the following statement, an integer variable named count is declared and assigned an integer value.

- ✓ Two other bases that can be used in integer literals are octal (base eight) and hexadecimal (base 16). Octal values are denoted in Java by a leading zero.
- ✓ A more common base for numbers used by programmers is hexadecimal, which matches cleanly with modulo 8 word sizes, such as 8, 16, 32, and 64 bits.
- ✓ A hexadecimal constant is signify with a leading zero-x, (0x or 0X). The range of a hexadecimal digit is 0 to 15, so A through F (or a through f ) are substituted for 10 through 15.

# Data Types, Variables, and Arrays X

## Integer Literals:

```
1 int x = 100; // Here 100 is a constant/literal.  
2 int y = 0b1010;  
3 int z = 123_456_789;  
4 int xx = 123_456_789;  
5  
6 long creditCardNumber = 1234_5678_9012_3456L;  
7 long socialSecurityNumber = 999_99_9999L;  
8 float pi = 3.14_15F;  
9 long hexBytes = 0xFF_EC_DE_5E; //Error  
10 long hexWords = 0xCAFE_BABE; //Error  
11 long maxLong = 0x7fff_ffff_ffff_ffffL;  
12 byte nybbles = 0b0010_0101;  
13 long bytes = 0b11010010_01101001_10010100_10010010;
```

## Data Types, Variables, and Arrays XI

```
1 // Java program to illustrate the application of Integer literals
2 public class Test
3 {
4     public static void main(String[] args)
5     {
6         int a = 101; // decimal-form literal
7         int b = 0100; // octal-form literal
8         int c = 0xFace; // Hexa-decimal form literal
9         int d = 0b1111; // Binary literal
10        System.out.println(a);
11        System.out.println(b);
12        System.out.println(c);
13        System.out.println(d);
14    }
15 }
16 Output:
17 101
18 64
19 64206
20 15
```

You can place underscores only between digits; you cannot place underscores in the following places:

## Data Types, Variables, and Arrays XII

- At the beginning or end of a number
- Adjacent to a decimal point in a floating point literal
- Prior to an F or L suffix
- In positions where a string of digits is expected

*Note: By default, every literal is of int type, we can specify explicitly as long type by suffixed with l or L. There is no way to specify byte and short literals explicitly but indirectly we can specify. Whenever we are assigning integral literal to the byte variable and if the value within the range of byte then the compiler treats it automatically as byte literals.*

✓ The following examples demonstrate valid and invalid underscore placements (which are highlighted) in numeric literals:

# Data Types, Variables, and Arrays XIII

```
1 // Invalid: cannot put underscores
2 // prior to an L suffix
3 long socialSecurityNumber1 = 999_99_9999_L;
4
5 // OK (decimal literal)
6 int x1 = 5_2;
7 // Invalid: cannot put underscores
8 // At the end of a literal
9 int x2 = 52_;
10 // OK (decimal literal)
11 int x3 = 5_____2;
12
13 // Invalid: cannot put underscores
14 // in the 0x radix prefix
15 int x4 = 0_x52;
16 // Invalid: cannot put underscores
17 // at the beginning of a number
18 int x5 = 0x_52;
19 // OK (hexadecimal literal)
20 int x6 = 0x5_2;
21 // Invalid: cannot put underscores
22 // at the end of a number
23 int x7 = 0x52_;
```

### Floating-Point Literals

- ✓ Floating-point literals in Java default to double precision.
- ✓ To specify a float literal, you must append an *F* or *f* to the constant. You can also explicitly specify a double literal by appending a *D* or *d*.
- ✓ Scientific notation uses a standard-notation, floating-point number plus a suffix that specifies a power of 10 by which the number is to be multiplied.
- ✓ Hexadecimal floating-point literals are also supported, but they are rarely used. They must be in a form similar to scientific notation, but a *P* or *p*, rather than an *E* or *e*, is used.
- ✓ For example, 0x12.2P2 is a valid floating-point literal. The value following the *P*, called the *binary exponent*, indicates the

## Data Types, Variables, and Arrays XV

power-of-two by which the number is multiplied. Therefore, 0x12.2P2 represents 72.5.

```
1  double num = 9_423_497_862.0;
2  double num = 9_423_497.1_0_9;
3  double d1 = 123.4;
4  // same value as d1, but in scientific notation
5  double d2 = 1.234e2;
6  float f1 = 123.4f;

1 // Java program to illustrate the application of floating-point literals
2 public class Test
3 {
4     public static void main(String[] args)
5     {
6         float a = 101.230; // decimal-form literal
7         float b = 0123.222; // It also acts as decimal literal
8         float c = 0x123.222; // Hexa-decimal form
9         System.out.println(a);
10        System.out.println(b);
11        System.out.println(c);
12    }
13 }
14 Output:
15 101.230
```

## Data Types, Variables, and Arrays XVI

16 | 123.222

```
1 // Invalid: cannot put underscores
2 // adjacent to a decimal point
3 float pi1 = 3_.1415F;
4 // Invalid: cannot put underscores
5 // adjacent to a decimal point
6 float pi2 = 3._1415F;
7 // Invalid: cannot put underscores
8 // prior to an F suffix
9 float pi1 =3.1415_F;
```

### Char literal

For char data types we can specify literals in 4 ways:

- ① Single quote : We can specify literal to char data type as single character within single quote.

```
1 char ch = ' a ';
```

- ② Char literal as Integral literal : we can specify char literal as integral literal which represents Unicode value of the character and that integral literals can be specified either in Decimal, Octal and Hexadecimal forms. But the allowed range is 0 to 65535.

```
1 char ch = 062;
```

- ③ Unicode Representation : We can specify char literals in Unicode representation \uxxxx . Here xxxx represents 4 hexadecimal numbers.

## Data Types, Variables, and Arrays XVIII

```
1 char ch = ' \ u0061'; // Here /u0061 represent a.
```

- ④ Escape Sequence : Every escape character can be specify as char literals.

```
1 char ch = ' \ n ';
```

```
1 // Java program to illustrate the application of char literals
2 public class Test
3 {
4     public static void main(String[] args)
5     {
6         char ch = ' a'; // signle character literl within signle quote
7         char b = 0789; // It is an Integer literal with octal form
8         char c = ' \ u0061'; // Unicode representation
9         System.out.println(ch);
10        System.out.println(b);
11        System.out.println(c);
12        // Escape character literal
13        System.out.println("\\" is a symbol");
14    }
15 }
16 a
17 error:Integer number too large
```

## Data Types, Variables, and Arrays XIX

18  
19

```
a
" is a symbol
```

## Data Types, Variables, and Arrays XX

### String literal

Any sequence of characters within double quotes is treated as String literals.

```
1 String s = "Hello";
```

- ✓ String literals may not contain unescaped newline or linefeed characters.
- ✓ However, the Java compiler will evaluate compile time expressions, so the following String expression results in a string with three lines of text:

```
1 Example:  
2 String text = "This is a String literal\n"  
3 + "which spans not one and not two\n"  
4 + "but three lines of text.\n";
```

## Data Types, Variables, and Arrays XXI

```
1 // Java program to illustrate the application of String literals
2 public class Test
3 {
4     public static void main(String[] args)
5     {
6         String s = "Hello";
7         // If we assign without "" then it treats as a variable and causes compiler error
8         String s1 = Hello;
9         System.out.println(s);
10        System.out.println(s1);
11    }
12 }
13 Output: ????
```

## Data Types, Variables, and Arrays XXII

```
1 Hello
2 error: cannot find symbol
3 symbol: variable Hello
4 location: class Test
```

Escape Sequence	Description
\ddd Octal	character (ddd)
\uxxxx	Hexadecimal Unicode character (xxxx)
\'	Single quote
textbackslash "	Double quote
\\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Backspace

# Data Types, Variables, and Arrays XXIII

## Boolean literals

Only two values are allowed for Boolean literals i.e. true and false.

```
1 boolean b = true;
2
3 // Java program to illustrate the application of boolean literals
4 public class Test
5 {
6     public static void main(String[] args)
7     {
8         boolean b = true;
9         boolean c = false;
10        boolean d = 0;
11        boolean e = 1;
12        System.out.println(b);
13        System.out.println(c);
14        System.out.println(d);
15        System.out.println(e);
16    }
17 }
```

Output: ???

## Data Types, Variables, and Arrays XXIV

```
1 true
2 false
3 error: incompatible types: int cannot be converted to boolean
4 error: incompatible types: int cannot be converted to boolean
1 //Java program to illustrate the behaviour of char literals and integer literals when
   we are performing addition
2 public class Test
3 {
4     public static void main(String[] args)
5     {
6         // ASCII value of 0 is 48
7         int first = '0';
8         // ASCII value of 7 is 55
9         int second = '7';
10        System.out.println(" MCAMSC ! " + first + '2' + second);
11    }
12 }
13 Output: ???
```

# Data Types, Variables, and Arrays XXV

1 MCAMSC!48255

2 Explanation : ???

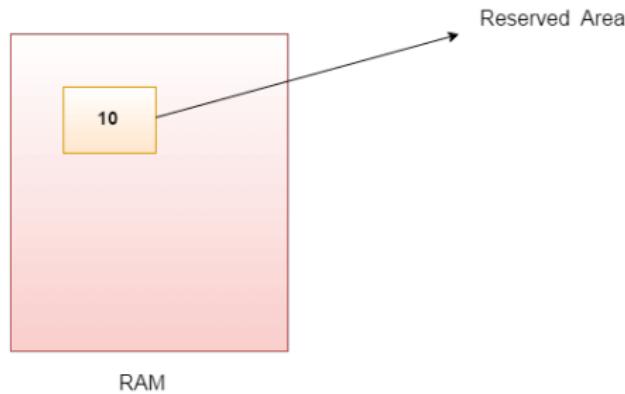
### Variables

- ✓ The variable is the basic unit of storage in a Java program.
- ✓ Variable is a name of memory location.
- ✓ A variable is defined by the combination of an identifier, a type, and an optional initializer.
- ✓ In addition, all variables have a scope, which defines their visibility, and a lifetime.
- ✓ Variable is name of reserved area allocated in memory. In other words, it is a name of memory location. It is a combination of " vary + able " that means its value can be changed.

**Declaring a Variable:** In Java, all variables must be declared before they can be used. The basic form of a variable declaration is :  
*type identifier [ = value ][, identifier [= value ] ...];*

## Data Types, Variables, and Arrays XXVII

```
1 int data=50; //Here data is variable
2 int a, b, c; //declares three ints, a, b, and c.
3 int d = 3, e, f = 5; //declares three more ints, initializing d and f.
4 byte z = 22; //initializes z.
5 double pi = 3.14159; //declares an approximation of pi.
6 char x = 'x'; //the variable x has the value 'x'.
```



### ► Dynamic Initialization

- ✓ Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.
- ✓ For example, here is a short program that computes the length of the hypotenuse of a right triangle given the lengths of its two opposing sides:

```
1 // Demonstrate dynamic initialization.
2 public class DynInit
3 {
4     public static void main(String args[])
5     {
6         double a = 3.0, b = 4.0;
7         // c is dynamically initialized
8         double c = Math.sqrt(a * a + b * b);
9         System.out.println(" Hypotenuse is " + c);
10    }
11 }
```

## Data Types, Variables, and Arrays XXIX

### ► The Scope and Lifetime of Variables:

- ✓ Java allows variables to be declared within any block.
- ✓ A block defines a *scope*. Thus, each time you start a new block, you are creating a new scope.
- ✓ A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

```
1 #include<stdio.h>
2 int square(int);
3 int main(void)
4 {
5     int a;
6     a=square(12);
7     printf("\n %d",a);
8 }
9
10 int square(int i)
11 {
12     int b;
13     b=i*i;
```

# Data Types, Variables, and Arrays XXX

```
14     return b;
15 }

1 // Demonstrate block scope.
2 public class Scope
3 {
4     public static void main(String args[])
5     {
6         int x; // known to all code within main
7         x = 10;
8         if(x == 10)
9         {
10             // start new scope
11             int y = 20; // known only to this block
12             // x and y both known here.
13             System.out.println(" x and y: " + x + " " + y);
14             x = y * 2;
15         }
16         y = 100; // Error! y not known here
17         // x is still known here.
18         System.out.println(" x is " + x);
19     }
20 }
```

## Data Types, Variables, and Arrays XXXI

- ✓ Within a block, variables can be declared at any point, but are valid only after they are declared.

```
1 // This fragment is wrong!
2 count = 100; // oops! cannot use count before it is declared!
3 int count;
```

- ✓ Variables are created when their scope is entered, and destroyed when their scope is left.
- ✓ This means that a variable will not hold its value once it has gone out of scope.
- ✓ Therefore, variables declared within a method will not hold their values between calls to that method.
- ✓ Also, a variable declared within a block will lose its value when the block is left.

## Data Types, Variables, and Arrays XXXII

- ✓ If a variable declaration includes an initializer, then that variable will be reinitialized each time the block in which it is declared is entered.
- ✓ For example, consider the next program:

```
1 // Demonstrate lifetime of a variable.
2 class LifeTime
3 {
4     public static void main(String args[])
5     {
6         int x;
7         for(x = 0; x < 3; x++)
8         {
9             int y = -1; // y is initialized each time block is entered
10            System.out.println(" y is: " + y); // this always prints -1
11            y = 100;
12            System.out.println(" y is now: " + y);
13        }
14    }
15 }
16 The output generated by this program is shown here:
17 y is: -1
18 y is now: 100
19 y is: -1
20 y is now: 100
```

## Data Types, Variables, and Arrays XXXIII

```
21 y is: -1  
22 y is: 100
```

- ✓ Although blocks can be nested, you cannot declare a variable to have the same name as one in an outer scope.
- ✓ For example, the following program is illegal:

```
1 // This program will not compile  
2 public class ScopeErr  
3 {  
4     public static void main(String args[])  
5     {  
6         int var = 1;  
7         {  
8             int var = 2;  
9                 // creates a new scope  
10                // Compile-time error - var already defined!  
11            }  
12        }  
13    }
```

### ► Type Conversion and Casting:

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

### Java's Automatic Conversions:

## Data Types, Variables, and Arrays XXXV

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

- The two types are compatible.
  - The destination type is larger than the source type.
- ✓ When these two conditions are met, a widening conversion takes place.
- ✓ For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement is required.
- ✓ For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other.

## Data Types, Variables, and Arrays XXXVI

- ✓ However, there are no automatic conversions from the numeric types to *char* or *boolean*. Also, *char* and *boolean* are not compatible with each other.
- ✓ Java also performs an automatic type conversion when storing a literal integer constant into variables of type *byte*, *short*, *long*, or *char*.

### Casting Incompatible Types:

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

*(target-type) value*

```
1 int a;  
2 byte b;  
3 // ...  
4 b = (byte) a;
```

## Data Types, Variables, and Arrays XXXVII

- ✓ A different type of conversion will occur when a floating-point value is assigned to an integer type: *truncation*.
- ✓ when a floating-point value is assigned to an integer type, the fractional component is lost. For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1.

```
1 // Demonstrate casts.
2 class Conversion
3 {
4     public static void main(String args[])
5     {
6         byte b;
7         int i = 257;
8         double d = 323.142;
9         System.out.println(" \ nConversion of int to byte.");
10        b = (byte) i;
11        System.out.println(" i and b " + i + " " + b);
12        System.out.println(" \ nConversion of double to int.");
13        i = (int) d;
14        System.out.println(" d and i " + d + " " + i);
15        System.out.println(" \ nConversion of double to byte.");
16        b = (byte) d;
```

## Data Types, Variables, and Arrays XXXVIII

```
17     System.out.println(" d and b " + d + " " + b);
18 }
19 }
20 This program generates the following output:
21 Conversion of int to byte.
22 i and b 257 1
23 Conversion of double to int.
24 d and i 323.142 323
25 Conversion of double to byte.
26 d and b 323.142 67
```

- ✓ When the value 257 is cast into a byte variable, the result is the remainder of the division of 257 by 256 (the range of a byte), which is 1 in this case.
- ✓ When the d is converted to an int, its fractional component is lost.
- ✓ When d is converted to a byte, its fractional component is lost, and the value is reduced modulo 256, which in this case is 67.

### Automatic Type Promotion in Expressions:

- ✓ There is another place where certain type conversions may occur in expressions.
- ✓ In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand.

```
1 byte a = 40;  
2 byte b = 50;  
3 byte c = 100;  
4 int d = a * b / c;  
5 Output:  
6 c= ???
```

## Data Types, Variables, and Arrays XL

- ✓ The result of the intermediate term  $a * b$  easily exceeds the range of either of its byte operands.
- ✓ To handle this kind of problem, Java automatically promotes each byte, short, or char operand to int when evaluating an expression.
- ✓ This means that the subexpression  $a * b$  is performed using integers—not bytes.
- ✓ Thus, 2,000, the result of the intermediate expression,  $50 * 40$ , is legal even though  $a$  and  $b$  are both specified as type byte.
- ✓ As useful as the automatic promotions are, they can cause confusing compile-time errors. For example, this seemingly correct code causes a problem:

## Data Types, Variables, and Arrays XLI

```
1 byte b = 50;  
2 b = b * 2;  
3  
4 Output:  
5 b = ????  
6
```

## Data Types, Variables, and Arrays XLII

Output: Error! Cannot assign an int to a byte!

- The code is attempting to store  $50 * 2$ , a perfectly valid byte value, back into a byte variable.
  - However, because the operands were automatically promoted to int when the expression was evaluated, the result has also been promoted to int.
  - Thus, the result of the expression is now of type int, which cannot be assigned to a byte without the use of a cast.
- ✓ In cases where you understand the consequences of overflow, you should use an explicit cast.

```
1 byte b = 50;  
2 b = (byte)(b * 2);  
3 Output: ???
```

## Data Types, Variables, and Arrays XLIII

Output: 100.

### The Type Promotion Rules:

- ✓ Java defines several type promotion rules that apply to expressions.
- ✓ They are as follows: First, all *byte*, *short*, and *char* values are promoted to *int*.
- ✓ Then, if one operand is a *long*, the whole expression is promoted to *long*.
- ✓ If one operand is a *float*, the entire expression is promoted to *float*.
- ✓ If any of the operands are *double*, the result is *double*.

# Data Types, Variables, and Arrays XLV

```
1 public class Promote
2 {
3     public static void main(String args[])
4     {
5         byte b = 42;
6         char c = 'a';
7         short s = 1024;
8         int i = 50000;
9         float f = 5.67f;
10        double d = .1234;
11        double result = (f * b) + (i/ c) - (d * s);
12        System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
13        System.out.println(" result = " + result);
14    }
15 }
16
17 double result = (f * b) + (i / c) - (d * s);
```

## Data Types, Variables, and Arrays XLVI

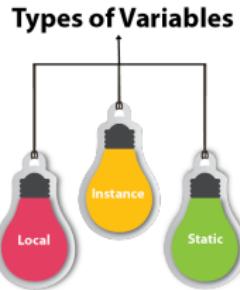
- ⇒ In the first subexpression,  $f * b$ ,  $b$  is promoted to a *float* and the result of the subexpression is *float*.
- ⇒ Next, in the subexpression  $i/c$ ,  $c$  is promoted to *int*, and the result is of type *int*.
- ⇒ Then, in  $d * s$ , the value of  $s$  is promoted to *double*, and the type of the subexpression is *double*. Finally, these three intermediate values, *float*, *int*, and *double*, are considered.
- ⇒ The outcome of *float* plus an *int* is a *float*. Then the resultant *float* minus the last *double* is promoted to *double*, which is the type for the final result of the expression.

# Types of Variables I

## ► Types of Variables:

There are three types of variables in Java:

- ① local variable
- ② instance variable
- ③ static variable



## Types of Variables II

1. Local Variable: A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

Can a local variable be defined with " static " keyword?

**If no, then why ????????**

## Types of Variables III

In Java, a static variable is a class variable (for whole class). So if we have static local variable (a variable with scope limited to function), it violates the purpose of static. Hence compiler does not allow static local variable.

② **Instance Variable:** A variable declared inside the class but outside the body of the method, is called instance variable. It is not declared as static.

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When space is allocated for an object in the heap, a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.

## Types of Variables IV

- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
- Instance variables can be declared at the class level before or after use.
- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors, and block in the class. Normally, it is recommended to make these variables private (access level). However, visibility for subclasses can be given for these variables with the use of access modifiers.

## Types of Variables V

- Instance variables have default values. For numbers, the default value is 0, for Booleans it is false, and for object references it is null. Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class. However, within static methods (when instance variables are given accessibility), they should be called using the fully qualified name. ObjectReference.VariableName.

It is called instance variable because its value is instance specific and is not shared among instances.

## Types of Variables VI

```
1 import java.io.*;
2 public class Employee
3 {
4
5     // this instance variable is visible for any child class.
6     public String name;
7
8     // salary variable is visible in Employee class only.
9     private double salary;
10
11    // The name variable is assigned in the constructor.
12    public Employee (String empName)
13    {
14        name = empName;
15    }
16
17    // The salary variable is assigned a value.
18    public void setSalary(double empSal)
19    {
20        salary = empSal;
21    }
22
23    // This method prints the employee details.
24    public void printEmp()
25    {
```

## Types of Variables VII

```
26     System.out.println("name : " + name );
27     System.out.println("salary :" + salary);
28 }
29
30 public static void main(String args[])
31 {
32     Employee empOne = new Employee("Ransika");
33     empOne.setSalary(1000);
34     empOne.printEmp();
35 }
36 }
```

## Types of Variables VIII

③ Static variable: A variable which is declared as static is called static variable.

- ✓ It cannot be local.
- ✓ You can create a single copy of static variable and share among all the instances of the class.
- ✓ Memory allocation for static variable happens only once when the class is loaded in the memory.

```
1 public class A
2 {
3     int data=50; //instance variable
4     static int m=100; //static variable
5     void method()
6     {
7         int n=90; //local variable
8     }
9 }
10 //end of class
```

## Types of Variables IX

*static* is a non-access modifier in Java which is applicable for the following:

- A.** blocks
- B.** variables
- C.** methods
- D.** nested classes

### **A.** Blocks:

- ✓ To create a static member (block, variable, method, nested class), precede its declaration with the keyword `static`.
- ✓ When a member is declared `static`, it can be accessed before any objects of its class are created, and without reference to any object.

## Types of Variables X

**Static blocks in Java:** Java supports a special block, called static block (also called static clause) which can be used for static initializations of a class.

```
1 public class Test
2 {
3     static int i;
4     int j;
5
6     // start of static block
7     static
8     {
9         i = 10;
10        System.out.println("static block called ");
11    }
12    // end of static block
13 }
14 class Main
15 {
16     public static void main(String args[])
17     {
18         // Although we don't have an object of Test, static block is
19         // called because i is being accessed in following statement.
20         System.out.println(Test.i);
```

## Types of Variables XI

```
21     }  
22 }
```

✓ Also, static blocks are executed before constructors.

```
1 class Test  
2 {  
3     static int i;  
4     int j;  
5     static  
6     {  
7         i = 10;  
8         System.out.println("static block called ");  
9     }  
10    Test()  
11    {  
12        System.out.println("Constructor called");  
13    }  
14}  
15 public class Main  
16 {  
17     public static void main(String args[])  
18     {  
19         // Although we have two objects, static block is executed only once.  
20         Test t1 = new Test();  
21         Test t2 = new Test();
```

## Types of Variables XII

```
22     }
23 }
24 Output:
25 static block called
26 Constructor called
27 Constructor called
```

## Types of Variables XIII

```
1 // Java program to demonstrate use of static blocks
2 public class Test
3 {
4     // static variable
5     static int a = 10;
6     static int b;
7     // static block
8     static
9     {
10         System.out.println("Static block initialized.");
11         b = a * 4;
12     }
13     public static void main(String[] args)
14     {
15         System.out.println("from main");
16         System.out.println("Value of a : "+a);
17         System.out.println("Value of b : "+b);
18     }
19 }
```

## Types of Variables XIV

B. Static Variables: When a variable is declared as static, then a single copy of variable is created and shared among all objects at class level. Static variables are, essentially, global variables. All instances of the class share the same static variable.

Important points for static variables :-

- We can create static variables at class-level only. See here
- static block and static variables are executed in order they are present in a program.

## Types of Variables XV

```
1 // java program to demonstrate execution of static blocks and variables
2 public class Test
3 {
4     // static variable
5     static int a = m1();
6     // static block
7     static
8     {
9         System.out.println("Inside static block");
10    }
11    // static method
12    static int m1()
13    {
14        System.out.println("from m1");
15        return 20;
16    }
17    // static method(main !!)
18    public static void main(String[] args)
19    {
20        System.out.println("Value of a : "+a);
21        System.out.println("from main");
22    }
23 }
```

## Types of Variables XVI

c. Static methods: When a method is declared with static keyword, it is known as static method.

- ✓ The most common example of a static method is main( ) method.
- ✓ Any static member can be accessed before any objects of its class are created, and without reference to any object.
- ✓ Methods declared as static have several restrictions:
  - They can only directly call other static methods.
  - They can only directly access static data.
  - They cannot refer to this or super in any way.
- ✓ For example, in below java program, we are accessing static method m1() without creating any object of Test class.

## Types of Variables XVII

```
1 // Java program to demonstrate that a static member
2 // can be accessed before instantiating a class
3 public class Test
4 {
5     // static method
6     static void m1()
7     {
8         System.out.println("from m1");
9     }
10    public static void main(String[] args)
11    {
12        // calling m1 without creating
13        // any object of class Test
14        m1();
15    }
16 }
```

## Types of Variables XVIII

```
1 // java program to demonstrate restriction on static methods
2 public class Test
3 {
4     // static variable
5     static int a = 10;
6     // instance variable
7     int b = 20;
8     // static method
9     static void m1()
10    {
11        a = 20;
12        System.out.println("from m1");
13
14        // Cannot make a static reference to the non-static field b
15        b = 10; // compilation error
16
17        // Cannot make a static reference to the
18        // non-static method m2() from the type Test
19        m2(); // compilation error
20
21        // Cannot use super in a static context
22        System.out.println(super.a); // compiler error
23    }
24    // instance method
25    void m2()
```

## Types of Variables XIX

```
26     {
27         System.out.println("from m2");
28     }
29     public static void main(String[] args)
30     {
31         // main method
32     }
33 }
```

## Types of Variables XX

### When to use static variables and methods?

- ✓ Use the static variable for the property that is common to all objects.
- ✓ For example, in class Student, all students shares the same college name. Use static methods for changing static variables.

```
1 // A java program to demonstrate use of static keyword with methods and variables Student class
2 class Student
3 {
4     String name;
5     int rollNo;
6     // static variable
7     static String cllgName;
8     // static counter to set unique roll no
9     static int counter = 0;
10    public Student(String name)
11    {
12        this.name = name;
13        this.rollNo = setRollNo();
14    }
15    //getting unique rollNo through static variable(counter)
16    static int setRollNo()
```

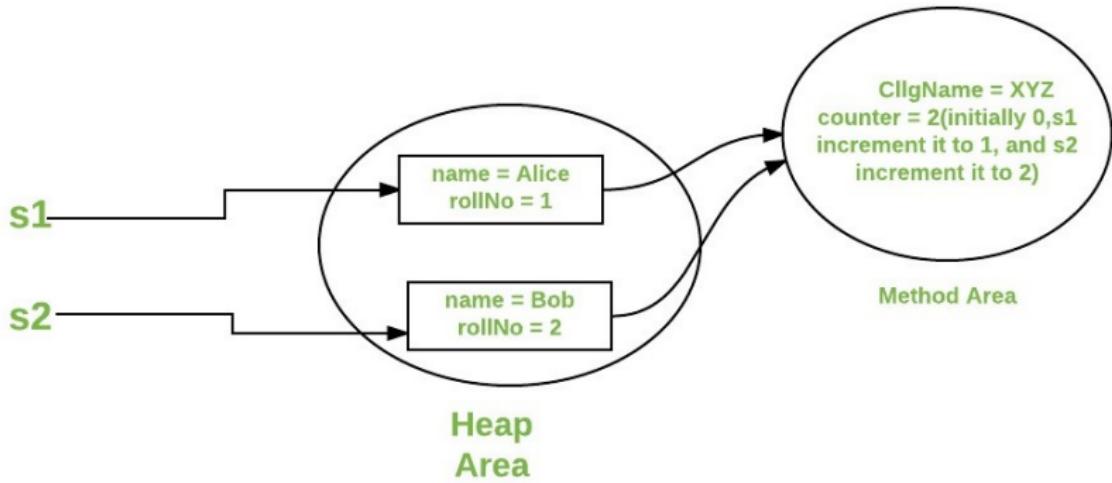
## Types of Variables XXI

```
17     {
18         counter++;
19         return counter;
20     }
21     // static method
22     static void setClgg(String name)
23     {
24         clggName = name ;
25     }
26     // instance method
27     void getStudentInfo()
28     {
29         System.out.println("name : " + this.name);
30         System.out.println("rollNo : " + this.rollNo);
31         // accessing static variable
32         System.out.println("clggName : " + clggName);
33     }
34 }
35 //Driver class
36 public class StaticDemo
37 {
38     public static void main(String[] args)
39     {
40         // calling static method without instantiating Student class
41         Student.setClgg("XYZ");
```

## Types of Variables XXII

```
42     Student s1 = new Student("Alice");
43     Student s2 = new Student("Bob");
44     s1.getStudentInfo();
45     s2.getStudentInfo();
46 }
47 }
48 Output:
49
50 name : Alice
51 rollNo : 1
52 cllgName : XYZ
53 name : Bob
54 rollNo : 2
55 cllgName : XYZ
```

## Types of Variables XXIII



- ④ Static nested classes : We can not declare top-level class with a static modifier, but can declare nested classes as static. Such type of classes are called Nested static classes.

