

A large, light blue, stylized watermark of the Java logo is centered in the background. It consists of three concentric, slightly offset circular arcs and a small solid circle in the center, creating a sense of depth and movement.

Core Java Training

Course Agenda

Day 1:

Introduction, Environment, OOP, Conditional Constructs, Loops, Arrays, Classes

Day 2:

Classes, Objects, Constructors, Methods, Passing Arguments, Packages

Day 3:

Inheritance , Polymorphism

Day 4:

Abstraction, Interface

Day 5:

Exception Handling

Day 6:

Multithreading

Day 7:

Collection Framework

Day 8:

Generics, File I/O

Day 9:

JDBC

Day 1

Introduction

- History of Java
- Java characteristics
- Java and OOP
- Classes, Objects, Encapsulation, Inheritance, Polymorphism
- Environment: JRE, JVM, Classpath

Constructs, Loops

- Basic syntax, comments, literals, identifiers, keywords
- Simple and complex types, variables, constants, casting

Arrays

- Single Dimensional
- Two Dimensional

Day 2

Classes, Objects Constructors

- Classes, Object creation, constructors
- Data Members and Methods
- Visibility Modes
- Static data members, Static Methods

Packages

- Packages
- String, String Buffer, String Builder, Math, Random, System
- Wrapper classes

Day 3

Inheritance

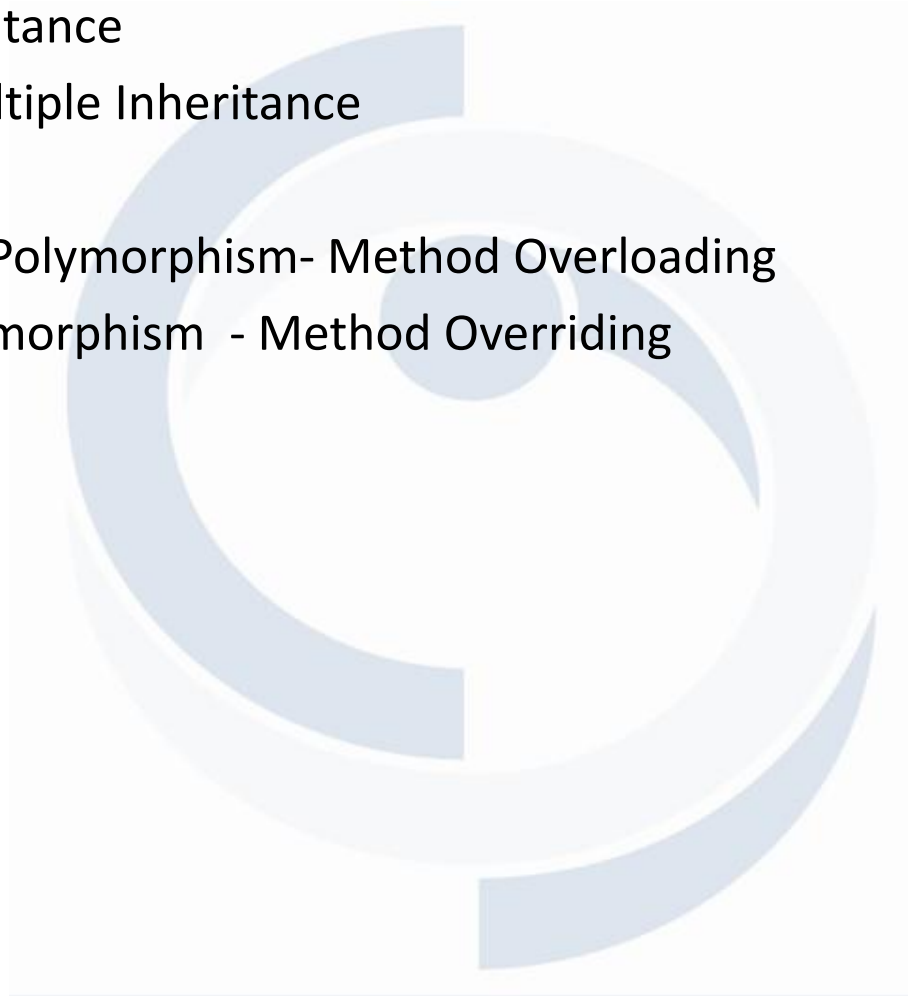
- Types of Inheritance
- Single and Multiple Inheritance