# CS204: Object Oriented Programming Concepts

**Sachchida Nand Chaurasia**  
Assistant Professor

Department of Computer Science  
Banaras Hindu University  
Varanasi

Email id: [snchaurasia@bhu.ac.in](mailto:snchaurasia@bhu.ac.in), [sachchidanand.mca07@gmail.com](mailto:sachchidanand.mca07@gmail.com)



January 19, 2021

## Java Operator I

You will learn about different types of operators in Java, their syntax and how to use them.

- ✓ Operators are symbols that perform operations on variables and values.
- ✓ For example, + is an operator used for addition, while \* is also an operator used for multiplication.
- Operators in Java can be classified into 6 types:
  - ① Arithmetic Operators
  - ② Assignment Operators

# Java Operator II

- ③ Relational Operators
- ④ Logical Operators
- ⑤ Unary Operators
- ⑥ Bitwise Operators

## Java Operator III

- ① Java Arithmetic Operators: Arithmetic operators are used to perform arithmetic operations on variables and data.

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo Operation (Remainder after division)

Example:

## Java Operator IV

```
1 a + b;  
2 a - b;  
3 a * b;  
4 a / b;  
5 a % b;
```

```
1 public class Main  
2 {  
3     public static void main(String[] args)  
4     {  
5         // declare variables  
6         int a = 12, b = 5;  
7         // addition operator
```

## Java Operator V

```
8  System.out.println("a + b = " + (a + b));  
9  // subtraction operator  
10 System.out.println("a - b = " + (a - b));  
11 // multiplication operator  
12 System.out.println("a * b = " + (a * b));  
13 // division operator  
14 System.out.println("a / b = " + (a / b));  
15 // modulo operator  
16 System.out.println("a % b = " + (a % b));  
17 }  
18 }
```

# Java Operator VI

## Output:

```
1 a + b = 17
2 a - b = 7
3 a * b = 60
4 a / b = 2
5 a % b = 2
```

## Java Operator VII

- ② Java Assignment Operators: Assignment operators are used in Java to assign values to variables.

Operator	Example	Equivalent to
=	a = b;	a = b;
+=	a += b;	a = a + b;
-=	a -= b;	a = a - b;
*=	a *= b;	a = a * b;
/=	a /= b;	a = a / b;
%=	a %= b;	a = a % b;

# Java Operator VIII

```
1 class Main
2 {
3     public static void main(String[] args)
4     {
5         // create variables
6         int a = 4;
7         int var;
8         // assign value using =
9         var = a;
10        System.out.println("var using =: " + var);
11        // assign value using +=
12        var += a;
13        System.out.println("var using +=: " + var);
```

## Java Operator IX

```
14     // assign value using *=  
15     var *= a;  
16     System.out.println("var using *=: " + var);  
17 }  
18 }
```

### Output:

```
1 var using =: 4  
2 var using +=: 8  
3 var using *=: 32
```

- ③ Java Relational Operators: Relational operators are used to check the relationship between two operands.

## Java Operator X

```
1 // check is a is less than b  
2 a < b;
```

Here, < operator is the relational operator. It checks if a is less than b or not.

It returns either true or false.

## Java Operator XI

Operator	Description	Example
<code>==</code>	Is Equal To	<code>3 == 5</code> returns <b>False</b>
<code>!=</code>	Not Equal To	<code>3 != 5</code> returns <b>True</b>
<code>&gt;</code>	Greater Than	<code>3 &gt; 5</code> returns <b>False</b>
<code>&lt;</code>	Less Than	<code>3 &lt; 5</code> returns <b>True</b>
<code>&gt;=</code>	Greater Than or Equal To	<code>3 &gt;= 5</code> returns <b>False</b>
<code>&lt;=</code>	Less Than or Equal To	<code>3 &lt;= 5</code> returns <b>False</b>

## Java Operator XII

```
1 public class Main
2 {
3     public static void main(String[] args)
4     {
5         // create variables
6         int a = 7, b = 11;
7         // value of a and b
8         System.out.println("a is " + a + " and b is " + b);
9         // == operator
10        System.out.println(a == b); // false
11        // != operator
12        System.out.println(a != b); // true
13        // > operator
```

## Java Operator XIII

```
14     System.out.println(a > b); // false
15     // < operator
16     System.out.println(a < b); // true
17     // >= operator
18     System.out.println(a >= b); // false
19     // <= operator
20     System.out.println(a <= b); // true
21 }
22 }
```

## Java Operator XIV

- ④ Java Logical Operators: Logical operators are used to check whether an expression is **True** or **False**. They are used in decision making.

Operator	Example	Meaning
&& (Logical AND)	expression1 && expression2	<b>True</b> only if both expression1 and expression2 are <b>True</b>
(Logical OR)	expression1    expression2	<b>True</b> if either expression1 or expression2 is <b>True</b>
! (Logical NOT)	!expression	<b>True</b> if expression is <b>False</b> and vice versa

## Java Operator XV

```
1 public class Main
2 {
3     public static void main(String[] args)
4     {
5         // && operator
6         System.out.println((5 > 3) && (8 > 5)); // true
7         System.out.println((5 > 3) && (8 < 5)); // false
8         // || operator
9         System.out.println((5 < 3) || (8 > 5)); // true
10        System.out.println((5 > 3) || (8 < 5)); // true
11        System.out.println((5 < 3) || (8 < 5)); // false
12        // ! operator
13        System.out.println(!(5 == 3)); // true
```

## Java Operator XVI

```
14     System.out.println(!(5 > 3)); // false
15 }
16 }
```

- ⑤ Java Unary Operators: Unary operators are used with only one operand. For example, `++` is a unary operator that increases the value of a variable by 1. That is, `++5` will return 6.

## Java Operator XVII

Operator	Meaning
+	Unary plus: not necessary to use since numbers are positive without using it
-	Unary minus: inverts the sign of an expression
++	Increment operator: increments value by 1
--	Decrement operator: decrements value by 1
!	Logical complement operator: inverts the value of a boolean

```
1 int num = 5;  
2 // increase num by 1  
3 ++num;
```

## Java Operator XVIII

```
1 public class Main
2 {
3     public static void main(String[] args)
4     {
5         // declare variables
6         int a = 12, b = 12;
7         int result1, result2;
8         // original value
9         System.out.println("Value of a: " + a);
10        // increment operator
11        result1 = ++a;
12        System.out.println("After increment: " + result1);
13        System.out.println("Value of b: " + b);
```

# Java Operator XIX

```
14     // decrement operator  
15     result2 = - -b;  
16     System.out.println("After decrement: " + result2);  
17 }  
18 }
```

## Output:

```
1 Value of a: 12  
2 After increment: 13  
3 Value of b: 12  
4 After decrement: 11
```

## Java Operator XX

```
1 public class Operator
2 {
3     public static void main(String[] args)
4     {
5         int var1 = 5, var2 = 5;
6         // var1 is displayed
7         // Then, var1 is increased to 6.
8         System.out.println(var1++);
9         // var2 is increased to 6
10        // Then, var2 is displayed
11        System.out.println(++var2);
12    }
13 }
```

## Java Operator XXI

Output: ?

- ⑥ Java Bitwise Operators: Bitwise operators in Java are used to perform operations on individual bits.

1 Bitwise complement Operation of 35

2  $35 = 00100011$  (In Binary)

3  $\sim 00100011$

4 -----

5  $11011100 = 220$  (In decimal)

6 Here,  $\sim$  is a bitwise operator.

It inverts the value of each bit (0 to 1 and 1 to 0).

## Java Operator XXII

Operator	Description
<code>~</code>	Bitwise Complement
<code>&lt;&lt;</code>	Left Shift
<code>&gt;&gt;</code>	Right Shift
<code>&gt;&gt;&gt;</code>	Unsigned Right Shift
<code>&amp;</code>	Bitwise AND
<code>^</code>	Bitwise exclusive OR

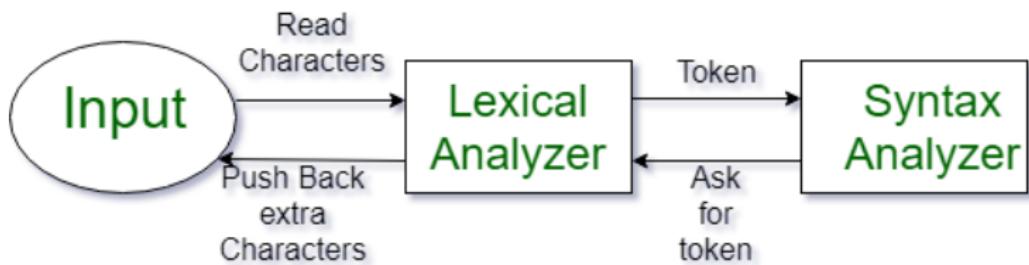
## C Operator Precedence and Associativity I

### Introduction of Lexical Analyzer:

- ✓ Lexical Analysis is the first phase of the compiler also known as a scanner. It converts the High level input program into a sequence of Tokens.
- ✓ The lexical analyzer is the part of the compiler that detects the token of the program and sends it to the syntax analyzer.
- ✓ Token is the smallest entity of the code, it is either a keyword, identifier, constant, string literal, symbol.

Examples of different types of tokens in C.

## C Operator Precedence and Associativity II



## C Operator Precedence and Associativity III

### What is a token?

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

Example of tokens:

- Type token (id, number, real, . . . )
- Punctuation tokens (IF, void, return, . . . )
- Alphabetic tokens (keywords)
- Keywords; Examples-for, while, if etc.

## C Operator Precedence and Associativity IV

- Identifier; Examples-Variable name, function name, etc.
- Operators; Examples '+', '++', '-' etc.
- Separators; Examples ',', ';' etc

### Example of Non-Tokens:

1 Comments, preprocessor directive, macros, blanks, tabs, newline, etc.

## C Operator Precedence and Associativity V

**Lexeme:** The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme.  
eg- “float”, “abs\_zero\_Kelvin”, “=”, “-”, “273”, “;” .

### How Lexical Analyzer functions

- ① It always matches the longest character sequence.
- ② Tokenization i.e. Dividing the program into valid tokens.
- ③ Remove white space characters.

## C Operator Precedence and Associativity VI

- ④ Remove comments.
  - ⑤ It also provides help in generating error messages by providing row numbers and column numbers.
- ✓ The lexical analyzer(scanner) identifies the error with the help of the automation machine and the grammar of the given language on which it is based like C, C++, and gives row number and column number of the error. Suppose we pass a statement through lexical analyzer–

## C Operator Precedence and Associativity VII

```
1 a = b + c ; //It will generate token sequence like this:  
2 id= id + id; //Where each id refers to it's variable in the  
symbol table referencing all details
```

For example, consider the program:

```
1 int a=5; // int a = 5 ; just for understanding.  
2 Tokens:  
3 |int| |a| |=| |5| |; |
```

Another example:

## C Operator Precedence and Associativity VIII

```
1 int main()
2 {
3     // 2 variables
4     int a, b;
5     a = 10;
6     return 0;
7 }
```

---

9 Tokens:

```
10 int
11 main
12 (
13 )
```

# C Operator Precedence and Associativity IX

```
14 {  
15     int  
16         a  
17         ,  
18         b  
19         ;  
20     a  
21     =  
22     10  
23     ;  
24     return  
25     0  
26     ;
```

# C Operator Precedence and Associativity X

27 }

```
1 //How many tokens?  
2  
3 printf("BSCPMKSMK");
```

```
1 int main()  
2 {  
3     int a = 10, b = 20;  
4     printf("sum is :%d",a+b);  
5     return 0;  
6 }
```

```
7 -----  
8 //How many tokens?
```

# C Operator Precedence and Associativity XI

```
1 int max(int i);  
2 -----  
3 //Count number of tokens :
```

## C Operator Precedence and Associativity XII

- Lexical analyzer first read int and finds it to be valid and accepts as token
- max is read by it and found to be a valid function name after reading (
- int is also a token , then again i as another token and finally ;

Answer: Total number of tokens 7:

```
|int| |max| |(| |int| |i| |)| |;| |
```

## C Operator Precedence and Associativity XIII

```
1 //Count number of tokens :  
2  
3 printf("i = %d, &i = %x", i, &i);
```

✓ Identification valid a token.

```
1 int a=10;  
2 int a=10;  
3 -----  
4 //Number of valid tokens: ????
```

### Lvalues and Rvalues in C:

There are two kinds of expressions in C -

**Ivalue** - Expressions that refer to a *memory location* are called "lvalue" expressions. An lvalue may appear as either the left-hand or right-hand side of an assignment operator (=).

✓ lvalue often represents as identifier.

lvalue(left value): simply means an object that has an identifiable location in memory (i.e. having an address).

## C Operator Precedence and Associativity XV

- In any assignment statement "lvalue" must have the capacity to hold the data
- lvalue **must be a variable** because they have the capability to store the data.
- lvalue cannot be a function, expression ( $a+b$ ) or a constant (like 3, 4 etc).
- rvalue(right value): simply means an object that has no identifiable location in memory.
- Anything which is capable of returning a constant expression or value.

## C Operator Precedence and Associativity XVI

→ Expression like  $(a+b)$  will return some constant value.  
For example:  $a++$ ; is equivalent to  $a = a+1$ ; here we have both lvalue and rvalue. before  $=$ ,  $a$  is lvalue and after  $= a+1$  is rvalue.

Take our example  $a=b++$ ; convert it into normal expression  
 $a=b=b + 1;$

Take our example  $(a=b)++$ ; convert it into normal expression  $(a=b) = (a=b) + 1;$

```
1 int g = 20; // valid statement
2 10 = 20; // invalid statement; would generate compile-time error.
```

## C Operator Precedence and Associativity XVII

```
1 // declare a an object of type 'int'  
2 int a;  
3 // a is an expression referring to an 'int' object as l-value  
4 a = 1;  
5 int b = a; // Ok, as l-value can appear on right  
6 // Switch the operand around '=' operator  
7 9 = a;  
8 // Compilation error: as assignment is trying to change the  
   value of assignment operator
```

## C Operator Precedence and Associativity XVIII

**rvalue** - The term rvalue refers to a *data value* that is stored at some address in memory.

- ✓ An rvalue is an expression that cannot have a value assigned to it which means an rvalue may appear on the right-hand side but not on the left-hand side of an assignment (=).
- ✓ Variables are lvalues and so they may appear on the left-hand side of an assignment.
- ✓ Numeric literals are rvalues and so they may not be assigned and cannot appear on the left-hand side.

## C Operator Precedence and Associativity XIX

```
1 // declare a, b an object of type 'int'
2 int a = 1, b;
3 a + 1 = b; // Error, left expression is  is not variable(a + 1)
4 // declare pointer variable 'p', and 'q'
5 int *p, *q; // *p, *q are lvalue
6 *p = 1; // valid l-value assignment
7 // below is invalid - "p + 2" is not an l-value  p + 2 = 18;
8 q = p + 5; // valid - "p + 5" is an r-value
9 // Below is valid - dereferencing pointer expression gives an l-
   value
10 *(p + 2) = 18;
11 p = &b;
12 int arr[20]; // arr[12] is an lvalue; equivalent to *(arr+12)
```

## C Operator Precedence and Associativity XX

```
13 // Note: arr itself is also an lvalue
14 struct S
15 {
16     int m;
17
18 }
19 ;
20 struct S obj; // obj and obj.m are lvalues
21 // ptr-> is an lvalue; equivalent to (*ptr).m
22 // Note: ptr and *ptr are also lvalues
23 struct S* ptr = &obj;
```

## Java Operators Precedence and Associativity I

- ✓ Precedence of operators come into picture when in an expression we need to decide which operator will be evaluated first.
- ✓ Operator with higher precedence will be evaluated first.

```
1 int a=1;
2 int b=4;
3 int c;
4 //expression
5 c= a + b;
6 // Which one is correct
7 (c=a) + b or
8 c = (a+b)
```

## Java Operators Precedence and Associativity II

\*Larger number means higher precedence.

Precedence	Operator	Type	Associativity
15	( [] . .	Parentheses Array subscript Member selection	Left to Right
14	++ --	Unary post-increment Unary post-decrement	Left to Right

# Java Operators Precedence and Associativity III

13	++	Unary pre-increment	Right to left
	--	Unary pre-decrement	
	+	Unary plus	
	-	Unary minus	
	!	Unary logical negation	
	~	Unary bitwise complement	
	( type )	Unary type cast	
12	*	Multiplication	Left to right
	/	Division	
	%	Modulus	
11	+	Addition	Left to right
	-	Subtraction	

## Java Operators Precedence and Associativity IV

10	<< >> ">>>	Bitwise left shift Bitwise right shift with sign extension Bitwise right shift with zero extension	Left to right
9	< <= > >= instanceof	Relational less than Relational less than or equal Relational greater than Relational greater than or equal Type comparison (objects only)	Left to right
8	== !=	Relational is equal to Relational is not equal to	Left to right
7	&	Bitwise AND	Left to right
6	^	Bitwise exclusive OR	Left to right

# Java Operators Precedence and Associativity V

5		Bitwise inclusive OR	Left to right
4	&&	Logical AND	Left to right
3		Logical OR	Left to right
2	? :	Ternary conditional	Right to left
1	= += -= *= /= %= 	Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment	Right to left

## Java Operators Precedence and Associativity VI

✓ + and \* operators. \* operator having greater precedence than + operator.

1    2+3\*5;

2     $(2+3)*5=25$  or

3     $2+(3*5)=17$

## Java Operators Precedence and Associativity VII

- ✓ Associativity of operators come into picture when precedence of operators are same and need to decide which operator will be evaluated first. Associativity can be either left-to-right or right-to-left.
- ✓ / and \* operators. Both having same precedence, then associated will come into a picture.

```
1 10/2*5;  
2 //if left-to-right  
3 (10/2)*5=25  
4 //if right-to-left  
5 10/(2*5)=1
```

## Java Operators Precedence and Associativity VIII

- ✓ ()- parenthesis in function calls.
- ✓ Parenthesis() operator having greater precedence than assignment(=) operator.

```
1 int val =fun();  
2 |int| |val| |=| |fun| |(| |)| |;  
3  
4 // if suppose = operator is having greater precedence then,  
   fun will belong to =  
   operator and therefore it will be treated as a variable.  
5 int (val = fun)();  
6
```

## Java Operators Precedence and Associativity IX

```
7 // = operator is having less less precedence as compared to ()
  therefore, ()
  belongs to fun and will be treated as a function.
8 int val = (fun());
```

```
1 //Which function will be called first.
2 int main()
3 {
4     int a;
5     a = MCA() + MSC();
6     printf("\n%d",a);
7     return 0;
8 }
```

## Java Operators Precedence and Associativity X

```
10 int MCA()
11 {
12     printf("MCA");
13     return 1;
14 }
15
16 int MSC()
17 {
18     printf("MSC");
19     return 1;
20 }
21 -----
22 Output: ???
```

## Java Operators Precedence and Associativity XI

23

24 Answer: MCAMSC2 or MSCMCA2.

25 // It is not defined whether MCA() will be called first or  
// whether MSC() will be called. Behaviour is undefined and  
// output is compiler dependent.

26

27 //Here associativity will not come into picture as we have just  
// one operator and which function will be called first is  
// undefined. Associativity will only work when we have more  
// than one operators of same precedence.

### ► Increment ++ and Decrement - - Operator as Prefix and Postfix

- ✓ Precedence of Postfix increment/Decrement operator is greater than Prefix increment/Decrement.
- ✓ Associativity of Postfix is also different from Prefix. Associativity of postfix operators is from left-to-right and that of prefix operators is from right-to-left.
- ✓ Operators with some precedence have same associativity as well.

## Java Operators Precedence and Associativity XIII

- If you use the `++` operator as prefix like: `++var`. The value of var is incremented by 1 then, it returns the value. or means first increment then assign it to another variable.
- If you use the `++` operator as postfix like: `var++`. The original value of var is returned first then, var is incremented by 1. or means first assign it to another variable then increment.

## Java Operators Precedence and Associativity XIV

- The `--` operator works in a similar way like the `++` operator except it decreases the value by 1.

✓ you cannot use rvalue before or after increment/decrement operator.

Example:

`(a+b)++;` Error

`++(a+b);` Error.

Error: lvalue required as increment operator(compiler is expecting a variable as an increment operand but we are providing an expression `(a+b)` which does not have the

## Java Operators Precedence and Associativity XV

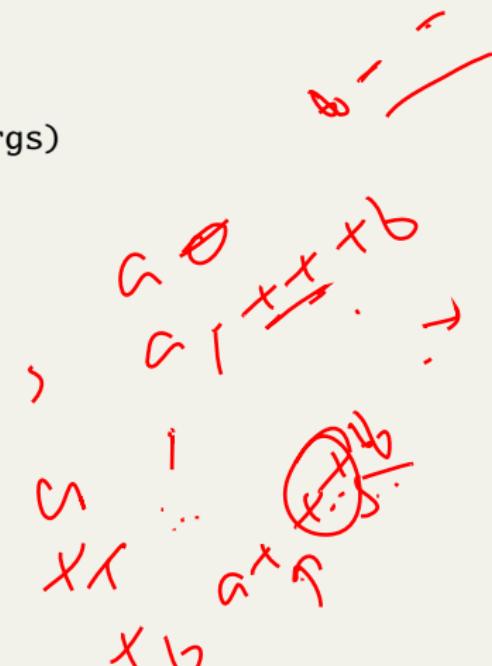
capability to store data). Because  $(a+b)$  is rvalue.  $(a+b)$  is an expression or you can say it is value not an operator.

```
1 int main()
2 {
3     int x = 1;
4     int y=0;
5     x=y++;
6     // (x=y) = (x=y) +1;
7     scanf("%d",&y);
8     printf("%d\n%d",x,y);
9     // return 0;
10 }
```

## Java Operators Precedence and Associativity XVI

✓ Unary operator must be associated with a valid operand.

```
1 public class Precedence  
2 {  
3     public static void main(String[] args)  
4     {  
5         int a = 10, b = 5, c = 1;  
6         System.out.println(a+++b);  
7         System.out.println(a+++ b);  
8         System.out.println(a++ + b);  
9         System.out.println(c.+++b);  
10        System.out.println(a.+++b);  
11        System.out.println(a+ +++b);  
12    }
```



# Java Operators Precedence and Associativity XVII

13

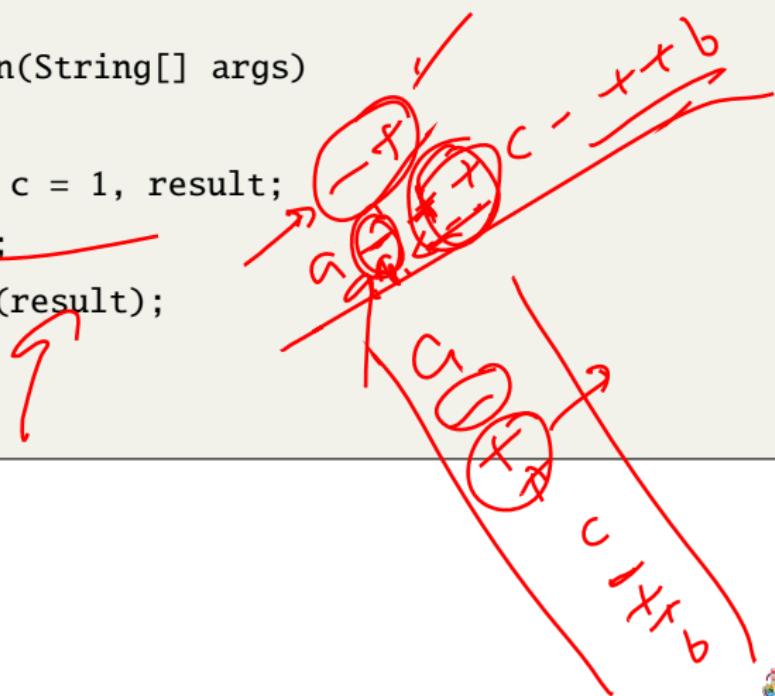
}

## Java Operators Precedence and Associativity XVIII

```
1 a+++b;  
2 //Valid tokens in line number 5:  
3 |a| |++| |+| |b| |; |  
4  
5 //Make valid syntax for post-increment and pre-increment  
6 // Unary operator must be associated with a valid operand.  
7 //++ will be associated with a  
8 a++ /  
9 +  
10 b /  
11 -----  
12 a++ + b;
```

# Java Operators Precedence and Associativity XIX

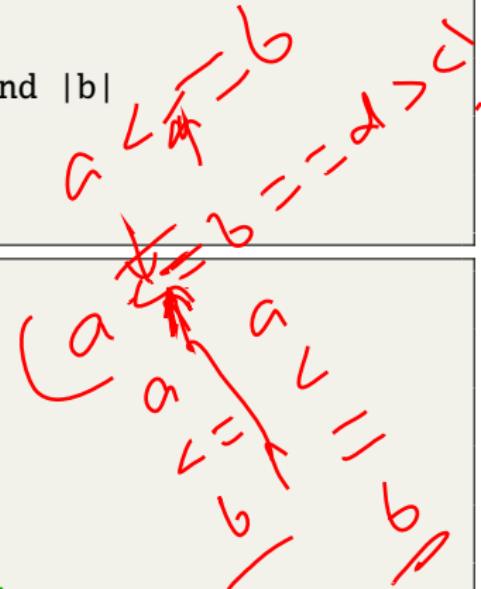
```
1 public class Precedence  
2 {  
3     public static void main(String[] args)  
4     {  
5         int a = 10, b = 5, c = 1, result;  
6         result = a-++c-++b;  
7         System.out.println(result);  
8     }  
9 }
```



# Java Operators Precedence and Associativity XX

```
1 result = a-++c-++b;  
2 //Valid tokens in line number 7:  
3 |result|, |=|, |a|, |-|, |++|, |-|, |++| and |b|  
4  
5 //Make valid syntax for post-increment
```

```
1 public class Precedence  
2 {  
3     public static void main(String[] args)  
4     {  
5         int a = 10, b = 15, c = 20, d=25;  
6         //int a = 17, b = 15, c = 20, d=25;  
7         if(a<= b == d > c)
```



## Java Operators Precedence and Associativity XXI

```
8    {
9        System.out.println("TRUE");
10    }
11 else
12 {
13     System.out.println("FALSE");
14 }
15 }
16 }
```

## Java Operators Precedence and Associativity XXII

```
1 |a| |<| |=| |b| |=| |=| |d| |>| |c|
2 OR
3 |a| |<=| |b| |==| |d| |>| |c|
4
5 |<=| --> Precedence 9
6 |==| --> Precedence 8
7 |>| --> Precedence 9
8
9 ((a<=b) == (d>c))
10
11 (1 == 1)
```

# CS204:Object Oriented Programming

## Concepts

**Sachchida Nand Chaurasia**  
Assistant Professor

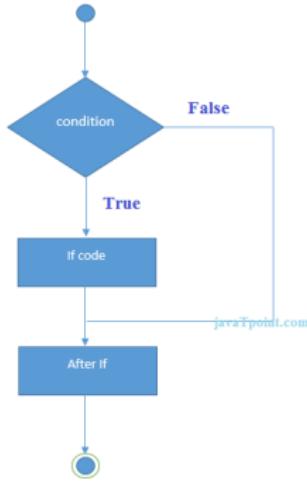
Department of Computer Science  
Banaras Hindu University  
Varanasi

Email id: [snchaurasia@bhu.ac.in](mailto:snchaurasia@bhu.ac.in), [sachchidanand.mca07@gmail.com](mailto:sachchidanand.mca07@gmail.com)



February 11, 2021

# If...else I



javatpoint.com

```
1 if(condition)
2 {
3     //code to be executed
4 }
```

## If...else II

```
1 //Java Program to demonstrate the use of if statement.  
2 public class IfExample  
3 {  
4     public static void main(String[] args)  
5     {  
6         //defining an 'age' variable  
7         int age=20;  
8         //checking the age  
9         if(age>18)  
10        {  
11             System.out.print("Age is greater than 18");  
12         }  
13     }  
14 }
```

### if-else statement:

- ✓ The Java if-else statement also tests the condition. It executes the if block if condition is true otherwise else block is executed.

## If...else III

```
1 if(condition)
2 {
3     //code if condition is true
4 }
5 else
6 {
7     //code if condition is false
8 }

1 //A Java Program to demonstrate the use of if-else statement.
2 //It is a program of odd and even number.
3 public class IfElseExample
4 {
5     public static void main(String[] args)
6     {
7         //defining a variable
8         int number=13;
9         //Check if the number is divisible by 2 or not
10        if(number%2==0)
11        {
12            System.out.println("even number");
13        }
14        else
15        {
16            System.out.println("odd number");
```

## If...else IV

```
17     }
18 }
19 }
```

### ✓ Leap Year Example:

A year is leap, if it is divisible by 4 and 400. But, not by 100.

```
1 public class LeapYearExample
2 {
3     public static void main(String[] args)
4     {
5         int year=2020;
6         if(((year % 4 ==0) && (year % 100 !=0)) || (year % 400==0))
7         {
8             System.out.println("LEAP YEAR");
9         }
10        else
11        {
12            System.out.println("COMMON YEAR");
13        }
14    }
15 }
```

# Using Ternary Operator:

✓ We can also use ternary operator (? :) to perform the task of if...else statement. It is a shorthand way to check the condition. If the condition is true, the result of ? is returned. But, if the condition is false, the result of : is returned.

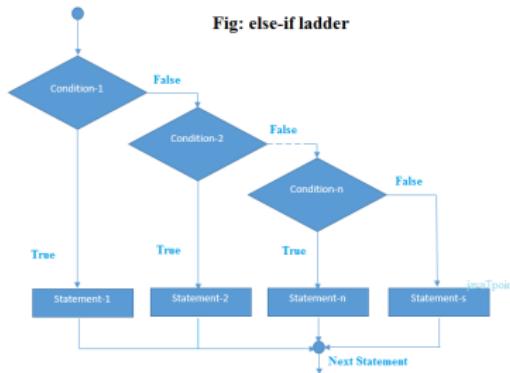
```
1 public class IfElseTernaryExample
2 {
3     public static void main(String[] args)
4     {
5         int number=13;
6         //Using ternary operator
7         String output=(number%2==0)?"even number":"odd number";
8         System.out.println(output);
9     }
10 }
```

# Java if-else-if ladder Statement:

The if-else-if ladder statement executes one condition from multiple statements.

```
1  if(condition1)
2  {
3      //code to be executed if condition1 is true
4  }
5  else if(condition2)
6  {
7      //code to be executed if condition2 is true
8  }
9  else if(condition3)
10 {
11     //code to be executed if condition3 is true
12 }
13 ...
14 else
15 {
16     //code to be executed if all the conditions are false
17 }
```

## If...else VII



```
1 //Java Program to demonstrate the use of If else-if ladder.
2 //It is a program of grading system for fail, D grade, C grade, B grade, A grade and A+.
3 public class IfElseIfExample
4 {
5     public static void main(String[] args)
6     {
7         int marks=65;
8
9         if(marks<50)
10        {
11            System.out.println("fail");
12        }
13        else if(marks>=50 && marks<60)
```

## If...else VIII

```
14
15     {
16         System.out.println("D grade");
17     }
18     else if(marks>=60 && marks<70)
19     {
20         System.out.println("C grade");
21     }
22     else if(marks>=70 && marks<80)
23     {
24         System.out.println("B grade");
25     }
26     else if(marks>=80 && marks<90)
27     {
28         System.out.println("A grade");
29     }
30     else if(marks>=90 && marks<100)
31     {
32         System.out.println("A+ grade");
33     }
34     else
35     {
36         System.out.println("Invalid!");
37     }
38 }
```

## If...else IX

✓ Program to check POSITIVE, NEGATIVE or ZERO:

```
1 public class PositiveNegativeExample
2 {
3     public static void main(String[] args)
4     {
5         int number=-13;
6         if(number>0)
7         {
8             System.out.println("POSITIVE");
9         }
10        else if(number<0)
11        {
12            System.out.println("NEGATIVE");
13        }
14        else
15        {
16            System.out.println("ZERO");
17        }
18    }
19 }
```

### Java Nested if statement:

The nested if statement represents the if block within another if block. Here, the inner if block condition executes only when outer if block condition is true.

```
1  if(condition)
2  {
3      //code to be executed
4      if(condition)
5      {
6          //code to be executed
7      }
8 }
```

✓ Example:

## If...else XI

```
1 //Java Program to demonstrate the use of Nested If Statement.
2 public class JavaNestedIfExample
3 {
4     public static void main(String[] args)
5     {
6         //Creating two variables for age and weight
7         int age=20;
8         int weight=80;
9         //applying condition on age and weight
10        if(age>=18)
11        {
12            if(weight>50)
13            {
14                System.out.println("You are eligible to donate blood");
15            }
16        }
17    }
18 }
19 }
```

# Java while and do-while loop I

```
1 while(condition)
2 {
3     //code to be executed
4 }
```

'condition' must be a boolean value. We cannot pass any other value.

```
1 public class WhileExample
2 {
3     public static void main(String[] args)
4     {
5         int i=1;
6         while(i<=10)
7         {
8             System.out.println(i);
9             i++;
10        }
11    }
12 }
```

## do-while loop:

## Java while and do-while loop II

```
1  do
2  {
3      //code to be executed
4  }
5  while(condition);
```

### ✓ Example:

```
1  public class DoWhileExample
2  {
3      public static void main(String[] args)
4      {
5          int i=1;
6          do
7          {
8              System.out.println(i);
9              i++;
10         }
11         while(i<=10);
12     }
13 }
```

## Java for loop I

✓ A simple for loop is the same as C/C++. We can initialize the variable, check condition and increment/decrement value. It consists of four parts:

- ① Initialization: It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.
- ② Condition: It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either true or false. It is an optional condition.
- ③ Statement: The statement of the loop is executed each time until the second condition is false.

## Java for loop II

- ④ Increment/Decrement: It increments or decrements the variable value. It is an optional condition.

Syntax:

```
1 for(initialization; condition; incr/decr)
2 {
3     //statement or code to be executed
4 }
```

'condition' must be a boolean value. We cannot pass any other value.

## Java for loop III

```
1 //Java Program to demonstrate the example of for loop
2 //which prints table of 1
3 public class ForExample
4 {
5     public static void main(String[] args)
6     {
7         //Code of Java for loop
8         for(int i=1; i<=10; i++)
9         {
10             System.out.println(i);
11         }
12     }
13 }
```

## Java for loop IV

```
1 // Calculate the sum of all elements of an array
2
3 public class Main
4 {
5     public static void main(String[] args)
6     {
7
8         // an array of numbers
9         int[] numbers =
10        {
11            3, 4, 5, -5, 0, 12
12        }
13        ;
14        int sum = 0;
15
16        // iterating through each element of the array
17        for (int number: numbers)
18        {
19            sum += number;
20        }
21
22        System.out.println("Sum = " + sum);
23    }
24}
```

# Java for loop V

```
1 public class Main
2 {
3     public static void main(String[] args)
4     {
5
6         char[] vowels =
7         {
8             'a', 'e', 'i', 'o', 'u'
9         }
10
11
12         // iterating through an array using a for loop
13         for (int i = 0; i < vowels.length; ++ i)
14         {
15             System.out.println(vowels[i]);
16         }
17     }
18 }
```

## Java for loop VI

### ✓ Using for-each Loop:

```
1 for(dataType item : array)
2 {
3     ...
4 }
```

Here,

- array - an array or a collection
- item - each item of array/collection is assigned to this variable
- dataType - the data type of the array/collection

## Java for loop VII

```
1 public class Main
2 {
3     public static void main(String[] args)
4     {
5
6         char[] vowels =
7         {
8             'a', 'e', 'i', 'o', 'u'
9         }
10    ;
11
12    // iterating through an array using the for-each loop
13    for (char item: vowels)
14    {
15        System.out.println(item);
16    }
17}
18}
```

# while, do-while and for loop I

Comparison	for loop	while loop	do while loop
Introduction	The Java for loop is a control flow statement that iterates a part of the programs multiple times.	The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given boolean condition.	The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends upon the given boolean condition.
When to use	If the number of iteration is fixed, it is recommended to use for loop.	If the number of iteration is not fixed, it is recommended to use while loop.	If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use the do-while loop.
Syntax	<code>for(init;condition;incr/decr){ // code to be executed }</code>	<code>while(condition){ //code to be executed }</code>	<code>do{ //code to be executed }while(condition);</code>
Example	<code>//for loop      for(int i=1;i&lt;=10;i++){                     System.out.println(i); }</code>	<code>//while loop    int i=1;                   while(i&lt;=10){                     System.out.println(i);                     i++;                   }</code>	<code>//do-while loop int i=1; do{                   System.out.println(i);                   i++;                   }while(i&lt;=10);</code>
Syntax for infinitive loop	<code>for(;){ //code to be executed }</code>	<code>while(true){ //code to be executed }</code>	<code>do{ //code to be executed }while(true);</code>

# CS204:Object Oriented Programming

## Concepts

**Sachchida Nand Chaurasia**  
Assistant Professor

Department of Computer Science  
Banaras Hindu University  
Varanasi

Email id: [snchaurasia@bhu.ac.in](mailto:snchaurasia@bhu.ac.in), [sachchidanand.mca07@gmail.com](mailto:sachchidanand.mca07@gmail.com)



February 10, 2021

## Non-Primitive Data Types or Reference Data Types I

- ✓ The Reference Data Types will contain a memory address of variable value because the reference types won't store the variable value directly in memory.
- ✓ They are *arrays, strings, objects*.

### Arrays in Java:

An array is a group of like-typed variables that are referred to by a common name. Arrays in Java work differently than they do in C/C++. Following are some important points about Java arrays.

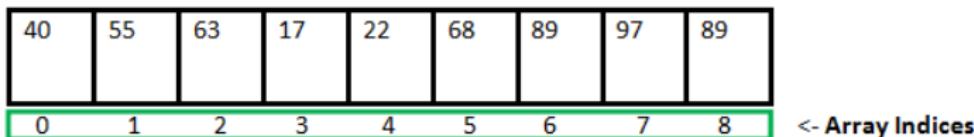
- In Java all arrays are dynamically allocated.
- Since arrays are objects in Java, we can find their length using the object property length. This is different from C/C++ where we find length using sizeof.

## Non-Primitive Data Types or Reference Data Types II

- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered and each have an index beginning from 0.
- Java array can be also be used as a static field, a local variable or a method parameter.
- The size of an array must be specified by an int value and not long or short.
- The direct superclass of an array type is Object.
- Every array type implements the interfaces Cloneable and java.io.Serializable.

## Non-Primitive Data Types or Reference Data Types III

- ✓ Array can contain primitives (int, char, etc.) as well as object (or non-primitive) references of a class depending on the definition of the array.
- ✓ In case of primitive data types, the actual values are stored in contiguous memory locations.
- ✓ In case of objects of a class, the actual objects are stored in heap segment.



**Array Length = 9**

**First Index = 0**

**Last Index = 8**

### Creating, Initializing, and Accessing an Array

#### One-Dimensional Arrays :

The general form of a one-dimensional array declaration is:

```
1 type var-name[];  
2 OR  
3 type[] var-name;
```

- ✓ An array declaration has two components: the type and the name.  
*type* declares the element type of the array.
- ✓ The element type determines the data type of each element that comprises the array.
- ✓ Like an array of integers, we can also create an array of other primitive data types like char, float, double, etc. or user-defined data types (objects of a class).

## Non-Primitive Data Types or Reference Data Types V

- ✓ Thus, the element type for the array determines what type of data the array will hold.

```
1 // both are valid declarations
2 int intArray[];
3 or int[] intArray;
4
5 byte byteArray[];
6 short shortsArray[];
7 boolean booleanArray[];
8 long longArray[];
9 float floatArray[];
10 double doubleArray[];
11 char charArray[];
12
13
14 Object[] ao, // array of Object
15 Collection[] ca; // array of Collection of unknown type
```

## Non-Primitive Data Types or Reference Data Types VI

- ✓ Although the first declaration above establishes the fact that intArray is an array variable, no actual array exists.
- ✓ It merely tells the compiler that this variable (intArray) will hold an array of the integer type.
- ✓ To link intArray with an actual, physical array of integers, you must allocate one using *new* and assign it to intArray.

### Instantiating an Array in Java:

- ✓ When an array is declared, only a reference of array is created.
- ✓ Obtaining an array is a two-step process. First, you must declare a variable of the desired array type. Second, you must allocate the memory that will hold the array, using new, and assign it to the array variable. Thus, in Java all arrays are **dynamically allocated**.

## Non-Primitive Data Types or Reference Data Types VII

- ✓ To actually create or give memory to array, you create an array like this: The general form of new as it applies to one-dimensional arrays appears as follows:

```
1 //Instantiating an Array in Java
2 var-name = new type [size];
3 //Here, type specifies the type of data being allocated, size specifies the number of elements in the array,
   and var-name is the name of array variable that is linked to the array. That is,
   to use new to allocate an array, you must specify the type and number of elements to allocate.
4 -----
5 int[] anArray;
6 anArray = new int[10];
7 OR
8
9 int[] intArray = new int[20]; // combining both statements in one
```

- ✓ The elements in the array allocated by new will automatically be initialized to *zero* (for numeric types), *false* (for boolean), or *null* (for reference types).

Below are the default assigned values.

## Non-Primitive Data Types or Reference Data Types VIII

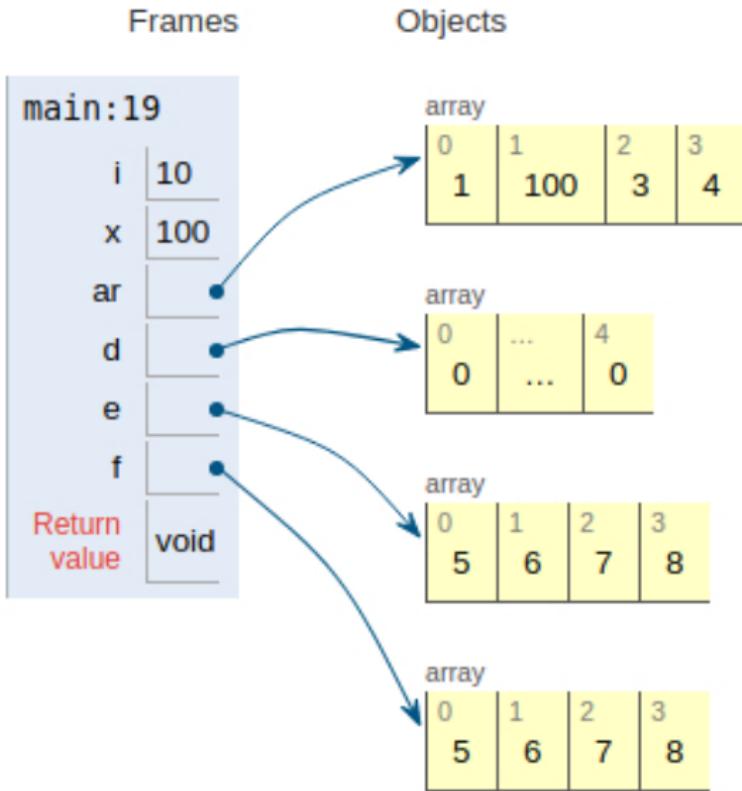
```
1 boolean : false
2 int : 0
3 double : 0.0
4 String : null
5 User Defined Type : null

1 import java.io.*;
2 public class GFG
3 {
4     public static void main(String args[]) throws IOException
5     {
6         int i, x;
7         i=10;
8         x=i;
9         x=100;
10        int ar[] =
11        {
12            1, 2, 3, 4
13        }
14        ;
15        int[] d=ar;
16        d[1]=100;
17        d =new int[5];
18        int[] e=
19        {
```

## Non-Primitive Data Types or Reference Data Types IX

```
20          5,6,7,8
21      }
22      ;
23      int f[]=
24      {
25          5,6,7,8
26      }
27      ; //even values are same but pointing different object
28
29  }
30 }
```

## Non-Primitive Data Types or Reference Data Types X



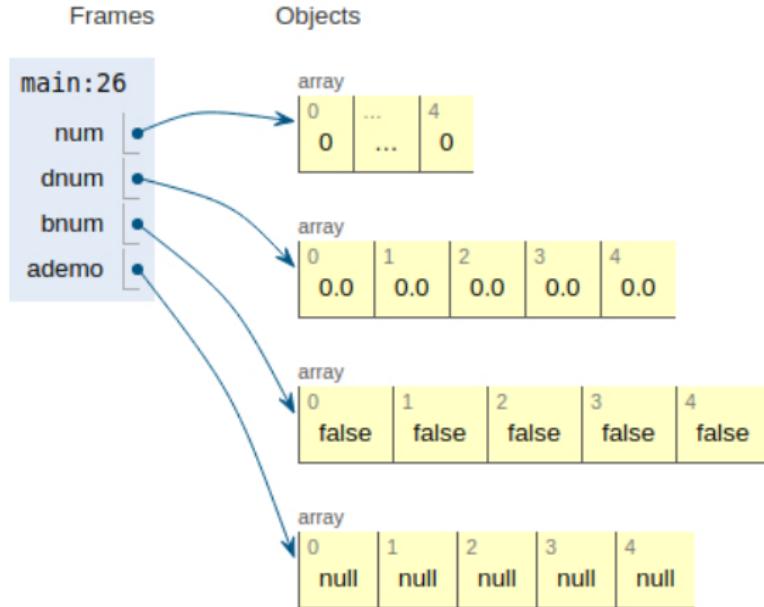
# Non-Primitive Data Types or Reference Data Types XI

```
1 // Java program to demonstrate default values of array elements
2 public class ArrayDemo
3 {
4     public static void main(String[] args)
5     {
6         System.out.println("\n\nInteger array default values:");
7         int num[] = new int[5];
8         for (int val : num)
9             System.out.print(val + " ");
10
11        System.out.println("\n\nDouble array default values:");
12        double dnum[] = new double[5];
13        for (double val : dnum)
14            System.out.print(val + " ");
15
16        System.out.println("\n\nBoolean array default values:");
17        boolean bnum[] = new boolean[5];
18        for (boolean val : bnum)
19            System.out.print(val + " ");
20
21        System.out.println("\n\nReference Array default values:");
22        ArrayDemo ademo[] = new ArrayDemo[5];
23        for (ArrayDemo val : ademo)
24            System.out.print(val + " ");
25    }
```

## Non-Primitive Data Types or Reference Data Types XII

```
26 }  
27 -----  
28 Output:  
29 Integer array default values:  
30 0 0 0 0 0  
31  
32 Double array default values:  
33 0.0 0.0 0.0 0.0 0.0  
34  
35 Boolean array default values:  
36 false false false false false  
37  
38 Reference Array default values:  
39 null null null null null
```

# Non-Primitive Data Types or Reference Data Types XIII



## Non-Primitive Data Types or Reference Data Types XIV

```
1 class ArrayDemo
2 {
3     public static void main(String[] args)
4     {
5         // declares an array of integers
6         int[] anArray;
7
8         // allocates memory for 10 integers
9         anArray = new int[10];
10
11        // initialize first element
12        anArray[0] = 100;
13        // initialize second element
14        anArray[1] = 200;
15        // and so forth
16        anArray[2] = 300;
17        anArray[3] = 400;
18        anArray[4] = 500;
19        anArray[5] = 600;
20        anArray[6] = 700;
21        anArray[7] = 800;
22        anArray[8] = 900;
23        anArray[9] = 1000;
24
25        System.out.println("Element at index 0: " + anArray[0]);
```

## Non-Primitive Data Types or Reference Data Types XV

```
26     System.out.println("Element at index 1: " + anArray[1]);
27     System.out.println("Element at index 2: " + anArray[2]);
28     System.out.println("Element at index 3: " + anArray[3]);
29     System.out.println("Element at index 4: " + anArray[4]);
30     System.out.println("Element at index 5: " + anArray[5]);
31     System.out.println("Element at index 6: " + anArray[6]);
32     System.out.println("Element at index 7: " + anArray[7]);
33     System.out.println("Element at index 8: " + anArray[8]);
34     System.out.println("Element at index 9: " + anArray[9]);
35 }
36 }
37 The output from this program is:
```

```
38
39 Element at index 0: 100
40 Element at index 1: 200
41 Element at index 2: 300
42 Element at index 3: 400
43 Element at index 4: 500
44 Element at index 5: 600
45 Element at index 6: 700
46 Element at index 7: 800
47 Element at index 8: 900
48 Element at index 9: 1000
```

## Non-Primitive Data Types or Reference Data Types XVI

### ✓ Accessing array elements using loop.

```
1 // Java program to iterate over an array using for loop
2 import java.io.*;
3 import java.util.Arrays;
4 import java.lang.reflect.*;
5 public class GFG
6 {
7
8     public static void main(String args[]) throws IOException
9     {
10         int ar[] =
11         {
12             1, 2, 3, 4
13         }
14         ;
15         int i, x;
16
17         // iterating over an array
18         for (i = 0; i < ar.length; i++)
19         {
20
21             // accessing each element of array
22             x = ar[i];
23             System.out.print(x + " ");
24         }
25     }
26 }
```

## Non-Primitive Data Types or Reference Data Types XVII

```
25 System.out.println(Array.get(ar,2));
26 for (i = 0; i < Array.getLength(ar); i++)
27 {
28     // accessing each element of array
29     //x = ar[i];
30     System.out.print(ar[i] + " ");
31 }
32 }
33 }
```

## Non-Primitive Data Types or Reference Data Types XVIII

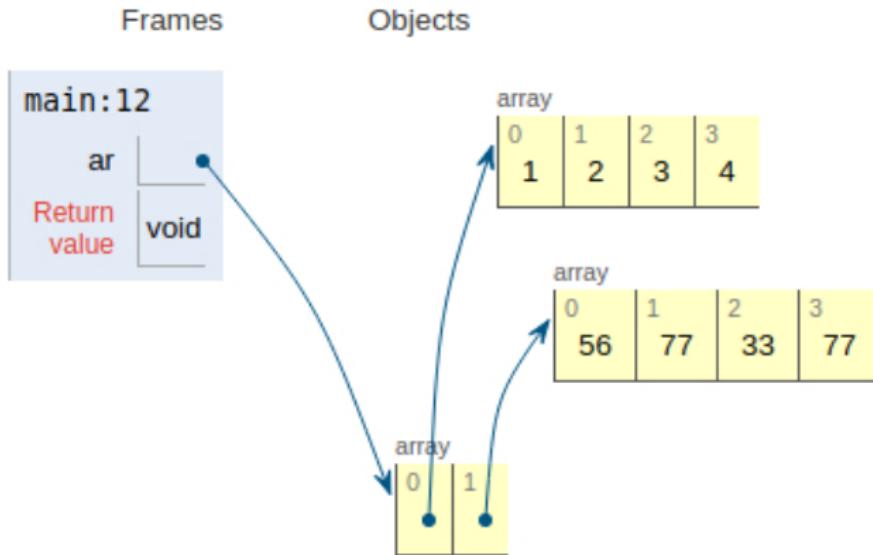
### Multidimensional array

```
1 Two dimensional array:  
2 int[][] twoD_arr = new int[10][20];  
3  
4 Three dimensional array:  
5 int[][][] threeD_arr = new int[10][20][30];  
  
1 import java.io.*;  
2 import java.util.Arrays;  
3 import java.lang.reflect.*;  
4 public class MultiDimArrayDemo  
{  
    6     public static void main(String args[])  
    {  
        8         int ar[][] =  
        9             {  
            10                {  
            11                    1, 2, 3, 4  
            12                }  
            13                ,  
            14                {  
            15                    56, 77, 33, 77  
            16                }  
            17            }  
            18        }  
    }
```

## Non-Primitive Data Types or Reference Data Types XIX

```
19      }
20      ;
21      int[][] ar1 =
22      {
23
24          {
25              31,22, 43, 74
26          }
27          ,
28          {
29              356,377,633,777
30          }
31
32      }
33      ;
34      int i, x;
35  }
36 }
```

## Non-Primitive Data Types or Reference Data Types XX



## Non-Primitive Data Types or Reference Data Types XXI

### ✓ Indirect Method of Declaration:

```
1 class GFG
2 {
3     public static void main(String[] args)
4     {
5
6         int[][] arr = new int[10][20];
7         arr[0][0] = 1;
8
9         System.out.println("arr[0][0] = " + arr[0][0]);
10    }
11 }
```

### ✓ Direct Method of Declaration:

## Non-Primitive Data Types or Reference Data Types XXII

```
1 class GFG
2 {
3     public static void main(String[] args)
4     {
5
6         int[][] arr =
7         {
8
9             {
10                 1, 2
11             }
12             ,
13             {
14                 3, 4
15             }
16
17         }
18 ;
19
20
21         for (int i = 0; i < 2; i++)
22         for (int j = 0; j < 2; j++)
23             System.out.println("arr[" + i + "][" + j + "] = "
24             + arr[i][j]);
25 }
```

## Non-Primitive Data Types or Reference Data Types XXIII

### ✓ Accessing Elements of Two-Dimensional Arrays:

```
1 class GFG
2 {
3     public static void main(String[] args)
4     {
5
6         int[][] arr =
7         {
8
9             {
10                 1, 2
11             }
12             ,
13             {
14                 3, 4
15             }
16
17         }
18         ;
19
20         System.out.println("arr[0][0] = " + arr[0][0]);
21     }
22 }
```