Polymorphism

- Compile time Polymorphism- Method Overloading
- Run time Polymorphism - Method Overriding

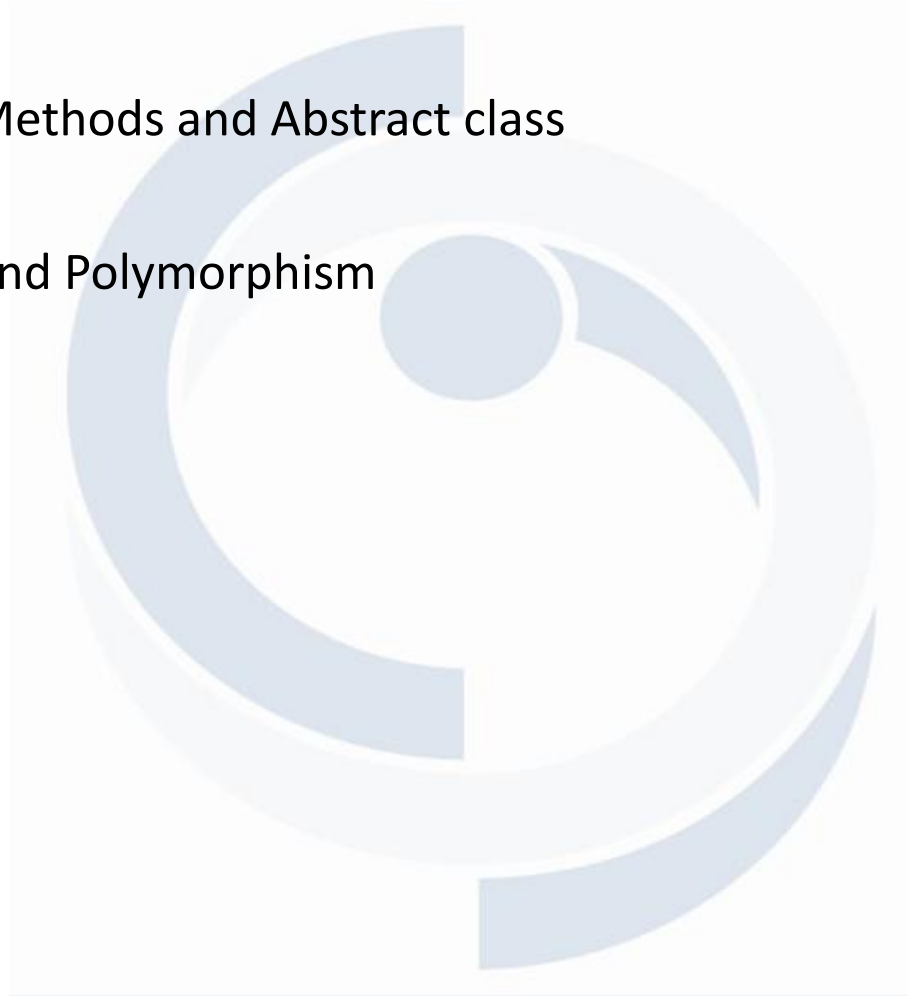
Final keyword

- Final Class
- Final methods



Day 4

- **Abstraction**
Abstract Methods and Abstract class
- **Interface**
Interface and Polymorphism