## Non-Primitive Data Types or Reference Data Types XXIV

### Three – dimensional Array (3D-Array):

Indirect Method of Declaration:

Declaration – Syntax:

```
data_type[][][] array_name = new data_type[x][y][z];
```

For example: int[][][] arr = new int[10][20][30];

Initialization – Syntax:

```
array_name[array_index][row_index][column_index] = value;
```

For example: arr[0][0][0] = 1;

## Non-Primitive Data Types or Reference Data Types XXV

```
1 class GFG
2 {
3     public static void main(String[] args)
4     {
5
6         int[][][] arr = new int[10][20][30];
7         arr[0][0][0] = 1;
8
9         System.out.println("arr[0][0][0] = " + arr[0][0][0]);
10    }
11 }
```

✓ Direct Method of Declaration:

Syntax:

```
data_type[][][] array_name = {
{
{valueA1R1C1, valueA1R1C2, ....},
valueA1R2C1, valueA1R2C2, ....}}
```

## Non-Primitive Data Types or Reference Data Types XXVI

```
},  
{  
valueA2R1C1, valueA2R1C2, ....},  
{valueA2R2C1, valueA2R2C2, ....}  
}  
};
```

For example: int[][][] arr = { {{1, 2}, {3, 4}}, {{5, 6}, {7, 8}} };

## Non-Primitive Data Types or Reference Data Types XXVII

```
1 class GFG
2 {
3     public static void main(String[] args)
4     {
5
6         int[][][] arr =
7         {
8
9             {
10
11                 {
12                     1, 2
13                 }
14
15                 ,
16
17                 {
18                     3, 4
19                 }
20
21             }
22
23             ,
24
25             {
26
27                 {
28                     5, 6
29                 }
30
31             }
32
33         }
34
35     }
36 }
```

## Non-Primitive Data Types or Reference Data Types XXVIII

```
26
27
28
29
30
31
32
33
34
35
36     for (int i = 0; i < 2; i++)
37     for (int j = 0; j < 2; j++)
38     for (int z = 0; z < 2; z++)
39         System.out.println("arr[" + i + "][" + j + "][" + z + "] = " + arr[i][j][z]);
40
41 }
```

## Non-Primitive Data Types or Reference Data Types XXIX

✓ Accessing Elements of Three-Dimensional Arrays:

```
class GFG {  
    public static void main(String[] args)  
    {  
        int[][][] arr = { { { 1, 2 }, { 3, 4 } }, { { 5, 6 }, { 7, 8 } } };  
        System.out.println("arr[0][0][0] = " + arr[0][0][0]);  
    }  
}
```

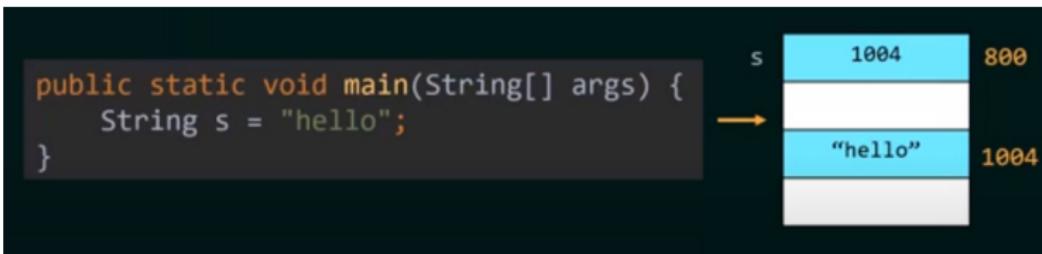
## Non-Primitive Data Types or Reference Data Types XXX

✓ Representation of 3D array in Tabular Format: A three – dimensional array can be seen as a tables of arrays with ‘x’ rows and ‘y’ columns where the row number ranges from 0 to (x-1) and column number ranges from 0 to (y-1). A three – dimensional array with 3 array containing 3 rows and 3 columns is shown below:

Columns					
	Column 1	Column 2	Column 3		
Row 1	111	112	113		
Row 2	121	211	212	213	
Row 3	131	221	311	312	313
	231	321	322	323	
		331	332	333	

The diagram illustrates a 3D array structure. The vertical axis is labeled "Rows" and the horizontal axis is labeled "Columns". The array is divided into three main sections: Row 1 (green), Row 2 (yellow), and Row 3 (pink). Each section contains three columns. Arrows point from each section to a label: "Array 1" for Row 1, "Array 2" for Row 2, and "Array 3" for Row 3. A large green bracket on the right side of the diagram is labeled "Arrays", indicating that the entire structure represents a 3D array.

## String I



- ✓ The String class represents character strings.
- ✓ All string literals in Java programs, such as "abc", are implemented as instances of this class.
- ✓ Strings in Java are Objects that are backed internally by a char array.
- ✓ Since arrays are immutable(cannot grow), Strings are immutable as well. Whenever a change to a String is made, an entirely new String is created.

## String II

- ✓ Strings are constant; their values cannot be changed after they are created.
- ✓ String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
1 String str = "abc";
2 is equivalent to:
3 char data[] =
4 {
5     'a', 'b', 'c'
6 }
7 ;
8
9 String str = new String(data);
```

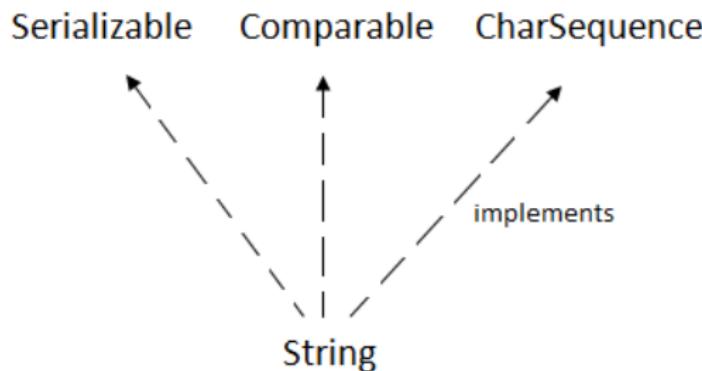
Here are some more examples of how strings can be used:

## String III

```
1 System.out.println("abc");
2 String cde = "cde";
3 System.out.println("abc" + cde);
4 String c = "abc".substring(2,3);
5 String d = cde.substring(1, 2);
6
7 char[] ch=
8 {
9     'j','a','v','a','t','p','o','i','n','t'
10 }
11 ;
12 String s=new String(ch);
13
14 or
15 String s="javatpoint";
```

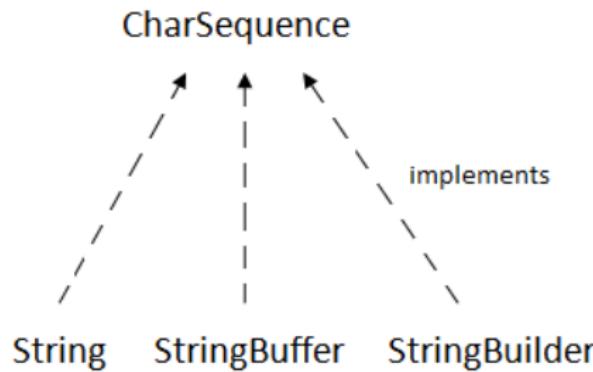
## String IV

- ✓ Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.
- ✓ The `java.lang.String` class implements `Serializable`, `Comparable` and `CharSequence` interfaces.



### CharSequence Interface:

- ✓ The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in java by using these three classes.



## String VI

✓ The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

### How to create a string object?

There are two ways to create String object:

① By string literal

② By new keyword

① String Literal : Java String literal is created by using double quotes. For Example:

```
1 String s="welcome";
```

## String VII

- ✓ Each time you create a string literal, the JVM checks the "string constant pool" first. This allows JVM to optimize the initialization of String literal.
- ✓ If the string already exists in the pool, a reference to the pooled instance is returned.
- ✓ If the string doesn't exist in the pool, a new string instance is created and placed in the pool.

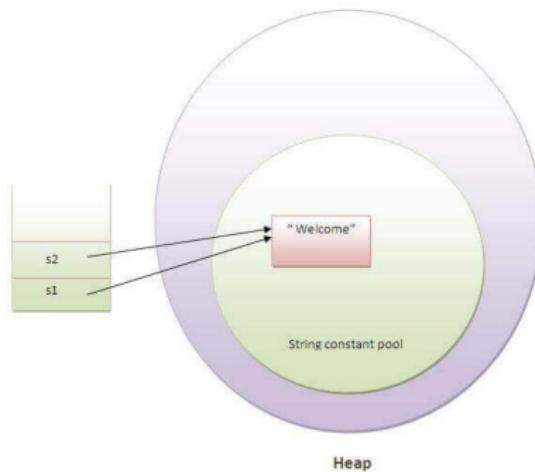
For example:

```
1 String s1="Welcome";  
2 String s2="Welcome"; //It doesn't create a new instance
```

## String VIII

```
1 public class JavaSample
2 {
3     public static void main(String[] args)
4     {
5         String s1="Welcome";
6         String s2="Welcome"; //It doesn't create a new instance
7         System.out.println(Integer.toHexString(s1.hashCode()));
8         System.out.println(Integer.toHexString(s2.hashCode()));
9     }
10 }
11 }
```

# String IX



## String X

- ✓ In the above example, only one object will be created.
- ✓ Firstly, JVM will not find any string object with the value "Welcome" in string constant pool, that is why it will create a new object.
- ✓ After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

Note: String objects are stored in a special memory area known as the "string constant pool".

- ✓ To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

### ② By new keyword:

```
1 String s=new String("Welcome"); //creates two objects and one reference variable
```

- ✓ In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable 's' will refer to the object in a heap (non-pool).

## String XII

### Java String Example:

```
1 public class StringExample
2 {
3     public static void main(String args[])
4     {
5         String s1="java"; //creating string by java string literal
6         char ch[]=
7         {
8             's','t','r','i','n','g','s'
9         }
10        ;
11        String s2=new String(ch); //converting char array to string
12        String s3=new String("example"); //creating java string by new keyword
13        System.out.println(s1);
14        System.out.println(s2);
15        System.out.println(s3);
16    }
17
18 }
```

## String XIII

- ✓ The class String includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase. Case mapping is based on the Unicode Standard version specified by the Character class.
- ✓ The Java language provides special support for the string concatenation operator ( + ), and for conversion of other objects to strings.
- ✓ String concatenation is implemented through the StringBuilder(or StringBuffer) class and its append method.
- ✓ String conversions are implemented through the method `toString`, defined by Object and inherited by all classes in Java. For additional

## String XIV

information on string concatenation and conversion, see Gosling, Joy, and Steele, The Java Language Specification.

- ✓ Unless otherwise noted, passing a null argument to a constructor or method in this class will cause a NullPointerException to be thrown.
- ✓ A String represents a string in the UTF-16 format in which supplementary characters are represented by surrogate pairs.
- ✓ Index values refer to char code units, so a supplementary character uses two positions in a String.
- ✓ The String class provides methods for dealing with Unicode code points (i.e., characters), in addition to those for dealing with Unicode code units (i.e., char values).

### Java String compare:

- ✓ We can compare string in java on the basis of content and reference.
- ✓ It is used in authentication (by equals() method), sorting (by compareTo() method), reference matching (by == operator) etc.
- ✓ There are three ways to compare string in java:

① By equals() method

② By == operator

③ By compareTo() method

① String compare by equals() method

The String equals() method compares the original content of the string. It compares values of string for equality. String class provides two methods:

## String XVI

- public boolean equals(Object another) compares this string to the specified object.
- public boolean equalsIgnoreCase(String another) compares this String to another string, ignoring case.

```
1 public class Teststringcomparison1
2 {
3     public static void main(String args[])
4     {
5         String s1="Sachin";
6         String s2="Sachin";
7         String s3=new String("Sachin");
8         String s4="Saurav";
9         System.out.println(s1.equals(s2)); //true
10        System.out.println(s1.equals(s3)); //true
11        System.out.println(s1.equals(s4)); //false
12    }
13 }
```

## String XVII

```
1 class Teststringcomparison2
2 {
3     public static void main(String args[])
4     {
5         String s1="Sachin";
6         String s2="SACHIN";
7
8         System.out.println(s1.equals(s2)); //false
9         System.out.println(s1.equalsIgnoreCase(s2)); //true
10    }
11 }
```

### ② String compare by == operator

The == operator compares references not values.

## String XVIII

```
1 class Teststringcomparison3
2 {
3     public static void main(String args[])
4     {
5         String s1="Sachin";
6         String s2="Sachin";
7         String s3=new String("Sachin");
8         System.out.println(s1==s2); //true (because both refer to same instance)
9         System.out.println(s1==s3); //false(because s3 refers to instance created in nonpool)
10    }
11 }
```

### ③ String compare by compareTo() method:

The String compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two string variables. If:

- $s1 == s2 : 0$

## String XIX

- $s1 > s2$  :positive value
- $s1 < s2$  :negative value

```
1 class Teststringcomparison4
2 {
3     public static void main(String args[])
4     {
5         String s1="Sachin";
6         String s2="Sachin";
7         String s3="Ratan";
8         System.out.println(s1.compareTo(s2)); //0
9         System.out.println(s1.compareTo(s3)); //1(because s1>s3)
10        System.out.println(s3.compareTo(s1)); //-1(because s3 < s1 )
11    }
12 }
```

### StringBuilder:

- ✓ The StringBuilder in Java represents a mutable sequence of characters.
- ✓ Since the String Class in Java creates an immutable sequence of characters, the StringBuilder class provides an alternate to String Class, as it creates a mutable sequence of characters.

#### Syntax:

```
1  StringBuilder str = new StringBuilder();  
2  str.append("GFG");
```

# String XXI

```
1 public class JavaSample
2 {
3     public static void main(String[] args)
4     {
5         StringBuilder str = new StringBuilder();
6         System.out.println(Integer.toHexString(str.hashCode()));
7         str.append("GFG");
8         System.out.println(str);
9         System.out.println(Integer.toHexString(str.hashCode()));
10        System.out.println(str);
11        str.append(" MCA");
12        System.out.println(Integer.toHexString(str.hashCode()));
13        System.out.println(str);
14    }
15 }
```

# Immutable String in Java

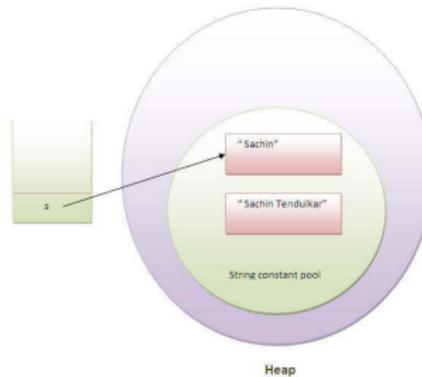
In java, string objects are immutable. Immutable simply means unmodifiable or unchangeable.

Once string object is created its data or state can't be changed but a new string object is created.

Let's try to understand the immutability concept by the example given below:

```
1 class Testimmutablestring
2 {
3     public static void main(String args[])
4     {
5         String s="Sachin";
6         s.concat(" Tendulkar"); //concat() method appends the string at the end
7         System.out.println(s); //will print Sachin because strings are immutable objects
8     }
9 }
```

## String XXIII



- ✓ But if we explicitly assign it to the reference variable, it will refer to "Sachin Tendulkar" object. For example:

## String XXIV

```
1 class Testimmutablestring1
2 {
3     public static void main(String args[])
4     {
5         String s="Sachin";
6         s=s.concat(" Tendulkar");
7         System.out.println(s);
8     }
9 }
```

- ✓ Because java uses the concept of string literal. Suppose there are 5 reference variables, all refers to one object "sachin". If one reference variable changes the value of the object, it will be affected to all the reference variables. That is why string objects are immutable in java.

## String XXV

# String Concatenation in Java:

✓ In java, string concatenation forms a new string that is the combination of multiple strings. There are two ways to concat string in java:

① By + (string concatenation) operator

② By concat() method

① By + (string concatenation) operator

```
1 class TestStringConcatenation1
2 {
3     public static void main(String args[])
4     {
5         String s="Sachin"+" Tendulkar";
6         System.out.println(s); //Sachin Tendulkar
7     }
8 }
```

## String XXVI

```
String s=(new StringBuilder()).append("Sachin").append("Tendulkar).toString();
```

```
1 class TestStringConcatenation2
2 {
3     public static void main(String args[])
4     {
5         String s=50+30+"Sachin"+40+40;
6         System.out.println(s); //80Sachin4040
7     }
8 }
```

### ② String Concatenation by concat() method:

## String XXVII

```
1 //public String concat(String another)
2 class TestStringConcatenation3
3 {
4     public static void main(String args[])
5     {
6         String s1="Sachin ";
7         String s2="Tendulkar";
8         String s3=s1.concat(s2);
9         System.out.println(s3); //Sachin Tendulkar
10    }
11 }
```

## String XXVIII

### Java StringBuffer class:

✓ Java StringBuffer class is used to create mutable (modifiable) string.

The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

### Important Constructors of StringBuffer class.

Constructor	Description
StringBuffer()	creates an empty string buffer with the initial capacity of 16.
StringBuffer(String str)	creates a string buffer with the specified string.
StringBuffer(int capacity)	creates an empty string buffer with the specified capacity as length.

## String XXIX

### Important methods of StringBuffer class:

#### StringBuffer append() method

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

```
1 class StringBufferExample
2 {
3     public static void main(String args[])
4     {
5         StringBuffer sb=new StringBuffer("Hello ");
6         sb.append("Java"); //now original string is changed
7         System.out.println(sb); //prints Hello Java
8     }
9 }
```

#### StringBuffer insert() method:

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

# String XXX

```
1 class StringBufferExample2
2 {
3     public static void main(String args[])
4     {
5         StringBuffer sb=new StringBuffer("Hello ");
6         sb.insert(1,"Java"); //now original string is changed
7         System.out.println(sb); //prints HJavaello
8     }
9 }
```

## StringBuffer replace() method

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

```
1 class StringBufferExample3
2 {
3     public static void main(String args[])
4     {
5         StringBuffer sb=new StringBuffer("Hello");
6         sb.replace(1,3,"Java");
7         System.out.println(sb); //prints HJavaelo
8     }
9 }
```

### StringBuffer delete() method

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

```
1 class StringBufferExample4
2 {
3     public static void main(String args[])
4     {
5         StringBuffer sb=new StringBuffer("Hello");
6         sb.delete(1,3);
7         System.out.println(sb); //prints Hlo
8     }
9 }
```

### StringBuffer reverse() method

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

## String XXXII

```
1 class StringBufferExample5
2 {
3     public static void main(String args[])
4     {
5         StringBuffer sb=new StringBuffer("Hello");
6         sb.reverse();
7         System.out.println(sb); //prints olleH
8     }
9 }
```

### StringBuffer capacity() method

- ✓ The capacity() method of StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by  $(\text{oldcapacity} * 2) + 2$ . For example if your current capacity is 16, it will be  $(16 * 2) + 2 = 34$ .

## String XXXIII

```
1 class StringBufferExample6
2 {
3     public static void main(String args[])
4     {
5         StringBuffer sb=new StringBuffer();
6         System.out.println(sb.capacity()); //default 16
7         sb.append("Hello");
8         System.out.println(sb.capacity()); //now 16
9         sb.append("java is my favourite language");
10        System.out.println(sb.capacity()); //now (16*2)+2=34 i.e (oldcapacity*2)+2
11    }
12 }
```

### StringBuffer ensureCapacity() method

- ✓ The ensureCapacity() method of StringBuffer class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity\*2)+2. For example if your current capacity is 16, it will be  $(16*2)+2=34$ .

## String XXXIV

```
1 class StringBufferExample7
2 {
3     public static void main(String args[])
4     {
5         StringBuffer sb=new StringBuffer();
6         System.out.println(sb.capacity()); //default 16
7         sb.append("Hello");
8         System.out.println(sb.capacity()); //now 16
9         sb.append("java is my favourite language");
10        System.out.println(sb.capacity()); //now (16*2)+2=34 i.e (oldcapacity*2)+2
11        sb.ensureCapacity(10); //now no change
12        System.out.println(sb.capacity()); //now 34
13        sb.ensureCapacity(50); //now (34*2)+2
14        System.out.println(sb.capacity()); //now 70
15    }
16 }
```

## String XXXV

# Difference between String and StringBuffer:

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

## String XXXVI

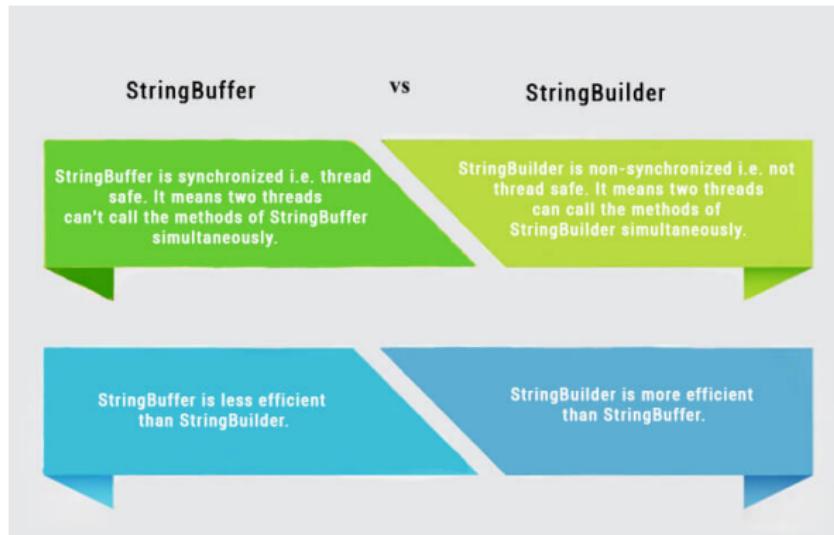
```
1 //Performance Test of String and StringBuffer
2 public class ConcatTest
3 {
4     public static String concatWithString()
5     {
6         String t = "Java";
7         for (int i=0; i<10000; i++)
8         {
9             t = t + "Tpoint";
10        }
11        return t;
12    }
13    public static String concatWithStringBuffer()
14    {
15        StringBuffer sb = new StringBuffer("Java");
16        for (int i=0; i<10000; i++)
17        {
18            sb.append("Tpoint");
19        }
20        return sb.toString();
21    }
22    public static void main(String[] args)
23    {
24        long startTime = System.currentTimeMillis();
25        concatWithString();
```

## String XXXVII

```
26     System.out.println("Time taken by Concatenating with String:  
27         "+(System.currentTimeMillis()-startTime)+"ms");  
28     startTime = System.currentTimeMillis();  
29     concatWithStringBuffer();  
30     System.out.println("Time taken by Concatenating with StringBuffer:  
31         "+(System.currentTimeMillis()-startTime)+"ms");  
32 }
```

# String XXXVIII

## Difference between StringBuffer and StringBuilder:



## String XXXIX

### Performance Test of StringBuffer and StringBuilder:

```
1 //Java Program to demonstrate the performance of StringBuffer and StringBuilder classes.
2 public class ConcatTest
3 {
4     public static void main(String[] args)
5     {
6         long startTime = System.currentTimeMillis();
7         StringBuffer sb = new StringBuffer("Java");
8         for (int i=0; i<10000; i++)
9         {
10             sb.append("Tpoint");
11         }
12         System.out.println("Time taken by StringBuffer: " + (System.currentTimeMillis() - startTime) +
13 "ms");
14         startTime = System.currentTimeMillis();
15         StringBuilder sb2 = new StringBuilder("Java");
16         for (int i=0; i<10000; i++)
17         {
18             sb2.append("Tpoint");
19         }
20         System.out.println("Time taken by StringBuilder: " + (System.currentTimeMillis() - startTime) +
21 "ms");
22     }
23 }
```

# How to read primitive data types in Java I

## Java Scanner Class

- ✓ Java Scanner class allows the user to take input from the console.
- ✓ It belongs to java.util package. It is used to read the input of primitive types like int, double, long, short, float, and byte. It is the easiest way to read input in Java program.

### Syntax

```
1 Scanner sc=new Scanner(System.in);
```

- ✓ The above statement creates a constructor of the Scanner class having System.in as an argument. It means it is going to read from the standard input stream of the program. The java.util package should be import while using Scanner class.
- ✓ It also converts the Bytes (from the input stream) into characters using the platform's default charset.

# How to read primitive data types in Java II

## Java Scanner Class

### Methods of Java Scanner Class:

Method	Description
int nextInt()	It is used to scan the next token of the input as an integer.
float nextFloat()	It is used to scan the next token of the input as a float.
double nextDouble()	It is used to scan the next token of the input as a double.
byte nextByte()	It is used to scan the next token of the input as a byte.
String nextLine()	Advances this scanner past the current line.
boolean nextBoolean()	It is used to scan the next token of the input into a boolean value.
long nextLong()	It is used to scan the next token of the input as a long.
short nextShort()	It is used to scan the next token of the input as a Short.
BigInteger nextBigInteger()	It is used to scan the next token of the input as a BigInteger.
BigDecimal nextBigDecimal()	It is used to scan the next token of the input as a BigDecimal.

# How to read primitive data types in Java III

## Java Scanner Class

### Example of integer Input from user

```
1 import java.util.*;
2 class UserInputDemo
3 {
4     public static void main(String[] args)
5     {
6         Scanner sc= new Scanner(System.in); //System.in is a standard input stream
7         System.out.print("Enter first number- ");
8         int a= sc.nextInt();
9         System.out.print("Enter second number- ");
10        int b= sc.nextInt();
11        System.out.print("Enter third number- ");
12        int c= sc.nextInt();
13        int d=a+b+c;
14        System.out.println("Total= " +d);
15    }
16 }
```

✓ Example of String Input from user

## How to read primitive data types in Java IV

### Java Scanner Class

```
1 import java.util.*;
2 class UserInputDemo1
3 {
4     public static void main(String[] args)
5     {
6         Scanner sc= new Scanner(System.in); //System.in is a standard input stream
7         System.out.print("Enter a string: ");
8         String str= sc.nextLine();          //reads string
9         System.out.print("You have entered: "+str);
10    }
11 }
```

- ✓ To read an element of an array uses these methods in a for loop:

# How to read primitive data types in Java V

## Java Scanner Class

```
1 import java.util.Arrays;
2 import java.util.Scanner;
3
4 public class ReadingWithScanner
5 {
6     public static void main(String args[])
7     {
8         Scanner s = new Scanner(System.in);
9         System.out.println("Enter the length of the array:");
10        int length = s.nextInt();
11        int [] myArray = new int[length];
12        System.out.println("Enter the elements of the array:");
13
14        for(int i=0; i<length; i++ )
15        {
16            myArray[i] = s.nextInt();
17        }
18
19        System.out.println(Arrays.toString(myArray));
20    }
21}
```

# CS101: Problem Solving through C Programming

**Sachchida Nand Chaurasia**  
Assistant Professor

Department of Computer Science  
Banaras Hindu University  
Varanasi

Email id: [snchaurasia@bhu.ac.in](mailto:snchaurasia@bhu.ac.in), [sachchidanand.mca07@gmail.com](mailto:sachchidanand.mca07@gmail.com)



February 10, 2021

## Arrays I

- ✓ The variable allows us to store a single value at a time, what if we want to store roll no. of 100 students?
- ✓ For this task, we have to declare 100 variables, then assign values to each of them.
- ✓ What if there are 10000 students or more?
- ✓ As you can see declaring that many variables for a single entity (i.e student) is not a good idea.
- ✓ In a situation like these arrays provide a better way to store data.

### What is an Array?

- ✓ An array is a collection of one or more values of the same type.
- ✓ Each value is called an element of the array.
- ✓ The elements of the array share the same variable name but each element has its own unique index number (also known as a subscript).
- ✓ An array can be of any type, For example: int, float, char etc. If an array is of type int then it's elements must be of type int only.

## Arrays III

**roll\_no[0]**



**array variable**



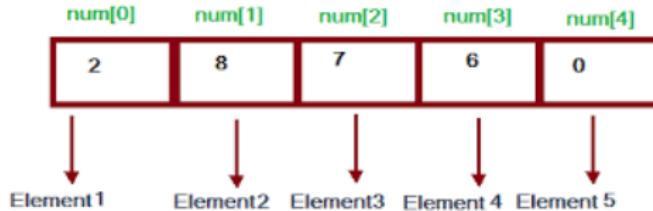
**subscript or index**

## Arrays IV

To store roll no. of 100 students, we have to declare an array of size 100 i.e `roll_no[100]`. Here size of the array is 100 , so it is capable of storing 100 values. In C, index or subscript starts from 0, so `roll_no[0]` is the first element, `roll_no[1]` is the second element and so on. Note that the last element of the array will be at `roll_no[99]` not at `roll_no[100]` because the index starts at 0.

# Arrays V

## One-dimensional array:



```
1 Syntax: datatype array_name[size];
2 datatype: It denotes the type of the elements in the array.
3 array_name: Name of the array. It must be a valid identifier.
4 size: Number of elements an array can hold. here are some example of array declarations:
5
6 int num[100];
7 float temp[20];
8 char ch[50];
9
10 num[0], num[1], num[2], ...., num[99]
11 temp[0], temp[1], temp[2], ...., temp[19]
12 ch[0], ch[1], ch[2], ...., ch[49]
```

## Arrays VI

```
1 #define SIZE 10
2 int main()
3 {
4     int size = 10;
5
6     int my_arr1[SIZE]; // ok
7     int my_arr1[10]; // ok
8     int my_arr2[size]; // not allowed
9     // ...
10 }
```

### Accessing elements of an array:

- ✓ The elements of an array can be accessed by specifying array name followed by subscript or index inside square brackets (i.e []).
- ✓ Array subscript or index starts at 0. If the size of an array is 10 then the first element is at index 0, while the last element is at index 9.
- ✓ The first valid subscript (i.e 0) is known as the lower bound, while last valid subscript is known as the upper bound.

# Arrays VII

```
1 int my_arr[5];
2
3 //then elements of this array are;
4
5 First element -> my_arr[0]
6 Second element -> my_arr[1]
7 Third element -> my_arr[2]
8 Fourth element -> my_arr[3]
9 Fifth element -> my_arr[4]
10
11 //Array subscript or index can be any expression that yields an integer value. For example:
12
13 int i = 0, j = 2;
14 my_arr[i]; // 1st element
15 my_arr[i+1]; // 2nd element
16 my_arr[i+j]; // 3rd element
17
18 //In the array my_arr, the last element is at my_arr[4], What if you try to access elements beyond the last
19 // valid index of the array?
20
21 printf("%d", my_arr[5]); // 6th element
22 printf("%d", my_arr[10]); // 11th element
23 printf("%d", my_arr[-1]); // element just before 0
```

# Arrays VIII

```
24 //Sure indexes 5, 10 and -1 are not valid but C compiler will not show any error message instead some garbage  
25 value will be printed.  
//The C language doesn't check bounds of the array. It is the responsibility of the programmer to check array  
bounds whenever required.
```

## Processing 1-D arrays:

```
1 #include<stdio.h>  
2 int main()  
3 {  
4     int arr[5], i;  
5     for(i = 0; i < 5; i++)  
6     {  
7         printf("Enter a[%d]: ", i);  
8         scanf("%d", &arr[i]);  
9     }  
10    printf("\nPrinting elements of the array: \n\n");  
11    for(i = 0; i < 5; i++)  
12    {  
13        printf("%d ", arr[i]);  
14    }  
15    return 0;  
16 }
```

# Arrays IX

```
1 //The following program prints the sum of elements of an array.
2 #include<stdio.h>
3 int main()
4 {
5     int arr[5], i, s = 0;
6     for(i = 0; i < 5; i++)
7     {
8         printf("Enter a[%d]: ", i);
9         scanf("%d", &arr[i]);
10    }
11    for(i = 0; i < 5; i++)
12    {
13        s += arr[i];
14    }
15    printf("\nSum of elements = %d ", s);
16    return 0;
17 }
```

### Initializing Array:

- ✓ When an array is declared inside a function the elements of the array have garbage value.
- ✓ If an array is global or static, then its elements are automatically initialized to 0.
- ✓ We can explicitly initialize elements of an array at the time of declaration using the following syntax:

Syntax:

- ⇒ datatype array\_name[size] = {val\_1, val\_2, val\_3, ..... val\_N};
- ⇒ datatype is the type of elements of an array.
- ⇒ array\_name is the variable name, which must be any valid identifier.
- ⇒ size is the size of the array.
- ✓ val\_1, val\_2, val\_3, ..... val\_N are the constants known as initializers.

## Arrays XI

Tick Each value is separated by a comma(,) and then there is a semi-colon (;) after the closing curly brace ()).

```
1 float temp[5] =
2 {
3     12.3, 4.1, 3.8, 9.5, 4.5
4 }
5 ; // an array of 5 floats
6 int arr[9] =
7 {
8     11, 22, 33, 44, 55, 66, 77, 88, 99
9 }
10 ; // an array of 9 ints
```

## Arrays XII

- ✓ While initializing 1-D array it is optional to specify the size of the array, so you can also write the above statements as:

```
1 float temp[] =  
2 {  
3     12.3, 4.1, 3.8, 9.5, 4.5  
4 }  
5 ; // an array of 5 floats  
6 int arr[] =  
7 {  
8     11, 22, 33, 44, 55, 66, 77, 88, 99  
9 }  
10; // an array of 9 ints
```

## Arrays XIII

```
1 #include<stdio.h>
2 #define SIZE 10
3
4 int main()
5 {
6     int my_arr[SIZE] =
7     {
8         34, 56, 78, 15, 43, 71, 89, 34, 70, 91
9     }
10    ;
11    int i, max, min;
12
13    max = min = my_arr[0];
14
15    for(i = 0; i < SIZE; i++)
16    {
17        // if value of current element is greater than previous value
18        // then assign new value to max
19        if(my_arr[i] > max)
20        {
21            max = my_arr[i];
22        }
23
24        // if the value of current element is less than previous element
25        // then assign new value to min
```

## Arrays XIV

```
26     if(my_arr[i] < min)
27     {
28         min = my_arr[i];
29     }
30
31 }
32
33 printf("Lowest value = %d\n", min);
34 printf("Highest value = %d", max);
35
36 // signal to operating system everything works fine
37 return 0;
38 }
```

# Arrays XV

```
1 // Program to find the average of n numbers using arrays
2 #include <stdio.h>
3 int main()
4 {
5     int marks[10], i, n, sum = 0, average;
6     printf("Enter number of elements: ");
7     scanf("%d", &n);
8
9     for(i=0; i<n; ++i)
10
11    {
12        printf("Enter number%d: ",i+1);
13        scanf("%d", &marks[i]);
14
15        // adding integers entered by the user to the sum variable
16        sum += marks[i];
17
18    }
19    average = sum/n;
20    printf("Average = %d", average);
21
22    return 0;
23 }
```

### Practice questions on 1-D Arrays

Type 1. Based on array declaration – These are few key points on array declaration:

A single dimensional array can be declared as int a[10] or int a[] = {1, 2, 3, 4}. It means specifying the number of elements is optional in 1-D array. A two dimensional array can be declared as int a[2][4] or int a[][] = {1, 2, 3, 4, 5, 6, 7, 8}. It means specifying the number of rows is optional but columns are mandatory. The declaration of int a[4] will give the values as garbage if printed. However, int a[4] = 1,1 will initialize remaining two elements as 0.

## Arrays XVII

```
1 int main()
2 {
3     int i;
4     int arr[5] =
5     {
6         1
7     }
8 ;
9     for (i = 0; i < 5; i++)
10    printf("%d ", arr[i]);
11    return 0;
12 }
13 Output: ???
```

## Arrays XVIII

Type 2. Finding address of an element with given base address - When an array is declared, a contiguous block of memory is assigned to it which helps in finding address of elements from base address.

For a single dimensional array  $a[100]$ , address of  $i$ th element can be found as:

$$\text{addr}(a[i]) = BA + i * \text{SIZE}$$

## Arrays XIX

Type 3. Accessing array elements using pointers -

In a single dimensional array  $a[100]$ , the element  $a[i]$  can be accessed as  $a[i]$  or  $*(a+i)$  or  $*(i+a)$  Address of  $a[i]$  can be accessed as  $a[i]$  or  $(a+i)$  or  $(i+a)$  In two dimensional array  $a[100][100]$ , the element  $a[i][j]$  can be accessed as  $a[i][j]$  or  $*(*(a+i)+j)$  or  $*(a[i]+j)$  Address of  $a[i][j]$  can be accessed as  $a[i][j]$  or  $a[i]+j$  or  $*(a+i)+j$  In two dimensional array, address of ith row can be accessed as  $a[i]$  or  $*(a+i)$

# Arrays XX

```
1 Assume the following C variable declaration
2
3 int *A [10], B[10][10];
4 Of the following expressions
5
6 I. A[2]
7 II. A[2][3]
8 III. B[1]
9 IV. B[2][3]
10 which will not give compile-time errors if used as left hand sides of assignment statements in a C program (
    GATE CS 2003)?
11 (A) I, II, and IV only
12 (B) II, III, and IV only
13 (C) II and IV only
14 (D) IV only
15
16 Solution: As given in the question, A is an array of 10 pointers and B is a two dimensional array. Considering
    this, we take an example as:
17
18 int *A[10], B[10][10];
19 int C[] =
20 {
21     1, 2, 3, 4, 5
22 }
23 ;
```

## Arrays XXI

- 24 As A[2] represents an integer pointer, it can store the address of integer array as: A[2] = C; therefore, I is valid.
- 25
- 26 As A[2] represents base address of C, A[2][3] can be modified as: A[2][3] = \*(C +3) = 0; it will change the value of C[3] to 0. Hence, II is also valid.
- 27
- 28 As B is 2D array, B[2][3] can be modified as: B[2][3] = 5; it will change the value of B[2][3] to 5. Hence, IV is also valid.
- 29
- 30 As B is 2D array, B[2] represent address of 2nd row which cant be used at LHS of statement as it is invalid to modify the address. Hence III is invalid.

## Arrays XXII

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

abc[0][1]	abc[0][2]	abc[0][3]	abc[1][0]	abc[1][1]	....	....	abc[4][2]	abc[4][3]
-----------	-----------	-----------	-----------	-----------	------	------	-----------	-----------

82206	82210	82214	82218	82222	.....	.....	82274	82278
-------	-------	-------	-------	-------	-------	-------	-------	-------

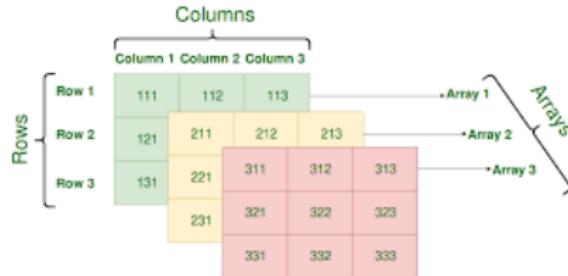
memory locations for the array elements

Array is of integer type so each element would use 4 bytes that's the reason there is a difference of 4 in element's addresses.

The addresses are generally represented in hex. This diagram shows them in integer just to show you that the elements are stored in contiguous locations, so that you can understand that the address difference between each element is equal to the size of one element(int size 4). For better understanding see the program below.

### Actual memory representation of a 2D array

# Arrays XXIII



```
int num[3][4] = {  
    {1, 2, 3, 4},  
    {5, 6, 7, 8},  
    {9, 10, 11, 12}  
};
```

row ↓      col →

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

CLASSROOM

dyclassroom.com

## Practice questions on 2-D Arrays

## Arrays XXIV

```
1 int main()
2 {
3     int a[][] =
4     {
5         {
6             1,2
7         }
8         ,
9         {
10            3,4
11        }
12    }
13
14 }
15 ;
16     int i, j;
17     for (i = 0; i < 2; i++)
18         for (j = 0; j < 2; j++)
19             printf("%d ", a[i][j]);
20     return 0;
21 }
```

- ✓ Consider the following declaration of a ‘two-dimensional array in C:

## Arrays XXV

```
1 char a[100][100];
2 Assuming that the main memory is byte-addressable and that the array is stored starting from memory address
   0, the address of a[40][50] is: (GATE CS 2002)
3 (A) 4040
4 (B) 4050
5 (C) 5040
6 (C) 5050
7
8 Solution:
9 addr[40][50] = 0 + (40*100 + 50) * 1 = 4050
```

- ✓ For a C program accessing  $X[i][j][k]$ , the following intermediate code is generated by a compiler. Assume that the size of an integer is 32 bits and the size of a character is 8 bits. (GATE-CS-2014)

## Arrays XXVI

Que - 4. For a C program accessing  $X[i][j][k]$ , the following intermediate code is generated by a compiler.  
Assume that the size of an integer is 32 bits and the size of a character is 8 bits. (GATE-CS-2014)

```
1 t0 = i * 1024
2 t1= j * 32
3 t2 = k * 4
4 t3 =t1 + t0
5 t4 = t3 + t2
6 t5 = X[t4]
```

Which one of the following statement about the source code of C program is correct?

- (A) X is declared as int X[32][32][8]
- (B) X is declared as int X[4][1024][32]
- (C) X is declared as char X[4][32][8]
- (D) X is declared as char X[32][16][2]

Solution: For a three dimensional array  $X[10][20][30]$ , we have 10 two dimensional matrices of size  $[20] \times [30]$ .  
Therefore, for a 3 D array  $X[M][N][O]$ , the address of  $X[i][j][k]$  can be calculated as:

BA + (i\*N\*0+j\*0+k)\*SIZE

Given different expressions, the final value of t5 can be calculated as:

t5 = X[t4] = X[t3+t2] = X[t1+t0+t2] = X[i\*1024+j\*32+k\*4]

By equating addresses,

## Arrays XXVII

```
24  
25 (i*N*0+j*0+k)SIZE = i*1024+j*32+k*4 = (i*256+j*8+k)4  
26 Comparing the values of i, j and SIZE, we get  
27  
28 SIZE = 4, N*0 = 256 and 0 = 8, hence, N = 32  
29 As size is 4, array will be integer. The option which matches value of N and 0 and array as integer is (A).
```

## String I

In C programming, a string is a sequence of characters terminated with a null character . For example:

```
1 char c[] = "c string";  
  
1 char c[] = "abcd";  
2 char c[50] = "abcd";  
3 char c[] =  
4 {  
5     'a', 'b', 'c', 'd', '\0'  
6 }  
7 ;  
8 char c[5] =  
9 {  
10    'a', 'b', 'c', 'd', '\0'  
11 }  
12 ;
```

# CS204:Object Oriented Programming

## Concepts

**Sachchida Nand Chaurasia**  
Assistant Professor

Department of Computer Science  
Banaras Hindu University  
Varanasi

Email id: [snchaurasia@bhu.ac.in](mailto:snchaurasia@bhu.ac.in), [sachchidanand.mca07@gmail.com](mailto:sachchidanand.mca07@gmail.com)



February 14, 2021

## Inheritance I

Inheritance is an important pillar of OOP(Object Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class.

### Important terminology:

- Super Class: The class whose features are inherited is known as super class(or a base class or a parent class).
- Sub Class: The class that inherits the other class is known as sub class(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

## Inheritance II

- Reusability: Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

How to use inheritance in Java

The keyword used for inheritance is extends. Syntax :

```
1 class derived-class extends base-class
2 {
3     //methods and fields
4 }
```

### Java Object Creation of Inherited Class:

- ✓ In inheritance, subclass acquires super class properties. An important point to note is, when subclass object is created, a separate object of super class object will not be created. Only a subclass object object is created that has super class variables.
- ✓ This situation is different from a normal assumption that a constructor call means an object of the class is created, so we can't blindly say that whenever a class constructor is executed, object of that class is created or not.

# Inheritance IV

```
1 // A Java program to demonstrate that both super class  
2 // and subclass constructors refer to same object  
3  
4 // super class  
5 class Fruit  
6 {  
7     public Fruit()  
8     {  
9         System.out.println("Super class constructor");  
10        System.out.println("Super class object hashCode :" +  
11            this.hashCode());  
12        System.out.println(this.getClass().getName());  
13    }  
14}  
15  
16 // sub class  
17 class Apple extends Fruit  
18 {  
19     public Apple()  
20     {  
21         System.out.println("Subclass constructor invoked");  
22         System.out.println("Sub class object hashCode :" +  
23             this.hashCode());  
24         System.out.println(this.hashCode() + " " +  
25             super.hashCode());  
26 }
```

## Inheritance V

```
26
27         System.out.println(this.getClass().getName() + " " +
28             super.getClass().getName());
29     }
30 }
31
32 // driver class
33 public class Test
34 {
35     public static void main(String[] args)
36     {
37         Apple myApple = new Apple();
38     }
39 }
```

- ✓ Example: In below example of inheritance, class Bicycle is a base class, class MountainBike is a derived class which extends Bicycle class and class Test is a driver class to run program.

# Inheritance VI

```
1 //Java program to illustrate the concept of inheritance
2 // base class
3 class Bicycle
4 {
5     // the Bicycle class has two fields
6     public int gear;
7     public int speed;
8     // the Bicycle class has one constructor
9     public Bicycle(int gear, int speed)
10    {
11        this.gear = gear;
12        this.speed = speed;
13    }
14    // the Bicycle class has three methods
15    public void applyBrake(int decrement)
16    {
17        speed -= decrement;
18    }
19    public void speedUp(int increment)
20    {
21        speed += increment;
22    }
23
24    // toString() method to print info of Bicycle
25    public String toString()
```

# Inheritance VII

```
26     {
27         return("No of gears are "+gear
28             +"\n"
29             + "speed of bicycle is "+speed);
30     }
31 }
32
33 // derived class
34 class MountainBike extends Bicycle
35 {
36
37     // the MountainBike subclass adds one more field
38     public int seatHeight;
39
40     // the MountainBike subclass has one constructor
41     public MountainBike(int gear,int speed,
42     int startHeight)
43     {
44         // invoking base-class(Bicycle) constructor
45         super(gear, speed);
46         seatHeight = startHeight;
47     }
48
49     // the MountainBike subclass adds one more method
50     public void setHeight(int newValue)
```

# Inheritance VIII

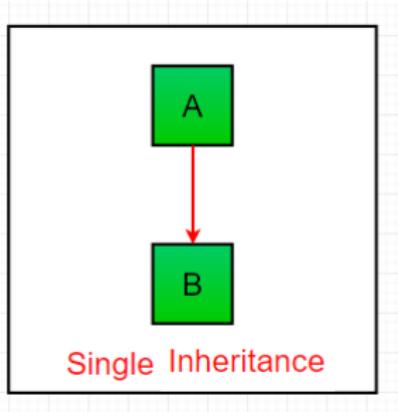
```
51     {
52         seatHeight = newValue;
53     }
54
55     // overriding toString() method
56     // of Bicycle to print more info
57     @Override
58     public String toString()
59     {
60         return (super.toString()+
61             "\nseat height is "+seatHeight);
62     }
63
64 }
65
66 // driver class
67 public class Test
68 {
69     public static void main(String args[])
70     {
71
72         MountainBike mb = new MountainBike(3, 100, 25);
73         System.out.println(mb.toString());
74     }
75     Output:
```

# Inheritance IX

76      No of gears are 3  
77      speed of bicycle is 100  
78      seat height is 25

## Types of Inheritance in Java:

- ① Single Inheritance : In single inheritance, subclasses inherit the features of one superclass. In image below, the class A serves as a base class for the derived class B.



# Inheritance X

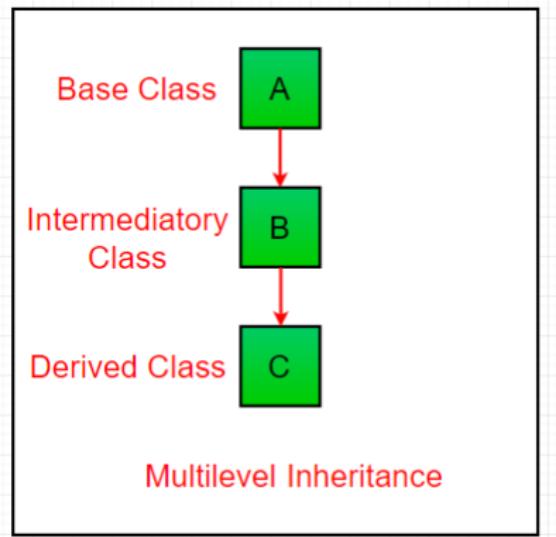
```
1 //Java program to illustrate the
2 // concept of single inheritance
3 import java.util.*;
4 import java.lang.*;
5 import java.io.*;
6
7 class one
8 {
9     public void print_geek()
10    {
11        System.out.println("Geeks");
12    }
13 }
14
15 class two extends one
16 {
17     public void print_for()
18    {
19        System.out.println("for");
20    }
21 }
22 // Driver class
23 public class Main
24 {
25     public static void main(String[] args)
```

## Inheritance XI

```
26     {  
27         two g = new two();  
28         g.print_geek();  
29         g.print_for();  
30         g.print_geek();  
31     }  
32 }
```

- ② Multilevel Inheritance : In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In below image, the class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the *grandparent's members*.

## Inheritance XII



# Inheritance XIII

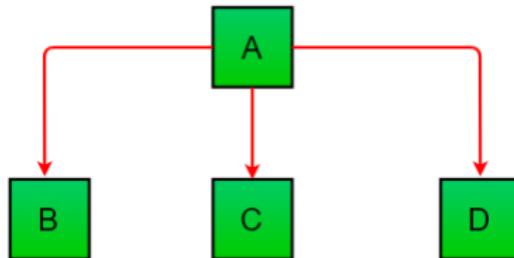
```
1 // Java program to illustrate the
2 // concept of Multilevel inheritance
3 import java.util.*;
4 import java.lang.*;
5 import java.io.*;
6
7 class one
8 {
9     public void print_geek()
10    {
11        System.out.println("Geeks");
12    }
13 }
14
15 class two extends one
16 {
17     public void print_for()
18    {
19        System.out.println("for");
20    }
21 }
22
23 class three extends two
24 {
25     public void print_geek()
```

## Inheritance XIV

```
26     {
27         System.out.println("Geeks");
28     }
29 }
30
31 // Drived class
32 public class Main
33 {
34     public static void main(String[] args)
35     {
36         three g = new three();
37         g.print_geek();
38         g.print_for();
39         g.print_geek();
40     }
41 }
```

- ③ Hierarchical Inheritance : In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one sub class. In below image, the class A serves as a base class for the derived class B,C and D.

# Inheritance XV



Hierarchical Inheritance

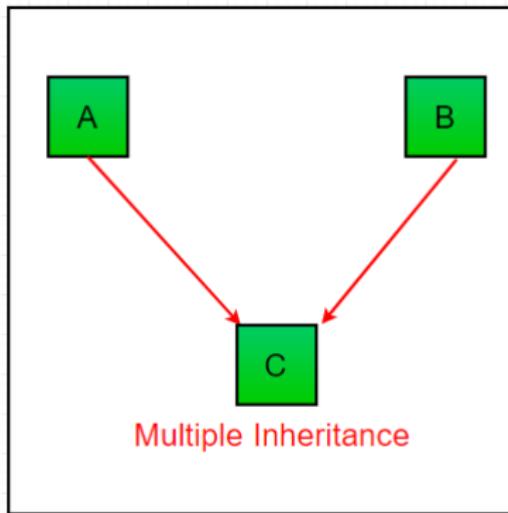
```
1 // Java program to illustrate the
2 // concept of Hierarchical inheritance
3 import java.util.*;
4 import java.lang.*;
5 import java.io.*;
6
7 class one
8 {
9     public void print_geek()
10    {
11        System.out.println("Geeks");
12    }
13 }
```

# Inheritance XVI

```
14  
15 class two extends one  
16 {  
17     public void print_for()  
18     {  
19         System.out.println("for");  
20     }  
21 }  
22  
23 class three extends one  
24 {  
25     /*.....*/  
26 }  
27  
28 // Drived class  
29 public class Main  
30 {  
31     public static void main(String[] args)  
32     {  
33         three g = new three();  
34         g.print_geek();  
35         two t = new two();  
36         t.print_for();  
37         g.print_geek();  
38     }  
}
```

- ④ Multiple Inheritance (Through Interfaces) : In Multiple inheritance ,one class can have more than one superclass and inherit features from all parent classes. Please note that Java does not support multiple inheritance with classes. In java, we can achieve multiple inheritance only through Interfaces. In image below, Class C is derived from interface A and B.

## Inheritance XVIII



# Inheritance XIX

```
1 // Java program to illustrate the
2 // concept of Multiple inheritance
3 import java.util.*;
4 import java.lang.*;
5 import java.io.*;
6
7 interface one
8 {
9     public void print_geek();
10}
11
12 interface two
13 {
14     public void print_for();
15}
16
17 interface three extends one,two
18 {
19     public void print_geek();
20}
21 class child implements three
22 {
23     @Override
24     public void print_geek()
25     {
```

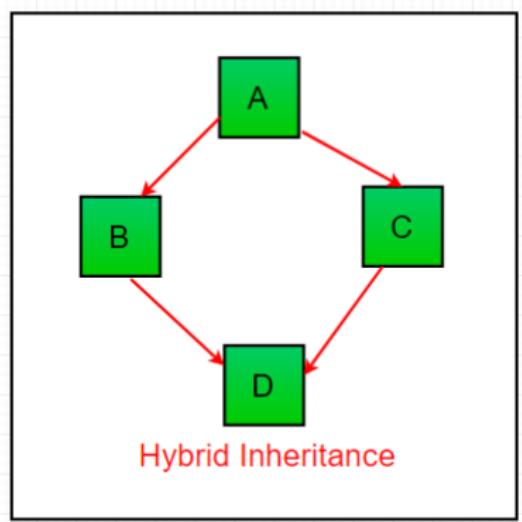
# Inheritance XX

```
26     System.out.println("Geeks");
27 }
28
29     public void print_for()
30 {
31     System.out.println("for");
32 }
33 }
34
35 // Drived class
36 public class Main
37 {
38     public static void main(String[] args)
39     {
40         child c = new child();
41         c.print_geek();
42         c.print_for();
43         c.print_geek();
44     }
45 }
```

## Inheritance XXI

- ⑤ Hybrid Inheritance(Through Interfaces) : It is a mix of two or more of the above types of inheritance. Since java doesn't support multiple inheritance with classes, the hybrid inheritance is also not possible with classes. In java, we can achieve hybrid inheritance only through Interfaces.

## Inheritance XXII



**Important facts about inheritance in Java:**

## Inheritance XXIII

- Default superclass: Except Object class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of Object class.
- Superclass can only be one: A superclass can have any number of subclasses. But a subclass can have only one superclass. This is because Java does not support multiple inheritance with classes. Although with interfaces, multiple inheritance is supported by java.

## Inheritance XXIV

- Inheriting Constructors: A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.
- Private member inheritance: A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods(like getters and setters) for accessing its private fields, these can also be used by the subclass.

## Encapsulations I

- ✓ Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.
- ✓ Encapsulation in Java is a process of wrapping code and data together into a single unit, for example, a capsule which is mixed of several medicines.

### **encapsulation in java**

- ✓ We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

### **Advantage of Encapsulation in Java**

- ✓ By providing only a setter or getter method, you can make the class read-only or write-only. In other words, you can skip the getter or setter methods.

## Encapsulations II

- ✓ It provides you the control over the data. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.
- ✓ It is a way to achieve data hiding in Java because other class will not be able to access the data through the private data members.
- ✓ In Java, encapsulation helps us to keep related fields and methods together, which makes our code cleaner and easy to read.
- ✓ It helps to control the values of our data fields.
- ✓ The encapsulate class is easy to test. So, it is better for unit testing.

## Encapsulations III