Day 5

Exception Handling

- Try catch finally

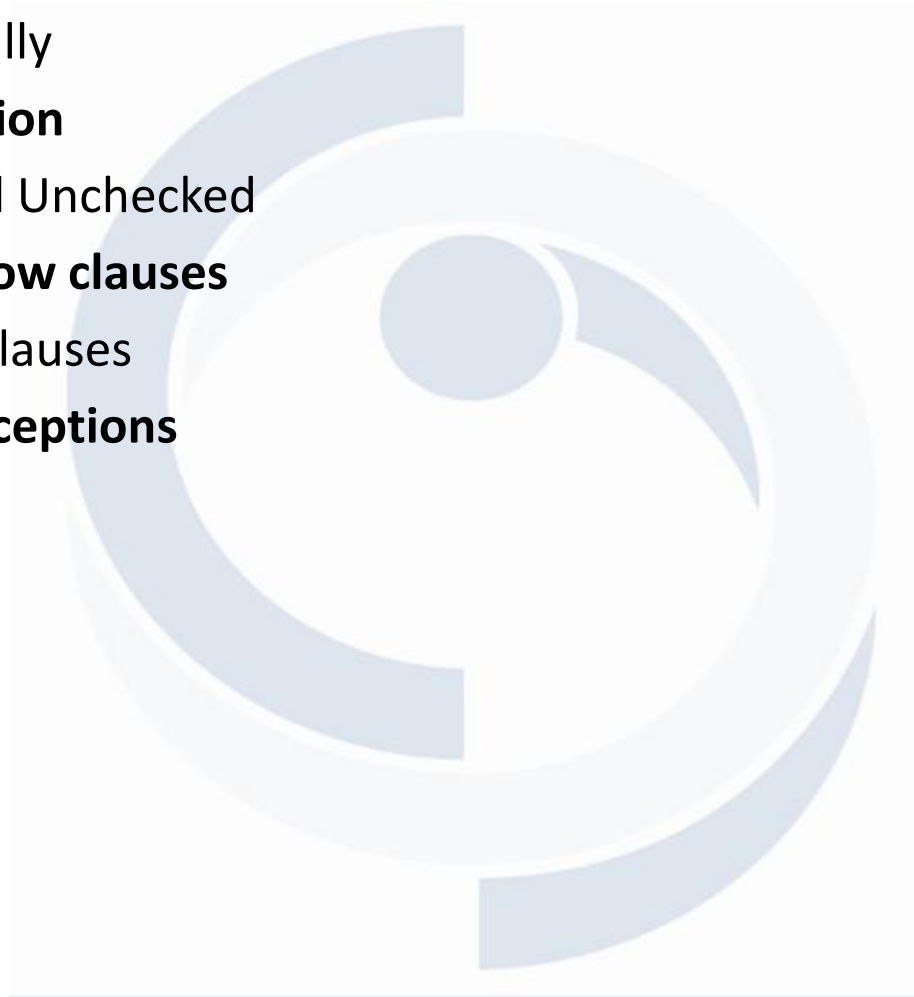
Types Of Exception

- Checked and Unchecked

Throws and Throw clauses

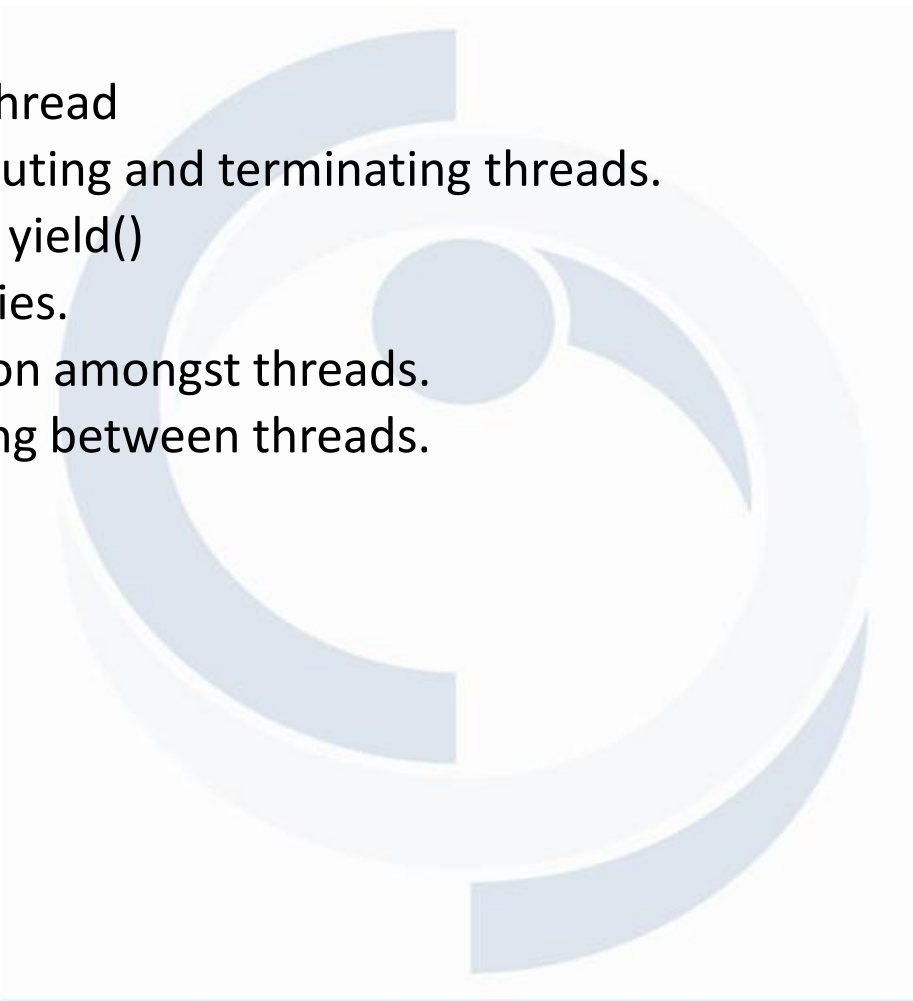
- Use of the clauses

User defined Exceptions



Day 6