```
1 class Area
2 {
3     // fields to calculate area
4     int length;
5     int breadth;
6     // constructor to initialize values
7     Area(int length, int breadth)
8     {
9         this.length = length;
10        this.breadth = breadth;
11    }
12    // method to calculate area
13    public void getArea()
14    {
15        int area = length * breadth;
16        System.out.println("Area: " + area);
17    }
18 }
19 class Main
20 {
21     public static void main(String[] args)
22     {
23         // create object of Area
24         // pass value of length and breadth
25         Area rectangle = new Area(5, 6);
```

## Encapsulations IV

```
26     rectangle.getArea();  
27 }  
28 }
```

- ✓ The getter and setter methods provide read-only or write-only access to our class fields. For example,

```
1 getName() // provides read-only access  
2 setName() // provides write-only access
```

### Data hiding using the private specifier:

```
1 class Person  
2 {  
3  
4     // private field  
5     private int age;  
6  
7     // getter method  
8     public int getAge()  
9     {  
10         return age;  
11     }  
12  
13     // setter method
```

## Encapsulations V

```
14     public void setAge(int age)
15     {
16         this.age = age;
17     }
18 }
19
20 class Main
21 {
22     public static void main(String[] args)
23     {
24
25         // create an object of Person
26         Person p1 = new Person();
27
28         // change age using setter
29         p1.setAge(24);
30
31         // access age using getter
32         System.out.println("My age is " + p1.getAge());
33     }
34 }
```

### Read-Only class

## Encapsulations VI

```
1 //A Java class which has only getter methods.  
2 public class Student  
3 {  
4     //private data member  
5     private String college="AKG";  
6     //getter method for college  
7     public String getCollege()  
8     {  
9         return college;  
10    }  
11 }
```

## Write-Only class

```
1 //A Java class which has only setter methods.  
2 public class Student  
3 {  
4     //private data member  
5     private String college;  
6     //getter method for college  
7     public void setCollege(String college)  
8     {  
9         this.college=college;  
10    }  
11 }
```

## Encapsulations VII

## Polymorphism I

✓ Polymorphism in Java is a concept by which we can perform a single action in different ways.

✓ Polymorphism is derived from 2 Greek words: poly and morphs.

The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

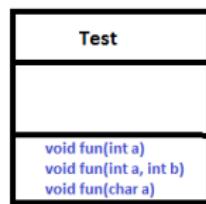
✓ There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

✓ If you overload a static method in Java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

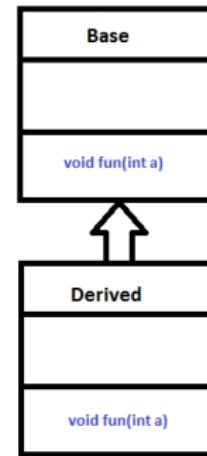
## Polymorphism II

### Compile-time polymorphism:

- ✓ It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading. But Java doesn't support the Operator Overloading.



Overloading



Overriding

## Polymorphism III

- ✓ Method Overloading: When there are multiple functions with same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments.

# Polymorphism IV

## Example 1: By using different types of arguments

```
1 // Java program for Method overloading
2 class MultiplyFun
3 {
4     // Method with 2 parameter
5     static int Multiply(int a, int b)
6     {
7         return a * b;
8     }
9     // Method with the same name but 2 double parameter
10    static double Multiply(double a, double b)
11    {
12        return a * b;
13    }
14 }
15 class Main
16 {
17     public static void main(String[] args)
18     {
19         System.out.println(MultiplyFun.Multiply(2, 4));
20
21         System.out.println(MultiplyFun.Multiply(5.5, 6.3));
22     }
23 }
```

# Polymorphism V

## Example 2: By using different numbers of arguments

```
1 // Java program for Method overloading
2 class MultiplyFun
3 {
4     // Method with 2 parameter
5     static int Multiply(int a, int b)
6     {
7         return a * b;
8     }
9     // Method with the same name but 3 parameter
10    static int Multiply(int a, int b, int c)
11    {
12        return a * b * c;
13    }
14 }
15 class Main
16 {
17     public static void main(String[] args)
18     {
19         System.out.println(MultiplyFun.Multiply(2, 4));
20
21         System.out.println(MultiplyFun.Multiply(2, 7, 3));
22     }
23 }
```

## Polymorphism VI

### Runtime Polymorphism in Java

- ✓ Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an **overridden method** is resolved at runtime rather than compile-time.
- ✓ In this process, an overridden method is called through the reference variable of a superclass.
- ✓ The determination of the method to be called is based on the object being referred to by the reference variable.
- ✓ Understand the upcasting before Runtime Polymorphism.

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:

## Polymorphism VII

```
1 class Bike
2 {
3     void run()
4     {
5         System.out.println("running");
6     }
7 }
8 class Splendor extends Bike
9 {
10    void run()
11    {
12        System.out.println("running safely with 60km");
13    }
14
15    public static void main(String args[])
16    {
17        Bike b = new Splendor(); //upcasting
18        b.run();
19    }
20 }
```

# Polymorphism VIII

## Java Runtime Polymorphism Example:

```
1 // Java program for Method overriding
2 class Parent
3 {
4     void Print()
5     {
6         System.out.println("parent class");
7     }
8 }
9 class subclass1 extends Parent
10 {
11     void Print()
12     {
13         System.out.println("subclass1");
14     }
15 }
16 class subclass2 extends Parent
17 {
18     void Print()
19     {
20         System.out.println("subclass2");
21     }
22 }
23 class TestPolymorphism3
24 {
```

# Polymorphism IX

```
25     public static void main(String[] args)
26     {
27         Parent a;
28         a = new subclass1();
29         a.Print();
30         a = new subclass2();
31         a.Print();
32     }
33 }
```

## ✓ Example:

```
1 class Shape
2 {
3     void draw()
4     {
5         System.out.println("drawing... ");
6     }
7 }
8 class Rectangle extends Shape
9 {
10    void draw()
11    {
12        System.out.println("drawing rectangle... ");
13    }
14 }
```

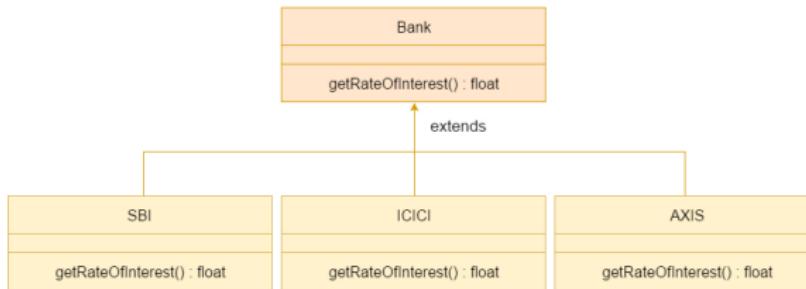
# Polymorphism X

```
15 class Circle extends Shape
16 {
17     void draw()
18     {
19         System.out.println("drawing circle...");
20     }
21 }
22 class Triangle extends Shape
23 {
24     void draw()
25     {
26         System.out.println("drawing triangle...");
27     }
28 }
29 class TestPolymorphism2
30 {
31     public static void main(String args[])
32     {
33         Shape s;
34         s=new Rectangle();
35         s.draw();
36         s=new Circle();
37         s.draw();
38         s=new Triangle();
39         s.draw();
```

## Polymorphism XI

40 }  
41 }

- ✓ Consider a scenario where Bank is a class that provides a method to get the rate of interest. However, the rate of interest may differ according to banks. For example, SBI, ICICI, and AXIS banks are providing 8.4%, 7.3%, and 9.7% rate of interest.



## Polymorphism XII

```
1 class Bank
2 {
3     float getRateOfInterest()
4     {
5         return 0;
6     }
7 }
8 class SBI extends Bank
9 {
10    float getRateOfInterest()
11    {
12        return 8.4f;
13    }
14 }
15 class ICICI extends Bank
16 {
17    float getRateOfInterest()
18    {
19        return 7.3f;
20    }
21 }
22 class AXIS extends Bank
23 {
24    float getRateOfInterest()
25    {
```

# Polymorphism XIII

```
26         return 9.7f;
27     }
28 }
29 class TestPolymorphism
30 {
31     public static void main(String args[])
32     {
33         Bank b;
34         b=new SBI();
35         System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
36         b=new ICICI();
37         System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
38         b=new AXIS();
39         System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
40     }
41 }
```

## Polymorphism XIV

### Java Runtime Polymorphism with Multilevel Inheritance:

```
1 class Animal
2 {
3     void eat()
4     {
5         System.out.println("eating");
6     }
7 }
8 class Dog extends Animal
9 {
10    void eat()
11    {
12        System.out.println("eating fruits");
13    }
14 }
15 class BabyDog extends Dog
16 {
17    void eat()
18    {
19        System.out.println("drinking milk");
20    }
21    public static void main(String args[])
22    {
23        Animal a1,a2,a3;
24        a1=new Animal();
```

# Polymorphism XV

```
25         a2=new Dog();
26         a3=new BabyDog();
27         a1.eat();
28         a2.eat();
29         a3.eat();
30     }
31 }
```

## Try for Output:

```
1 class Animal
2 {
3     void eat()
4     {
5         System.out.println("animal is eating...");
6     }
7 }
8 class Dog extends Animal
9 {
10    void eat()
11    {
12        System.out.println("dog is eating...");
13    }
14 }
15 public class BabyDog1 extends Dog
16 {
```

## Polymorphism XVI

```
17 public static void main(String args[])
18 {
19     Animal a=new BabyDog1();
20     a.eat();
21 }
22
23 }
```

- ✓ Since, BabyDog1 is not overriding the eat() method, so eat() method of Dog class is invoked.

# CS204:Object Oriented Programming

## Concepts

**Sachchida Nand Chaurasia**  
Assistant Professor

Department of Computer Science  
Banaras Hindu University  
Varanasi

Email id: [snchaurasia@bhu.ac.in](mailto:snchaurasia@bhu.ac.in), [sachchidanand.mca07@gmail.com](mailto:sachchidanand.mca07@gmail.com)



January 28, 2021

## Java Constructor I

- ✓ In Java, a constructor is a block of codes similar to the method.
- ✓ It is called when an instance of the class is created.
- ✓ At the time of calling constructor, memory for the object is allocated in the memory.
- ✓ It is a special type of method which is used to initialize the object.
- ✓ Every time an object is created using the new() keyword, at least one constructor is called.
- ✓ It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

## Java Constructor II

Note: It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

## Java Constructor III

**There are two rules defined for the constructor**

- Constructor name must be the same as its class name
- A Constructor must have no explicit return type
- A Java constructor cannot be abstract, static, final, and synchronized.

Note: We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

## Java Constructor IV

**There are three types of constructors in Java:**

- ① Default constructor
- ② No-Arg constructor
- ③ Parameterized constructor

### Java Default Constructor:

```
1 <class_name>()
2 {
3
4 }
```

#### What is the purpose of a default constructor?

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

## Java Constructor V

```
1 //Let us see another example of default constructor which displays the default values
2 class Student3
3 {
4     int id;
5     String name;
6     //method to display the value of id and name
7     void display()
8     {
9         System.out.println(id+" "+name);
10    }
11    public static void main(String args[])
12    {
13        //creating objects
14        Student3 s1=new Student3();
15        Student3 s2=new Student3();
16        s1.display();
17        s2.display();
18    }
19 }
20 Output:
21 @ null
22 @ null
23 In the above class,you are not creating any constructor so compiler provides you a default constructor.
24 Here @ and null values are provided by default constructor.
```

## Java Constructor VI

Rule: If there is no constructor in a class, compiler automatically creates a default constructor.

### Java No-Arg Constructor:

```
1 //Java Program to create and call a default constructor
2 class Bike1
3 {
4     //creating a default constructor
5     Bike1()
6     {
7         System.out.println("Bike is created");
8     }
9     //main method
10    public static void main(String args[])
11    {
12        //calling a default constructor
13        Bike1 b=new Bike1();
14    }
15 }
```

## Java Constructor VII

```
1 public class Main
2 {
3     private String name;
4
5     // constructor
6     Main()
7     {
8         System.out.println("Constructor Called:");
9         name = "Programiz";
10    }
11
12    public static void main(String[] args)
13    {
14
15        // constructor is invoked while
16        // creating an object of the Main class
17        Main obj = new Main();
18        System.out.println("The name is " + obj.name);
19    }
20}
```

# Java Constructor VIII

```
1 public class Main
2 {
3
4     int i;
5
6     // constructor with no parameter
7     private Main()
8     {
9         i = 5;
10        System.out.println("Constructor is called");
11    }
12
13    public static void main(String[] args)
14    {
15
16        // calling the constructor without any parameter
17        Main obj = new Main();
18        System.out.println("Value of i: " + obj.i);
19    }
20}
```

# Java Constructor IX

## Java Parameterized Constructor:

A constructor which has a specific number of parameters is called a parameterized constructor.

## Why use the parameterized constructor?

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

```
1 //Java Program to demonstrate the use of the parameterized constructor.  
2 class Student4  
3 {  
4     int id;  
5     String name;  
6     //creating a parameterized constructor  
7     Student4(int i, String n)  
8     {  
9         id = i;  
10        name = n;  
11    }  
12    void display()  
13    {  
14        System.out.println(id+" "+name);  
15    }  
16}
```

## Java Constructor X

```
15
16     public static void main(String args[])
17     {
18         //creating objects and passing values
19         Student4 s1 = new Student4(111,"Karan");
20         Student4 s2 = new Student4(222,"Aryan");
21         s1.display();
22         s2.display();
23     }
24 }
```

### Constructor Overloading in Java

- ✓ In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.
- ✓ Constructor overloading in Java is a technique of having more than one constructor with different parameter lists.
- ✓ They are arranged in a way that each constructor performs a different task.
- ✓ They are differentiated by the compiler by the number of parameters in the list and their types.

Example of Constructor Overloading

## Java Constructor XII

```
1 //Java program to overload constructors
2 public class Student5
3 {
4     int id;
5     String name;
6     int age;
7     //creating two arg constructor
8     Student5(int i, String n)
9     {
10         id = i;
11         name = n;
12     }
13     //creating three arg constructor
14     Student5(int i, String n, int a)
15     {
16         id = i;
17         name = n;
18         age=a;
19     }
20     void display()
21     {
22         System.out.println(id+" "+name+" "+age);
23     }
24     public static void main(String args[])
25     {
```

## Java Constructor XIII

```
26     Student5 s1 = new Student5(111,"Karan");
27     Student5 s2 = new Student5(222,"Aryan",25);
28     s1.display();
29     s2.display();
30 }
31 }
32 Output:
33
34 111 Karan 0
35 222 Aryan 25
```

## Java Constructor XIV

There are many differences between constructors and methods.

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not be same as the class name.

# Java Constructor XV

## Difference between constructor and method in Java

- 
- 1 A constructor is used to initialize the state of an object.
  - 2 A constructor must not have a return type.
  - 3 The constructor is invoked implicitly.
  - 4 The Java compiler provides a default constructor if you don't have any constructor in a class.
  - 5 The constructor name must be same as the class name.
  - 1 A method is used to expose the behavior of an object.
  - 2 A method must have a return type.
  - 3 The method is invoked explicitly.
  - 4 The method is not provided by the compiler in any case.
  - 5 The method name may or may not be same as class name.

### Java Copy Constructor

- ✓ There is no copy constructor in Java. However, we can copy the values from one object to another like copy constructor in C++.
- ✓ There are many ways to copy the values of one object into another in Java. They are:
  - By constructor
  - By assigning the values of one object into another
  - By clone() method of Object class

In this example, we are going to copy the values of one object into another using Java constructor.

## Java Constructor XVII

```
1 //Java program to initialize the values from one object to another object.
2 class Student6
3 {
4     int id;
5     String name;
6     //constructor to initialize integer and string
7     Student6(int i, String n)
8     {
9         id = i;
10        name = n;
11    }
12    //constructor to initialize another object
13    Student6(Student6 s)
14    {
15        id = s.id;
16        name = s.name;
17    }
18    void display()
19    {
20        System.out.println(id+" "+name);
21    }
22
23    public static void main(String args[])
24    {
25        Student6 s1 = new Student6(111, "Karan");
```

## Java Constructor XVIII

```
26     Student6 s2 = new Student6(s1);
27     s1.display();
28     s2.display();
29 }
30 }
31 Output:
32
33 111 Karan
34 111 Karan
```

## Java Constructor XIX

### Copying values without constructor

We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

```
1 class Student7
2 {
3     int id;
4     String name;
5     Student7(int i, String n)
6     {
7         id = i;
8         name = n;
9     }
10    Student7()
11    {
12    }
13
14    void display()
15    {
16        System.out.println(id+" "+name);
17    }
18
```

## Java Constructor XX

```
19 public static void main(String args[])
20 {
21     Student7 s1 = new Student7(111,"Karan");
22     Student7 s2 = new Student7();
23     s2.id=s1.id;
24     s2.name=s1.name;
25     s1.display();
26     s2.display();
27 }
28 }
29 Output:
30
31 111 Karan
32 111 Karan
```

# Java Constructor XXI

## Calling another constructor:

- Avoids redundancy between constructors
- Only a constructor (not a method) can call another constructor

```
1 public class Main
2 {
3
4     int sum;
5
6     // first constructor
7     Main()
8     {
9         // calling the second constructor
10        this(5, 2);
11    }
12
13    // second constructor
14    Main(int arg1, int arg2)
15    {
16        // add two value
17        this.sum = arg1 + arg2;
18    }
19
```

## Java Constructor XXII

```
20 void display()
21 {
22     System.out.println("Sum is: " + sum);
23 }
24
25 // main class
26 public static void main(String[] args)
27 {
28
29     // call the first constructor
30     Main obj = new Main();
31
32     // call display method
33     obj.display();
34 }
```

## Java Constructor XXIII

### Calling the constructor of the superclass from the constructor of the child class:

We can also call the constructor of the superclass from the constructor of child class using super().

```
1 // superclass
2 class Languages
3 {
4     // constructor of the superclass
5     Languages(int version1, int version2)
6     {
7
8         if (version1 > version2)
9         {
10             System.out.println("The latest version is: " + version1);
11         }
12         else
13         {
14             System.out.println("The latest version is: " + version2);
15         }
16     }
17 }
18 }
```

## Java Constructor XXIV

```
19
20 // child class
21 public class Main extends Languages
22 {
23     // constructor of the child class
24     Main()
25     {
26         // calling the constructor of super class
27         super(11, 8);
28     }
29     // main method
30     public static void main(String[] args)
31     {
32         // call the first constructor
33         Main obj = new Main();
34     }
35 }
36 Output
37 The latest version is: 11
```

## Java Constructor XXV

Q) Does constructor return any value?

Yes, it is the current class instance (You cannot use return type yet it returns a value).

Q) Can constructor perform other tasks instead of initialization?

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

Q) Can we define a static constructor in Java?

No, we cannot define a static constructor in Java, If we are trying to define a constructor with the static keyword a compile-time error will occur.

In general, static means class level. A constructor will be used to assign initial values for the instance variables. Both static and constructor are

## Java Constructor XXVI

different and opposite to each other. We need to assign initial values for an instance variable we can use a constructor. We need to assign static variables we can use Static Blocks.

Q) Is there Constructor class in Java?

Yes.

Q) What is the purpose of Constructor class?

Java provides a Constructor class which can be used to get the internal information of a constructor in the class. It is found in the `java.lang.reflect` package.

# CS204:Object Oriented Programming

## Concepts

**Sachchida Nand Chaurasia**  
Assistant Professor

Department of Computer Science  
Banaras Hindu University  
Varanasi

Email id: [snchaurasia@bhu.ac.in](mailto:snchaurasia@bhu.ac.in), [sachchidanand.mca07@gmail.com](mailto:sachchidanand.mca07@gmail.com)



February 1, 2021

## Access Modifiers I

As the name suggests access modifiers in Java helps to restrict the scope of a class, constructor, variable, method, or data member.

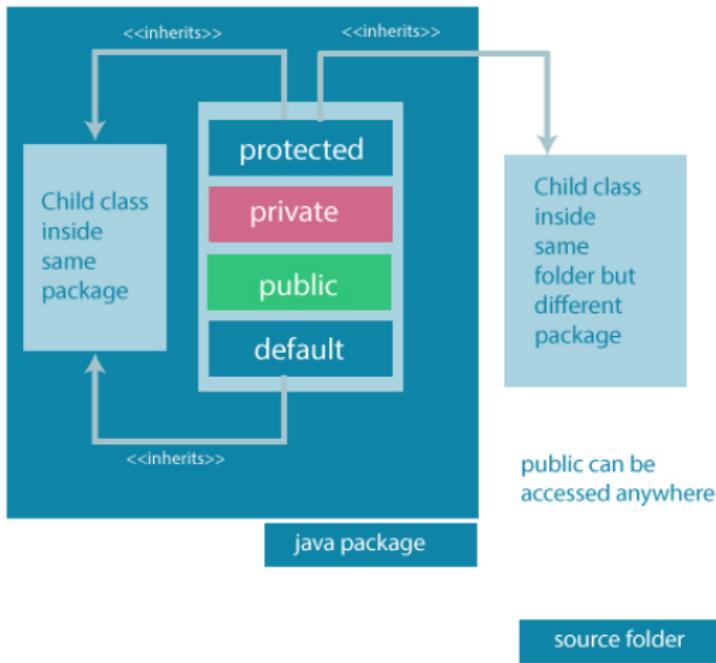
✓ There are four types of access modifiers available in java:

- ① Default – No keyword required
- ② Private
- ③ Protected
- ④ Public

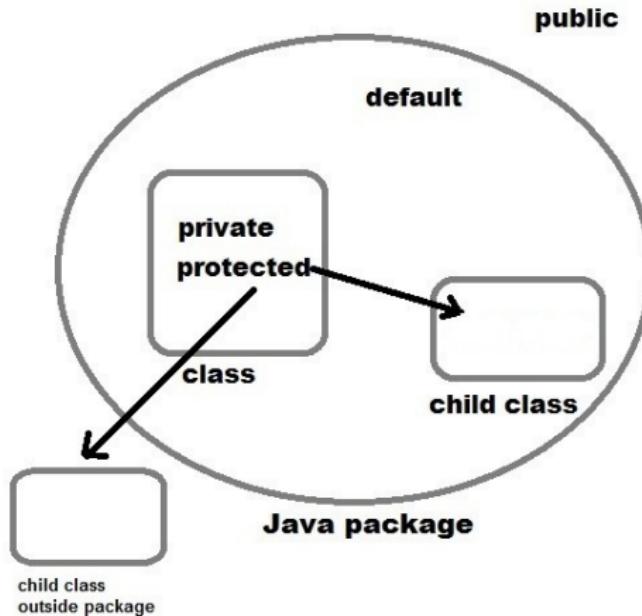
## Access Modifiers II

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

# Access Modifiers III



## Access Modifiers IV



## Access Modifiers V

- ✓ Default: When no access modifier is specified for a class, method, or data member – It is said to be having the default access modifier by default.
- ✓ The data members, class or methods which are not declared using any access modifiers i.e. having default access modifier are accessible only within the same package.
- ✓ In this example, we will create two packages and the classes in the packages will be having the default access modifiers and we will try to access a class from one package from a class of the second package.

## Access Modifiers VI

```
1 // Java program to illustrate default modifier
2 package p1;
3 // Class McaMsc is having Default access modifier
4 class McaMsc
5 {
6     void display()
7     {
8         System.out.println("Hello World!");
9     }
10 }

1 // Java program to illustrate error while using class from different package with default modifier
2 package p2;
3 import p1.*;
4 // This class is having default access modifier
5 class McaMscNew
6 {
7     public static void main(String args[])
8     {
9         // Accessing class McaMsc from package p1
10        McaMscs obj = new McaMsc();
11        obj.display();
12    }
13 }
```

## Access Modifiers VII

**Private:** The private access modifier is specified using the keyword `private`.

- ✓ The methods or data members declared as `private` are accessible only within the class in which they are declared.
- ✓ Any other class of the same package will not be able to access these members.
- ✓ Methods declared `private` are not inherited at all, so there is no rule for them.
- ✓ Top-level classes or interfaces can not be declared as `private` because `private` means “only visible within the enclosing class”. `protected` means “only visible within the enclosing class and any subclasses”.

## Access Modifiers VIII

- ✓ In this example, we will create two classes A and B within the same package p1. We will declare a method in class A as private and try to access this method from class B and see the result.

```
1 // Java program to illustrate error while using class from different package with private modifier
2 package p1;
3 class A
4 {
5     private void display()
6     {
7         System.out.println("McaMscsforMcaMscs");
8     }
9 }
10 class B
11 {
12     public static void main(String args[])
13     {
14         A obj = new A();
15         // Trying to access private method
16         // of another class
17         obj.display();
18     }
19 }
20 Output:
```

## Access Modifiers IX

```
21  
22 error: display() has private access in A  
23 obj.display();
```

## Access Modifiers X

**protected:** The protected access modifier is specified using the keyword `protected`.

- ✓ Variables, methods, and constructors, which are declared `protected` in a superclass can be accessed only by the subclasses in other package or any class within the package of the `protected` members' class.
- ✓ You can also say that the `protected` access modifier is similar to default access modifier with one exception that it has visibility in sub classes.
- ✓ The `protected` access modifier cannot be applied to class and interfaces. Methods, fields can be declared `protected`, however methods and fields in a interface cannot be declared `protected`.
- ✓ Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

## Access Modifiers XI

- ✓ In this example, we will create two packages p1 and p2. Class A in p1 is made public, to access it in p2. The method display in class A is protected and class B is inherited from class A and this protected method is then accessed by creating an object of class B.

## Access Modifiers XII

```
1 // Java program to illustrate protected modifier
2 package p1;
3 public class A
4 {
5     protected void display()
6     {
7         System.out.println("Hello World");
8     }
9 }
```

```
1 // Java program to illustrate protected modifier
2 package p2;
3 import p1.*; // importing all classes in package p1
4 // Class B is subclass of A
5 class B extends A
6 {
7     public static void main(String args[])
8     {
9         B obj = new B();
10        obj.display();
11    }
12
13 }
14 Output:
Hello World
```

## Access Modifiers XIII

**public:** The public access modifier is specified using the keyword public.

- ✓ The public access modifier has the widest scope among all other access modifiers.
- ✓ Classes, methods, or data members that are declared as public are accessible from everywhere in the program. There is no restriction on the scope of public data members.

## Access Modifiers XIV

```
1 // Java program to illustrate
2 // public modifier
3 package p1;
4 public class A
5 {
6     public void display()
7     {
8         System.out.println("McaMscsforMcaMscs");
9     }
10}
11package p2;
12import p1.*;
13class B
14{
15    public static void main(String args[])
16    {
17        A obj = new A();
18        obj.display();
19    }
20}
```

## Access Modifiers XV

- ✓ Access modifiers are mainly used for encapsulation. It can help us to control what part of a program can access the members of a class. So that misuse of data can be prevented.

# CS204:Object Oriented Programming

## Concepts

**Sachchida Nand Chaurasia**  
Assistant Professor

Department of Computer Science  
Banaras Hindu University  
Varanasi

Email id: [snchaurasia@bhu.ac.in](mailto:snchaurasia@bhu.ac.in), [sachchidanand.mca07@gmail.com](mailto:sachchidanand.mca07@gmail.com)



February 10, 2021

## Interfaces I

Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

- Interfaces specify what a class must do and not how. It is the blueprint of the class.
- An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.

## Interfaces II

- A Java library example is, Comparator Interface. If a class implements this interface, then it can be used to sort a collection.

Syntax :

```
1  interface <interface_name>
2  {
3      // declare constant fields
4      // declare methods that abstract
5      // by default.
6 }
```

- ✓ To declare an interface, use *interface* keyword.
- ✓ It is used to provide total abstraction.
- ✓ That means all the methods in an interface are declared with an empty body and are public and all fields are public, static and final by default.

## Interfaces III

- ✓ A class that implements an interface must implement all the methods declared in the interface. To implement interface use implements keyword.

### Why do we use interface ?

- It is used to achieve total abstraction.
- Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance .
- It is also used to achieve loose coupling.
- Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?

The reason is, abstract classes may contain non-final variables, whereas variables in interface are final, public and static.

## Interfaces IV

```
1 // A simple interface
2 interface Player
3 {
4     final int id = 10;
5     int move();
6 }
```

- ✓ To implement an interface we use keyword: implements  
interface Polygon void getArea(int length, int breadth);

```
1 // implement the Polygon interface
2 class Rectangle implements Polygon
3 {
4
5     // implementation of abstract method
6     public void getArea(int length, int breadth)
7     {
8         System.out.println("The area of the rectangle is " + (length * breadth));
9     }
10 }
11
12 class Main
13 {
14     public static void main(String[] args)
```

# Interfaces V

```
15      {
16          Rectangle r1 = new Rectangle();
17          r1.getArea(5, 6);
18      }
19 }

1 // create an interface
2 interface Language
3 {
4     void getName(String name);
5 }

6

7 // class implements interface
8 class ProgrammingLanguage implements Language
9 {
10
11     // implementation of abstract method
12     public void getName(String name)
13     {
14         System.out.println("Programming Language: " + name);
15     }
16 }
17
18 class Main
19 {
20     public static void main(String[] args)
```

# Interfaces VI

```
21     {
22         ProgrammingLanguage language = new ProgrammingLanguage();
23         language.getName("Java");
24     }
25 }
```

```
1 // Java program to demonstrate working of
2 // interface.
3 import java.io.*;
4 // A simple interface
5 interface In1
6 {
7     // public, static and final
8     final int a = 10;
9     // public and abstract
10    void display();
11 }
12 // A class that implements the interface.
13 class TestClass implements In1
14 {
15     // Implementing the capabilities of
16     // interface.
17     public void display()
18     {
19         System.out.println("Geek");
20     }
}
```

## Interfaces VII

```
21 // Driver Code  
22 public static void main (String[] args)  
23 {  
24     TestClass t = new TestClass();  
25     t.display();  
26     System.out.println(a);  
27 }  
28 }
```

### A real-world example:

Let's consider the example of vehicles like bicycle, car, bike....., they have common functionalities. So we make an interface and put all these common functionalities. And lets Bicycle, Bike, car . . .etc implement all these functionalities in their own class in their own way.

# Interfaces VIII

```
1 import java.io.*;
2
3 interface Vehicle
4 {
5
6     // all are the abstract methods.
7     void changeGear(int a);
8     void speedUp(int a);
9     void applyBrakes(int a);
10 }
11
12 class Bicycle implements Vehicle
13 {
14
15     int speed;
16     int gear;
17
18     // to change gear
19     @Override
20     public void changeGear(int newGear)
21     {
22
23         gear = newGear;
24     }
25 }
```

# Interfaces IX

```
26 // to increase speed
27 @Override
28 public void speedUp(int increment)
29 {
30
31     speed = speed + increment;
32 }
33
34 // to decrease speed
35 @Override
36 public void applyBrakes(int decrement)
37 {
38
39     speed = speed - decrement;
40 }
41
42 public void printStates()
43 {
44     System.out.println("speed: " + speed
45     + " gear: " + gear);
46 }
47 }
48
49 class Bike implements Vehicle
50 {
```

# Interfaces X

```
51
52     int speed;
53     int gear;
54
55     // to change gear
56     @Override
57     public void changeGear(int newGear)
58     {
59
60         gear = newGear;
61     }
62
63     // to increase speed
64     @Override
65     public void speedUp(int increment)
66     {
67
68         speed = speed + increment;
69     }
70
71     // to decrease speed
72     @Override
73     public void applyBrakes(int decrement)
74     {
75 }
```

# Interfaces XI

```
76         speed = speed - decrement;
77     }
78
79     public void printStates()
80     {
81         System.out.println("speed: " + speed
82             + " gear: " + gear);
83     }
84
85 }
86 class GFG
87 {
88
89     public static void main (String[] args)
90     {
91
92         // creating an instance of Bicycle
93         // doing some operations
94         Bicycle bicycle = new Bicycle();
95         bicycle.changeGear(2);
96         bicycle.speedUp(3);
97         bicycle.applyBrakes(1);
98
99         System.out.println("Bicycle present state :");
100        bicycle.printStates();
```

## Interfaces XII

```
01
02     // creating instance of the bike.
03     Bike bike = new Bike();
04     bike.changeGear(1);
05     bike.speedUp(4);
06     bike.applyBrakes(3);
07
08     System.out.println("Bike present state :");
09     bike.printStates();
10 }
11 }
```

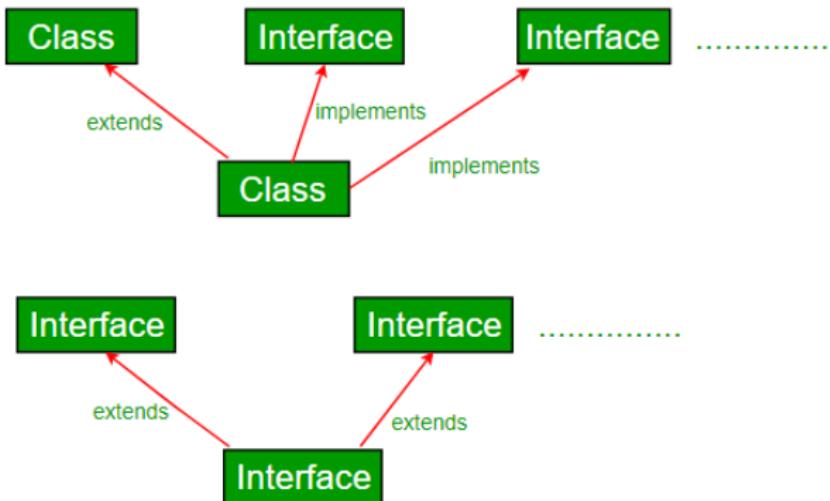
### Important points about interface or summary of article:

- We can't create instance(interface can't be instantiated) of interface but we can make reference of it that refers to the Object of its implementing class.
- A class can implement more than one interface.
- An interface can extends another interface or interfaces (more than one interface) .
- A class that implements interface must implements all the methods in interface.
- All the methods are public and abstract. And all the fields are public, static, and final.
- It is used to achieve multiple inheritance.

## Interfaces XIV

- It is used to achieve loose coupling.

### Implementing Multiple Interfaces



✓ In Java, a class can also implement multiple interfaces.

# Interfaces XV

```
1  interface A
2  {
3      // members of A
4  }
5  interface B
6  {
7      // members of B
8  }
9  class C implements A, B
10 {
11     // abstract members of A
12     // abstract members of B
13 }
```

# Interfaces XVI

```
1 // Java program to demonstrate that a class can
2 // implement multiple interfaces
3 import java.io.*;
4 interface intfA
5 {
6     void m1();
7 }
8
9 interface intfB
10 {
11     void m2();
12 }
13
14 // class implements both interfaces
15 // and provides implementation to the method.
16 class sample implements intfA, intfB
17 {
18     @Override
19     public void m1()
20     {
21         System.out.println("Welcome: inside the method m1");
22     }
23
24     @Override
25     public void m2()
```

# Interfaces XVII

```
26     {
27         System.out.println("Welcome: inside the method m2");
28     }
29 }
30
31 class GFG
32 {
33     public static void main (String[] args)
34     {
35         sample ob1 = new sample();
36
37         // calling the method implemented
38         // within the class.
39         ob1.m1();
40         ob1.m2();
41     }
42 }
```

## Interfaces XVIII

### Extending an Interface

- ✓ Similar to classes, interfaces can extend other interfaces. The extends keyword is used for extending interfaces. For example,

```
1 interface Line
2 {
3     // members of Line interface
4 }
5
6 // extending interface
7 interface Polygon extends Line
8 {
9     // members of Polygon interface
10    // members of Line interface
11 }
```

# CS204:Object Oriented Programming

## Concepts

**Sachchida Nand Chaurasia**  
Assistant Professor

Department of Computer Science  
Banaras Hindu University  
Varanasi

Email id: [snchaurasia@bhu.ac.in](mailto:snchaurasia@bhu.ac.in), [sachchidanand.mca07@gmail.com](mailto:sachchidanand.mca07@gmail.com)



February 15, 2021

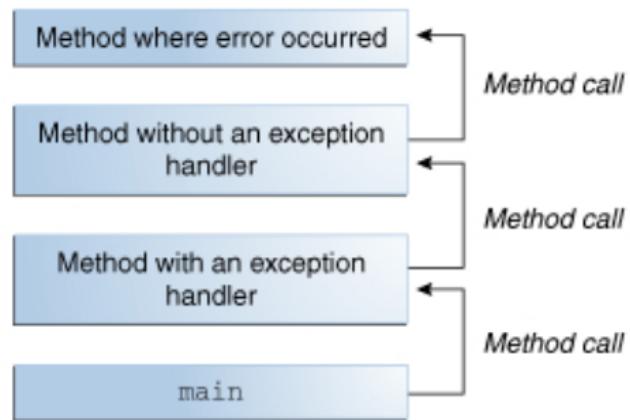
## Exception Handling I

**Definition:** An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

- ✓ The term exception is shorthand for the phrase "exceptional event."
- ✓ When an error occurs within a method, the method creates an object and hands it off to the run-time system (JVM).
- ✓ The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred.
- ✓ Creating an exception object and handing it to the run-time system is called **throwing an exception**.
- ✓ After a method throws an exception, the run-time system attempts to **find something to handle it**.

## Exception Handling II

- ✓ The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred.
- ✓ The list of methods is known as the **call stack**.



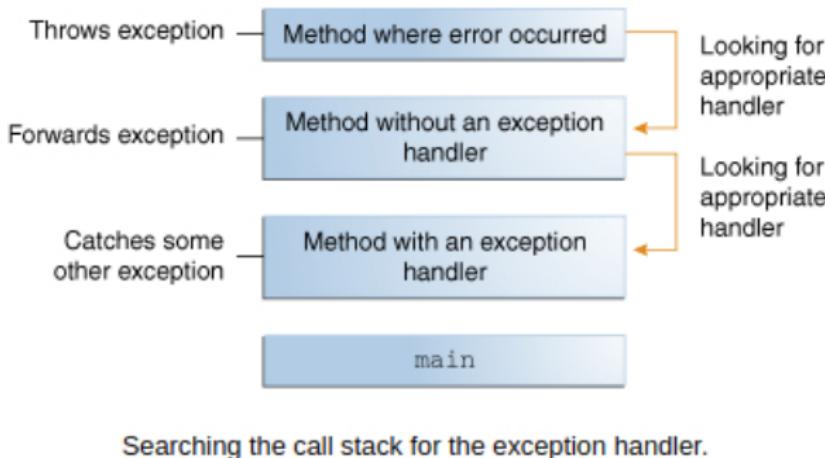
The call stack.

## Exception Handling III

- ✓ The run-time system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an **exception handler**.
- ✓ The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called.
- ✓ When an appropriate handler is found, the run-time system passes the exception to the handler.
- ✓ An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.
- ✓ The exception handler chosen is said to **catch the exception**.

## Exception Handling IV

- ✓ If the run-time system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the next figure, the run-time system (and, consequently, the program) terminates abnormally.



## Exception Handling V

- ✓ This handler prints the exception information in the following format and terminates program abnormally.

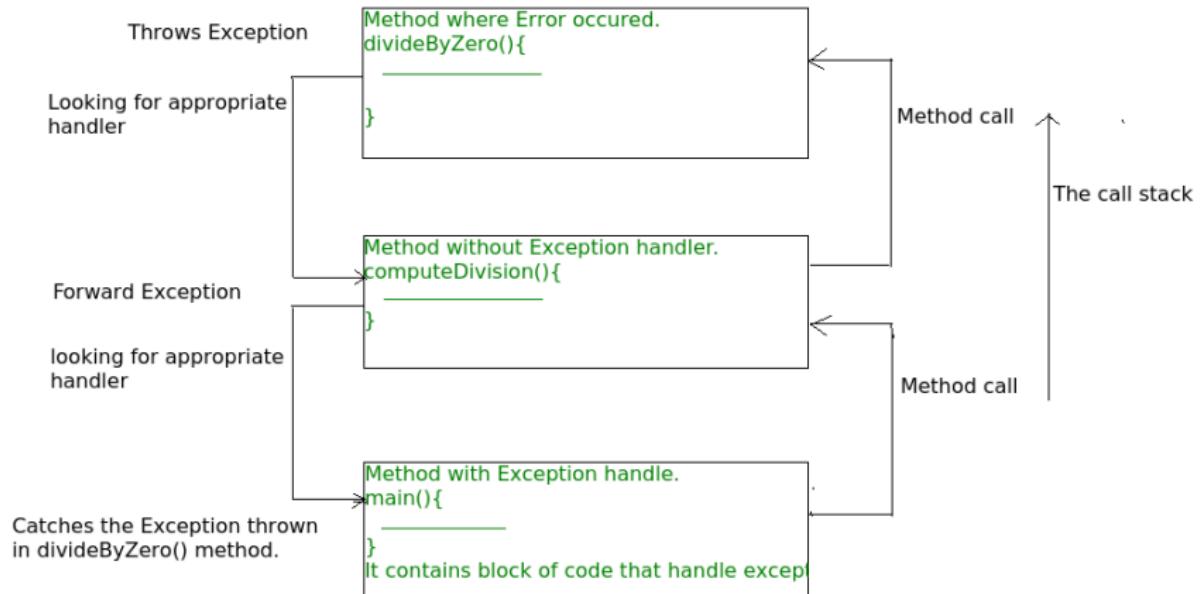
```
1 Exception in thread "xxx" Name of Exception : Description  
2 ... ..... . // Call Stack
```

### Example:

```
1 // Java program to demonstrate how exception is thrown.  
2 public class ThrowsExcp  
3 {  
4     public static void main(String args[])  
5     {  
6         String str = null;  
7         System.out.println(str.length());  
8     }  
9 }  
10 Output :  
11 Exception in thread "main" java.lang.NullPointerException at ThrowsExcp.main(ThrowsExcp.java:6)
```

## Exception Handling VI

The below diagram to understand the flow of the call stack.



The call stack and searching the call stack for exception handler.

## Exception Handling VII

- ✓ An example that illustrate how run-time system searches appropriate exception handling code on the call stack :

```
1 //Java program to demonstrate exception is thrown how the runTime system searches th call stack to find appropriate
2 class ExceptionThrown
3 {
4     // It throws the Exception(ArithmetricException).
5     // Appropriate Exception handler is not found within this method.
6     static int divideByZero(int a, int b)
7     {
8         // this statement will cause ArithmetricException(/ by zero)
9         int i = a/b;
10        return i;
11    }
12    //The runTime System searches the appropriate Exception handler in this method also
13    // but couldn't have found. So looking forward on the call stack.
14    static int computeDivision(int a, int b)
15    {
16        int res =0;
17        try
18        {
19            res = divideByZero(a,b);
20        }
21        //doesn't matches with ArithmetricException
22        catch(NumberFormatException ex)
```

# Exception Handling VIII

```
23
24     {
25         System.out.println("NumberFormatException is occurred");
26     }
27     return res;
28 }
29 // In this method found appropriate Exception handler. i.e. matching catch block.
30 public static void main(String args[])
31 {
32     int a = 1;
33     int b = 0;
34     try
35     {
36         int i = computeDivision(a,b);
37     }
38     // matching ArithmeticException
39     catch(ArithmeticException ex)
40     {
41         // getMessage will print description of exception(here / by zero)
42         System.out.println(ex.getMessage());
43     }
44 }
45 Output :
46 / by zero.
```

## Exception Handling IX

### Advantage of Exception Handling:

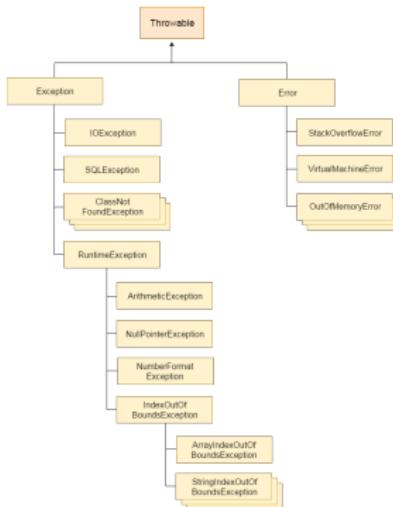
- ✓ The core advantage of exception handling is to maintain the normal flow of the application.
- ✓ An exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

```
1 statement 1;  
2 statement 2;  
3 statement 3;  
4 statement 4;  
5 statement 5; //exception occurs  
6 statement 6;  
7 statement 7;  
8 statement 8;  
9 statement 9;  
10 statement 10;
```

## Exception Handling X

### Hierarchy of Java Exception classes:

✓ The `java.lang.Throwable` class is the root class of Java Exception hierarchy which is inherited by two subclasses: `Exception` and `Error`. A hierarchy of Java Exception classes are given below:



### Types of Java Exceptions:

✓ There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

- Checked Exception
- Unchecked Exception
- Error

### Error vs Exception

**Error:** An Error indicates serious problem that a reasonable application should not try to catch.

**Exception:** Exception indicates conditions that a reasonable application might try to catch.

## Exception Handling XII

### Difference between Checked and Unchecked Exceptions:

- ① Checked Exception: The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

*✓ For example, consider the following Java program that opens file at location "C:\test\aa.txt" and prints the first three lines of it. The program doesn't compile, because the function main() uses FileReader() and FileReader() throws a checked exception FileNotFoundException. It also uses readLine() and close() methods, and these methods also throw checked exception IOException*

# Exception Handling XIII

```
1 import java.io.*;
2 class Main
3 {
4     public static void main(String[] args)
5     {
6         FileReader file = new FileReader("C:\test\a.txt");
7         BufferedReader fileInput = new BufferedReader(file);
8
9         // Print first 3 lines of file "C:\test\a.txt"
10        for (int counter = 0; counter < 3; counter++)
11            System.out.println(fileInput.readLine());
12
13        fileInput.close();
14    }
15 }
16 Output:
17
18 Exception in thread "main" java.lang.RuntimeException: Uncompilable source code -
19 unreported exception java.io.FileNotFoundException; must be caught or declared to be
20 thrown
21 at Main.main(Main.java:5)
```

## Exception Handling XIV

- ✓ To fix the above program, we either need to specify list of exceptions using throws, or we need to use try-catch block. Since FileNotFoundException is a subclass of IOException, we can just specify IOException in the throws list and make the above program compiler-error-free.

```
1 import java.io.*;
2 class Main
3 {
4     public static void main(String[] args) throws IOException
5     {
6         FileReader file = new FileReader("C:\\test\\a.txt");
7         BufferedReader fileInput = new BufferedReader(file);
8         // Print first 3 lines of file "C:\\test\\a.txt"
9         for (int counter = 0; counter < 3; counter++)
10            System.out.println(fileInput.readLine());
11            fileInput.close();
12    }
13 }
14 Output: First three lines of file "C:\\test\\a.txt"
```

## Exception Handling XV

- ② Unchecked Exception: The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at run-time.

```
1 +-----+
2 | Throwable |
3 +-----+
4 / \
5 / \
6 +---+ +-----+
7 | Error | | Exception |
8 +---+ +-----+
9 / | \ / | \
10 \____/ \____/ \
11 +-----+
12 unchecked checked| RuntimeException |
13 +-----+
14 / || \
15 \_____/
16 unchecked
```

## Exception Handling XVI

- ✓ Consider the following Java program. It compiles fine, but it throws ArithmeticException when run. The compiler allows it to compile, because ArithmeticException is an unchecked exception.

```
1 class Main
2 {
3     public static void main(String args[])
4     {
5         int x = 0;
6         int y = 10;
7         int z = y/x;
8     }
9 }
10 Output:
11
12 Exception in thread "main" java.lang.ArithmetricException: / by zero at Main.main(Main.java:5)
13 Java Result: 1
```

- ③ Error: Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, Assertion Error etc.

## Exception Handling XVII

### Common Scenarios of Java Exceptions:

- ✓ There are given some scenarios where unchecked exceptions may occur. They are as follows:

- ① A scenario where ArithmeticException occurs: If we divide any number by zero, there occurs an ArithmeticException.

```
1 int a=50/0; //ArithmetiException
```

- ② A scenario where NullPointerException occurs: If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

## Exception Handling XVIII

```
1 // Java program to demonstrate how exception is thrown.
2 class ThrowsExecp
3 {
4     public static void main(String args[])
5     {
6         String str = null;
7         System.out.println(str.length()); //NullPointerException
8     }
9 }
```

- ③ A scenario where NumberFormatException occurs: The wrong formatting of any value may occur NumberFormatException.
- Suppose I have a string variable that has characters, converting this variable into digit will occur NumberFormatException.
- ✓ Since NumberFormatException occurs due to the inappropriate format of string for the corresponding argument of the method

## Exception Handling XIX

which is throwing the exception, there can be various ways of it. A few of them are mentioned as follows-

- The input string provided might be null-  
Example- `Integer.parseInt(null);`
- The input string might be empty-  
Example- `Integer.parseInt("");`
- The input string might be having trailing space-  
Example- `Integer.parseInt("123 ")`;
- The input string might be having a leading space-  
Example- `Integer.parseInt(" 123")`;
- The input string may be alphanumeric-  
Example- `Long.parseLong("b2")`;

## Exception Handling XX

- The input string may have an input which might exceed the range of the datatype storing the parsed string-

Example- Integer.parseInt("135"); The maximum possible value of integer can be 127, but the value in the string is 135 which is out of range, so this will throw the exception.

- There may be a mismatch between the input string and the type of the method which is being used for parsing. If you provide the input string like "1.0" and you try to convert this string into an integer value, it will throw a NumberFormatException exception. Example- Integer.parseInt("1..0");

```
1 String s="abc";
2 int i=Integer.parseInt(s);
```

# Exception Handling XXI

```
1 public class Example
2 {
3     public static void main(String[] args)
4     {
5         int a = Integer.parseInt(null);
6         //throws Exception as //the input string is of illegal format for parsing as it is null.
7     }
}
```

## Exception Handling XXII

### How Programmer handles an exception?

- ✓ Customized Exception Handling : Java exception handling is managed via five keywords: **try, catch, throw, throws, and finally**.
- ✓ Briefly, here is how they work. Program statements that you think can raise exceptions are contained within a **try block**.
- ✓ If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch block) and handle it in some rational manner.
- ✓ System-generated exceptions are automatically thrown by the Java run-time system.
- ✓ To manually throw an exception, use the keyword `throw`.
- ✓ Any exception that is thrown out of a method must be specified as such by a `throws` clause.

## Exception Handling XXIII

- ✓ Any code that absolutely must be executed after a try block completes is put in a finally block.

## Exception Handling XXIV

### Java Exception Keywords:

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

## Exception Handling XXV

### Need of try-catch clause(Customized Exception Handling):

```
1 // java program to demonstrate
2 // need of try-catch clause
3 class GFG
4 {
5     public static void main (String[] args)
6     {
7         // array of size 4.
8         int[] arr = new int[4];
9         // this statement causes an exception
10        int i = arr[4];
11        // the following statement will never execute
12        System.out.println("Hi, I want to execute");
13    }
14 }
```

# Exception Handling XXVI

## How to use try-catch clause

```
1  try
2  {
3      // block of code to monitor for errors
4      // the code you think can raise an exception
5  }
6  catch (ExceptionType1 ex0b)
7  {
8      // exception handler for ExceptionType1
9  }
10 catch (ExceptionType2 ex0b)
11 {
12     // exception handler for ExceptionType2
13 }
14 // optional
15 finally
16 {
17     // block of code to be executed after try block ends
18 }
```

## Exception Handling XXVII

### Java Exception Handling Example:

```
1 public class JavaExceptionExample
2 {
3     public static void main(String args[])
4     {
5         try
6         {
7             //code that may raise exception
8             int data=100/0;
9         }
10        catch(ArithmaticException e)
11        {
12            System.out.println(e);
13        }
14        //rest code of the program
15        System.out.println("rest of the code...");
16    }
17}
```

- ✓ In the above example, 100/0 raises an ArithmaticException which is handled by a try-catch block.

## Exception Handling XXVIII

### Points to remember :

- In a method, there can be more than one statements that might throw exception, So put all these statements within its own try block and provide separate exception handler within own catch block for each of them.
- If an exception occurs within the try block, that exception is handled by the exception handler associated with it. To associate exception handler, we must put catch block after it. There can be more than one exception handlers. Each catch block is a exception handler that handles the exception of the type indicated by its argument. The argument, ExceptionType declares the type of the

## Exception Handling XXIX

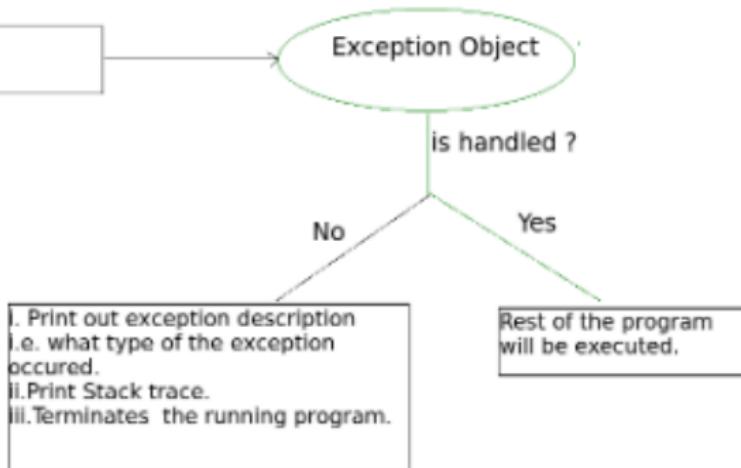
exception that it can handle and must be the name of the class that inherits from Throwable class.

- For each try block there can be zero or more catch blocks, but only one finally block.
- The finally block is optional. It always gets executed whether an exception occurred in try block or not . If exception occurs, then it will be executed after try and catch blocks. And if exception does not occur then it will be executed after the try block. The finally block in java is used to put important codes such as clean up code e.g. closing the file or closing the connection.

# Exception Handling XXX

An Exception Object is created and thrown.

```
int a = 10/0;
```



# Exception Handling XXXI

## Example 1:

```
1 public class TryCatchExample1
2 {
3     public static void main(String[] args)
4     {
5         int data=50/0; //may throw exception
6         System.out.println("rest of the code");
7     }
8 }
```

## Solution:

```
1 public class TryCatchExample2
2 {
3     public static void main(String[] args)
4     {
5         try
6         {
7             int data=50/0; //may throw exception
8         }
9         //handling the exception
10        catch(ArithmetcException e)
11        {
12            System.out.println(e);
13        }
14 }
```

## Exception Handling XXXII

```
14     System.out.println("rest of the code");
15 }
16 }
```

## Exception Handling XXXIII

### Example 2:

- ✓ In this example, we also kept the code in a try block that will not throw an exception.

```
1 public class TryCatchExample3
2 {
3     public static void main(String[] args)
4     {
5         try
6         {
7             int data=50/0; //may throw exception
8             // if exception occurs, the remaining statement will not execute
9             System.out.println("rest of the code");
10        }
11        // handling the exception
12        catch(ArithmaticException e)
13        {
14            System.out.println(e);
15        }
16    }
17 }
```

- ✓ Solution:

## Exception Handling XXXIV

```
1 public class TryCatchExample4
2 {
3     public static void main(String[] args)
4     {
5         try
6         {
7             int data=50/0; //may throw exception
8         }
9         // handling the exception by using Exception class
10        catch(Exception e)
11        {
12            System.out.println(e);
13        }
14        System.out.println("rest of the code");
15    }
16}
```

## Exception Handling XXXV

### Example 3: an example to print a custom message on exception.

```
1 public class TryCatchExample5
2 {
3     public static void main(String[] args)
4     {
5         try
6         {
7             int data=50/0; //may throw exception
8         }
9         // handling the exception
10        catch(Exception e)
11        {
12            // displaying the custom message
13            System.out.println("Can't divided by zero");
14        }
15    }
16 }
```

✓ Solution: an example to resolve the exception in a catch block.

## Exception Handling XXXVI

```
1 public class TryCatchExample6
2 {
3     public static void main(String[] args)
4     {
5         int i=50;
6         int j=0;
7         int data;
8         try
9         {
10             data=i/j; //may throw exception
11         }
12         // handling the exception
13         catch(Exception e)
14         {
15             // resolving the exception in catch block
16             System.out.println(i/(j+2));
17         }
18     }
19 }
```

## Exception Handling XXXVII

**Example 4: along with try block, we also enclose exception code in a catch block.**

```
1 public class TryCatchExample7
2 {
3     public static void main(String[] args)
4     {
5         try
6         {
7             int data1=50/0; //may throw exception
8         }
9         // handling the exception
10        catch(Exception e)
11        {
12            // generating the exception in catch block
13            int data2=50/0; //may throw exception
14        }
15        System.out.println("rest of the code");
16    }
17}
```

## Exception Handling XXXVIII

- ✓ Solution: handle the generated exception (Arithmetic Exception) with a different type of exception class (ArrayIndexOutOfBoundsException).

```
1 public class TryCatchExample8
2 {
3     public static void main(String[] args)
4     {
5         try
6         {
7             int data=50/0; //may throw exception
8         }
9         // try to handle the ArithmeticException using ArrayIndexOutOfBoundsException
10        catch(ArrayIndexOutOfBoundsException e)
11        {
12            System.out.println(e);
13        }
14        System.out.println("rest of the code");
15    }
16}
```

## Exception Handling XXXIX

### Example 5: to handle another unchecked exception.

```
1 public class TryCatchExample9
2 {
3     public static void main(String[] args)
4     {
5         try
6         {
7             int arr[]=
8             {
9                 1,3,5,7
10            }
11            ;
12            System.out.println(arr[10]); //may throw exception
13        }
14        // handling the array exception
15        catch(ArrayIndexOutOfBoundsException e)
16        {
17            System.out.println(e);
18        }
19        System.out.println("rest of the code");
20    }
21 }
```

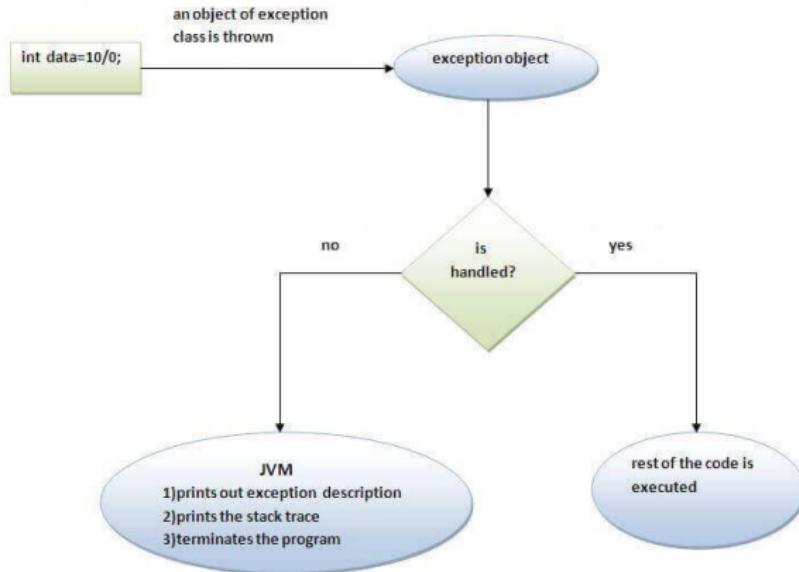
# Exception Handling XL

## Example 5: To handle checked exception.

```
1 import java.io.FileNotFoundException;
2 import java.io.PrintWriter;
3 public class TryCatchExample10
4 {
5     public static void main(String[] args)
6     {
7         PrintWriter pw;
8         try
9         {
10             pw = new PrintWriter("jtp.txt"); //may throw exception
11             pw.println("saved");
12         }
13         // providing the checked exception handler
14         catch (FileNotFoundException e)
15         {
16             System.out.println(e);
17         }
18         System.out.println("File saved successfully");
19     }
20 }
```

# Exception Handling XLI

## Internal working of java try-catch block:



## Exception Handling XLII

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

## Exception Handling XLIII

✓ Let us see an example that illustrate how run-time system searches appropriate exception handling code on the call stack :

```
1 //Java program to demonstrate exception is thrown how the runTime system searches th call stack to find appropriate
2 class ExceptionThrown
3 {
4     //It throws the Exception(ArithmetricException).
5     // Appropriate Exception handler is not found within this method.
6     static int divideByZero(int a, int b)
7     {
8         // this statement will cause ArithmetricException(/ by zero)
9         int i = a/b;
10        return i;
11    }
12    //The runTime System searches the appropriate Exception handler in this method also but couldn't have found
13    // So looking forward
14    // on the call stack.
15    static int computeDivision(int a, int b)
16    {
17        int res =0;
18        try
19        {
20            res = divideByZero(a,b);
21        }
22        // doesn't matches with ArithmetricException
```

## Exception Handling XLIV

```
22         catch(NumberFormatException ex)
23     {
24         System.out.println("NumberFormatException is occurred");
25     }
26     return res;
27 }
28 // In this method found appropriate Exception handler.
29 // i.e. matching catch block.
30 public static void main(String args[])
31 {
32     int a = 1;
33     int b = 0;
34     try
35     {
36         int i = computeDivision(a,b);
37     }
38     // matching ArithmeticException
39     catch(ArithmeticException ex)
40     {
41         // getMessage will print description of exception(here / by zero)
42         System.out.println(ex.getMessage());
43     }
44 }
45 }
```

# Java catch multiple exceptions:

### Java Multi-catch block

✓ A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

### Points to remember

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for `ArithmaticException` must come before catch for `Exception`.

## Exception Handling XLVI

### Example 1: java multi-catch block.

```
1 public class MultipleCatchBlock1
2 {
3     public static void main(String[] args)
4     {
5         try
6         {
7             int a[]={new int[5];
8             a[5]=30/0;
9         }
10        catch(ArithmetricException e)
11        {
12            System.out.println("Arithmetric Exception occurs");
13        }
14        catch(ArrayIndexOutOfBoundsException e)
15        {
16            System.out.println("ArrayIndexOutOfBoundsException occurs");
17        }
18        catch(Exception e)
19        {
20            System.out.println("Parent Exception occurs");
21        }
22        System.out.println("rest of the code");
23    }
24}
```

## Exception Handling XLVII

### Example 2: java multi-catch block.

```
1 public class MultipleCatchBlock2
2 {
3     public static void main(String[] args)
4     {
5         try
6         {
7             int a[]={};
8             System.out.println(a[10]);
9         }
10        catch(ArithmetcException e)
11        {
12            System.out.println("Arithmetc Exception occurs");
13        }
14        catch(ArrayIndexOutOfBoundsException e)
15        {
16            System.out.println("ArrayIndexOutOfBoundsException occurs");
17        }
18        catch(Exception e)
19        {
20            System.out.println("Parent Exception occurs");
21        }
22        System.out.println("rest of the code");
23    }
24}
```

## Exception Handling XLVIII

**Example 3: try block contains two exceptions. But at a time only one exception occurs and its corresponding catch block is invoked.**

```
1 public class MultipleCatchBlock3
2 {
3     public static void main(String[] args)
4     {
5         try
6         {
7             int a[]={new int[5];
8             a[5]=30/0;
9             System.out.println(a[10]);
10        }
11        catch(ArithmeticException e)
12        {
13            System.out.println("Arithmetric Exception occurs");
14        }
15        catch(ArrayIndexOutOfBoundsException e)
16        {
17            System.out.println("ArrayIndexOutOfBoundsException occurs");
18        }
19        catch(Exception e)
20        {
21            System.out.println("Parent Exception occurs");
22        }
23    }
24 }
```

## Exception Handling XLIX

```
23     System.out.println("rest of the code");
24 }
25 }
```

## Exception Handling L

**Example 4: generate NullPointerException, but didn't provide the corresponding exception type. In such case, the catch block containing the parent exception class Exception will invoked.**

```
1 public class MultipleCatchBlock4
2 {
3     public static void main(String[] args)
4     {
5         try
6         {
7             String s=null;
8             System.out.println(s.length());
9         }
10        catch(ArithmetricException e)
11        {
12            System.out.println("Arithmetric Exception occurs");
13        }
14        catch(ArrayIndexOutOfBoundsException e)
15        {
16            System.out.println("ArrayIndexOutOfBoundsException occurs");
17        }
18        catch(Exception e)
19        {
20            System.out.println("Parent Exception occurs");
```

## Exception Handling LI

```
21      }
22      System.out.println("rest of the code");
23  }
24 }
```

## Exception Handling LII

**Example 5: to handle the exception without maintaining the order of exceptions (i.e. from most specific to most general).**

```
1 class MultipleCatchBlock5
2 {
3     public static void main(String args[])
4     {
5         try
6         {
7             int a[]={new int[5];
8             a[5]=30/0;
9         }
10        catch(Exception e)
11        {
12            System.out.println("common task completed");
13        }
14        catch(ArithmaticException e)
15        {
16            System.out.println("task1 is completed");
17        }
18        catch(ArrayIndexOutOfBoundsException e)
19        {
20            System.out.println("task 2 completed");
21        }
22        System.out.println("rest of the code...");
```

## Exception Handling LIII

23      }  
24     }

### Java Nested try block

- ✓ The try block within a try block is known as nested try block in java.

#### Why use nested try block

- ✓ Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax:

# Exception Handling LV

```
1 ....  
2 try  
3 {  
4     statement 1;  
5     statement 2;  
6     try  
7     {  
8         statement 1;  
9         statement 2;  
10    }  
11    catch(Exception e)  
12    {  
13    }  
14}  
15 catch(Exception e)  
16 {  
17}  
18 ....
```

# Exception Handling LVI

## Java nested try example:

```
1 class Excep6
2 {
3     public static void main(String args[])
4     {
5         try
6         {
7             try
8             {
9                 System.out.println("going to divide");
10                int b =39/0;
11            }
12            catch(ArithmeticException e)
13            {
14                System.out.println(e);
15            }
16
17            try
18            {
19                int a[]=new int[5];
20                a[5]=4;
21            }
22            catch(ArrayIndexOutOfBoundsException e)
23            {
24                System.out.println(e);
25            }
26        }
27    }
28}
```

## Exception Handling LVII

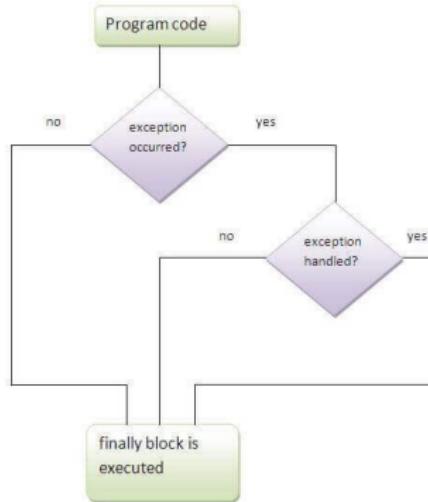
```
25        }
26
27        System.out.println("other statement");
28    }
29    catch(Exception e)
30    {
31        System.out.println("handled");
32    }
33
34    System.out.println("normal flow..");
35
36 }
```

### Java finally block

- ✓ Java finally block is a block that is used to execute important code such as closing connection, stream etc.
- ✓ Java finally block is always executed whether exception is handled or not.
- ✓ Java finally block follows try or catch block.
- ✓ The finally block always executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs. But finally is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break.

## Exception Handling LIX

- ✓ Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.



## Exception Handling LX

Note: If you don't handle exception, before terminating the program, JVM executes finally block(if any).

# Exception Handling LXI

## Why use java finally

- ✓ Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

**Example to see the different cases where java finally block can be used.**

**Case 1: the java finally example where exception doesn't occur.**

```
1 class TestFinallyBlock
2 {
3     public static void main(String args[])
4     {
5         try
6         {
7             int data=25/5;
8             System.out.println(data);
9         }
10        catch(NullPointerException e)
11        {
12            System.out.println(e);
13        }
14        finally
```

## Exception Handling LXII

```
15      {
16          System.out.println("finally block is always executed");
17      }
18      System.out.println("rest of the code...");
19  }
20 }
```

### Case 2: the java finally example where exception occurs and not handled.

```
1 class TestFinallyBlock1
2 {
3     public static void main(String args[])
4     {
5         try
6         {
7             int data=25/0;
8             System.out.println(data);
9         }
10        catch(NullPointerException e)
11        {
12            System.out.println(e);
13        }
14        finally
15        {
```

## Exception Handling LXIII

```
16         System.out.println("finally block is always executed");
17     }
18     System.out.println("rest of the code...");
19 }
20 }
```

### Case 3: the java finally example where exception occurs and handled.

```
1 public class TestFinallyBlock2
2 {
3     public static void main(String args[])
4     {
5         try
6         {
7             int data=25/0;
8             System.out.println(data);
9         }
10        catch(ArithmetricException e)
11        {
12            System.out.println(e);
13        }
14        finally
15        {
16            System.out.println("finally block is always executed");
17        }
18    }
19 }
```

## Exception Handling LXIV

```
17     }
18     System.out.println("rest of the code...");
19 }
20 }
```

**Rule:** For each try block there can be zero or more catch blocks, but only one finally block.

**Note:** The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

### Java throw exception

#### Java throw keyword

- ✓ The Java throw keyword is used to explicitly throw an exception.
- ✓ We can throw either checked or unchecked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception.
- ✓ The syntax of java throw keyword is given below.

```
1     throw exception;
```

```
2  
3 Example: throw new IOException("sorry device error");
```

#### java throw keyword example:

- ✓ In this example, created the validate method that takes integer value as a parameter. If the age is less than 18, it is throwing the ArithmeticException otherwise print a message welcome to vote.

# Exception Handling LXVI

```
1 public class TestThrow1
2 {
3     static void validate(int age)
4     {
5         if(age<18)
6             throw new ArithmeticException("not valid");
7         else
8             System.out.println("welcome to vote");
9     }
10    public static void main(String args[])
11    {
12        validate(13);
13        System.out.println("rest of the code...");
14    }
15 }
```

# Java Exception propagation

- ✓ An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method, If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

Rule: By default Unchecked Exceptions are forwarded in calling chain (propagated).

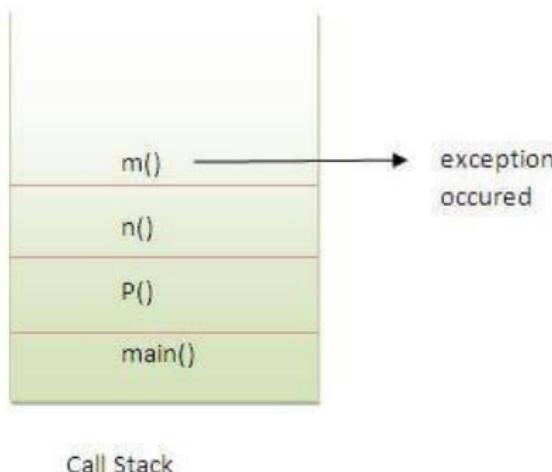
# Exception Handling LXVIII

## Program of Exception Propagation:

```
1 class TestExceptionPropagation1
2 {
3     void m()
4     {
5         int data=50/0;
6     }
7     void n()
8     {
9         m();
10    }
11    void p()
12    {
13        try
14        {
15            n();
16        }
17        catch(Exception e)
18        {
19            System.out.println("exception handled");
20        }
21    }
22    public static void main(String args[])
23    {
24        TestExceptionPropagation1 obj=new TestExceptionPropagation1();
```

## Exception Handling LXIX

```
25      obj.p();
26      System.out.println("normal flow...");
27  }
28 }
```



## Exception Handling LXX

- ✓ In the above example exception occurs in m() method where it is not handled, so it is propagated to previous n() method where it is not handled, again it is propagated to p() method where exception is handled.
- ✓ Exception can be handled in any method in call stack either in main() method, p() method, n() method or m() method.

Rule: By default, Checked Exceptions are not forwarded in calling chain (propagated).

## Exception Handling LXXI

Program which describes that checked exceptions are not propagated:

```
1 class TestExceptionPropagation2
2 {
3     void m()
4     {
5         throw new java.io.IOException("device error"); //checked exception
6     }
7     void n()
8     {
9         m();
10    }
11    void p()
12    {
13        try
14        {
15            n();
16        }
17        catch(Exception e)
18        {
19            System.out.println("exception handled");
20        }
21    }
22    public static void main(String args[])
23    {
24        TestExceptionPropagation2 obj=new TestExceptionPropagation2();
```

## Exception Handling LXXII

```
25         obj.p();
26         System.out.println("normal flow");
27     }
28 }
```

### Java throws keyword

- ✓ The Java throws keyword is used to declare an exception.
- ✓ It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.
- ✓ Exception Handling is mainly used to handle the checked exceptions.
- ✓ If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

## Exception Handling LXXIII

Syntax of java throws:

```
1 return_type method_name() throws exception_class_name  
2 {  
3     //method code  
4 }
```

### Which exception should be declared

Ans) checked exception only, because:

- unchecked Exception: under your control so correct your code.
- error: beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

### Advantage of Java throws keyword

- ✓ Now Checked Exception can be propagated (forwarded in call stack).
- ✓ It provides information to the caller of the method about the exception.

### Java throws example:

Example 1: throws clause which describes that checked exceptions can be propagated by throws keyword.

## Exception Handling LXXV

```
1 import java.io.IOException;
2 class Testthrows1
3 {
4     void m() throws IOException
5     {
6         throw new IOException("device error"); //checked exception
7     }
8     void n() throws IOException
9     {
10        m();
11    }
12    void p()
13    {
14        try
15        {
16            n();
17        }
18        catch(Exception e)
19        {
20            System.out.println("exception handled");
21        }
22    }
23    public static void main(String args[])
24    {
25        Testthrows1 obj=new Testthrows1();
```

## Exception Handling LXXVI

```
26     obj.p();
27     System.out.println("normal flow...");
28 }
29 }
```

Rule: If you are calling a method that declares an exception, you must either catch or declare the exception.

There are two cases:

- Case1: You caught the exception i.e. handle the exception using try/catch.
- Case2: You declare the exception i.e. specifying throws with the method.

## Exception Handling LXXVII

Case1: You handle the exception

- ✓ In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

```
1 import java.io.*;
2 class M
3 {
4     void method()throws IOException
5     {
6         throw new IOException("device error");
7     }
8 }
9 public class Testthrows2
10 {
11     public static void main(String args[])
12     {
13         try
14         {
15             M m=new M();
16             m.method();
17         }
18         catch(Exception e)
19         {
20             System.out.println("exception handled");
```

## Exception Handling LXXVIII

```
21      }
22
23     System.out.println("normal flow...");
24   }
25 }
```

Case2: You declare the exception:

- A.** In case you declare the exception, if exception does not occur, the code will be executed fine.
- B.** In case you declare the exception if exception occurs, an exception will be thrown at run-time because throws does not handle the exception.

Example A: Program if exception does not occur

## Exception Handling LXXIX

```
1 import java.io.*;
2 class M
3 {
4     void method()throws IOException
5     {
6         System.out.println("device operation performed");
7     }
8 }
9 class Testthrows3
10 {
11     public static void main(String args[])throws IOException
12     {
13         //declare exception
14         M m=new M();
15         m.method();
16
17         System.out.println("normal flow...");
18     }
19 }
```

## Exception Handling LXXX

### Example B: Program if exception occurs

```
1 import java.io.*;
2 class M
3 {
4     void method()throws IOException
5     {
6         throw new IOException("device error");
7     }
8 }
9 class Testthrows4
{
10     public static void main(String args[])throws IOException
11     {
12         //declare exception
13         M m=new M();
14         m.method();
15
16         System.out.println("normal flow...");
17     }
18 }
19 }
```

## Exception Handling LXXXI

### Difference between throw and throws in Java

No.	throw	throws
1	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3	Throw is followed by an instance.	Throws is followed by class.
4	Throw is used within the method.	Throws is used with the method signature.
5	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method() throws IOException, SQLException.

## Exception Handling LXXXII

### Java throw example

```
1 void m()
2 {
3     throw new ArithmeticException("sorry");
4 }
```

### Java throws example

```
1 void m()throws ArithmeticException
2 {
3     //method code
4 }
```

### Java throw and throws example

```
1 void m()throws ArithmeticException
2 {
3     throw new ArithmeticException("sorry");
4 }
```

## Exception Handling LXXXIII

### Difference between final, finally and finalize

✓ There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

## Exception Handling LXXXIV

### Java final example:

```
1 class FinalExample
2 {
3     public static void main(String[] args)
4     {
5         final int x=100;
6         x=200; //Compile Time Error
7     }
8 }
9 }
```

### Java finally example:

```
1 class FinallyExample
2 {
3     public static void main(String[] args)
4     {
5         try
6         {
7             int x=300;
8         }
9         catch(Exception e)
10        {
11             System.out.println(e);
12         }
13 }
```

## Exception Handling LXXXV

```
13         finally
14     {
15         System.out.println("finally block is executed");
16     }
17 }
18
19 }
```

### Java finalize example:

```
1 class FinalizeExample
2 {
3     public void finalize()
4     {
5         System.out.println("finalize called");
6     }
7     public static void main(String[] args)
8     {
9         FinalizeExample f1=new FinalizeExample();
10        FinalizeExample f2=new FinalizeExample();
11        f1=null;
12        f2=null;
13        System.gc();
14    }
15 }
```

## ExceptionHandling with MethodOverriding in Java I

There are many rules if we talk about methodoverriding with exception handling. The Rules are as follows:

- If the superclass method does not declare an exception
  - If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.
- If the superclass method declares an exception
  - If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

## ExceptionHandling with MethodOverriding in Java II

If the superclass method does not declare an exception:

1) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception.

```
1 import java.io.*;
2 class Parent
3 {
4     void msg()
5     {
6         System.out.println("parent");
7     }
8 }
9 class TestExceptionChild extends Parent
10 {
11     void msg()throws IOException
12     {
13         System.out.println("TestExceptionChild");
14     }
15     public static void main(String args[])
16     {
17         Parent p=new TestExceptionChild();
18         p.msg();
```

## Exception Handling with Method Overriding in Java III

```
19     }
20 }
21 Output:Compile Time Error
```

- 2) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.

```
1 import java.io.*;
2 class Parent
3 {
4     void msg()
5     {
6         System.out.println("parent");
7     }
8 }
9 class TestExceptionChild1 extends Parent
10 {
11     void msg() throws ArithmeticException
12     {
13         System.out.println("child");
```

## ExceptionHandling with MethodOverriding in Java IV

```
14
15     }
16
17     public static void main(String args[])
18     {
19         Parent p=new TestExceptionChild1();
20         p.msg();
21     }
21 Output:child
```

- 3) Rule: If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

## ExceptionHandling with MethodOverriding in Java V

**Example: in case subclass overridden method declares parent exception**

```
1 import java.io.*;
2 class Parent
3 {
4     void msg()throws ArithmeticException
5     {
6         System.out.println("parent");
7     }
8 }
9
10 class TestExceptionChild2 extends Parent
11 {
12     void msg()throws Exception
13     {
14         System.out.println("child");
15     }
16
17     public static void main(String args[])
18     {
19         Parent p=new TestExceptionChild2();
20         try
21         {
22             p.msg();
```

## ExceptionHandling with MethodOverriding in Java VI

```
23 }  
24 catch(Exception e)  
25 {  
26  
27 }  
28 }  
29 }  
30 Output:Compile Time Error
```

## ExceptionHandling with MethodOverriding in Java VII

### Example: in case subclass overridden method declares same exception

```
1 import java.io.*;
2 class Parent
3 {
4     void msg()throws Exception
5     {
6         System.out.println("parent");
7     }
8 }
9 class TestExceptionChild3 extends Parent
10 {
11     void msg()throws Exception
12     {
13         System.out.println("child");
14     }
15
16     public static void main(String args[])
17     {
18         Parent p=new TestExceptionChild3();
19         try
20         {
21             p.msg();
22         }
```

## ExceptionHandling with MethodOverriding in Java VIII

```
23         catch(Exception e)
24     {
25
26     }
27 }
28 Output:child
29
```

## ExceptionHandling with MethodOverriding in Java IX

**Example: in case subclass overridden method declares subclass exception**

```
1 import java.io.*;
2 class Parent
3 {
4     void msg()throws Exception
5     {
6         System.out.println("parent");
7     }
8 }
9 class TestExceptionChild4 extends Parent
10 {
11     void msg()throws ArithmeticException
12     {
13         System.out.println("child");
14     }
15
16     public static void main(String args[])
17     {
18         Parent p=new TestExceptionChild4();
19         try
20         {
21             p.msg();
22         }
```

## ExceptionHandling with MethodOverriding in Java X

```
23         catch(Exception e)
24     {
25
26     }
27 }
28 Output:child
29
```

## ExceptionHandling with MethodOverriding in Java XI

### Example: in case subclass overridden method declares no exception

```
1 import java.io.*;
2 class Parent
3 {
4     void msg()throws Exception
5     {
6         System.out.println("parent");
7     }
8 }
9 class TestExceptionChild5 extends Parent
{
10    void msg()
11    {
12        System.out.println("child");
13    }
14
15
16    public static void main(String args[])
17    {
18        Parent p=new TestExceptionChild5();
19        try
20        {
21            p.msg();
22        }
23        catch(Exception e)
24        {
```

## ExceptionHandling with MethodOverriding in Java XII

```
25  
26     }  
27 }  
28 }  
29 Output:child
```

### Java Custom Exception

- ✓ If you are creating your own Exception that is known as custom exception or user-defined exception.
- ✓ Java custom exceptions are used to customize the exception according to user need.
- ✓ By the help of custom exception, you can have your own exception and message.

## ExceptionHandling with MethodOverriding in Java XIII

Let's see a simple example of java custom exception.

```
1 class InvalidAgeException extends Exception
2 {
3     InvalidAgeException(String s)
4     {
5         super(s);
6     }
7 }
8
9 class TestCustomException1
10 {
11
12     static void validate(int age) throws InvalidAgeException
13     {
14         if(age<18)
15             throw new InvalidAgeException("not valid");
16         else
17             System.out.println("welcome to vote");
18     }
19
20     public static void main(String args[])
21     {
22         try
23         {
24             validate(13);
```

## ExceptionHandling with MethodOverriding in Java XIV

```
17 }  
18 catch(Exception m)  
19 {  
20     System.out.println("Exception occured: "+m);  
21 }  
22  
23 System.out.println("rest of the code...");  
24 }  
25 }  
26 Output:Exception occured: InvalidAgeException:not valid  
27 rest of the code...
```

## Java Exception Handling Quiz

<https://www.javatpoint.com/directload.jsp?val=89>

<https://www.javatpoint.com/directload.jsp?val=96>

<https://www.javatpoint.com/directload.jsp?val=101>