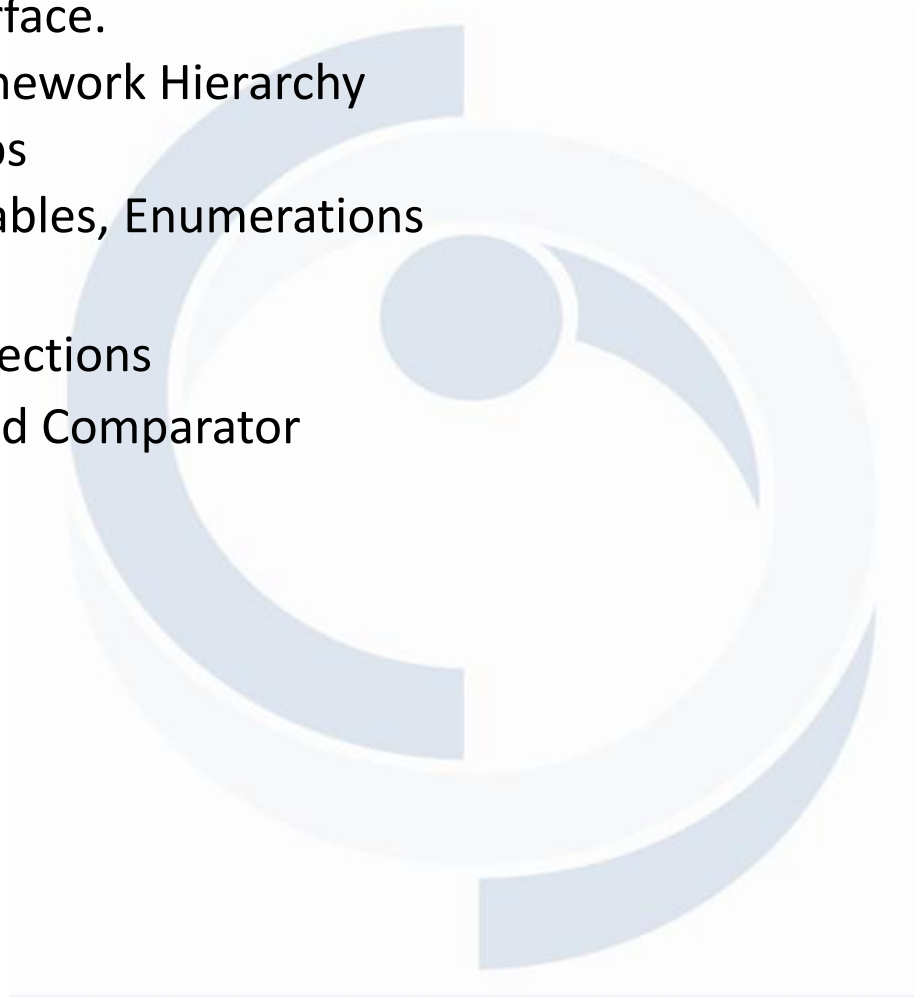
Multithreading

- Introduction.
 - Life Cycle of Thread
 - Creating, executing and terminating threads.
 - sleep(), join(), yield()
 - Thread priorities.
 - Synchronization amongst threads.
 - Communicating between threads.
- 

Day 7

Collection API

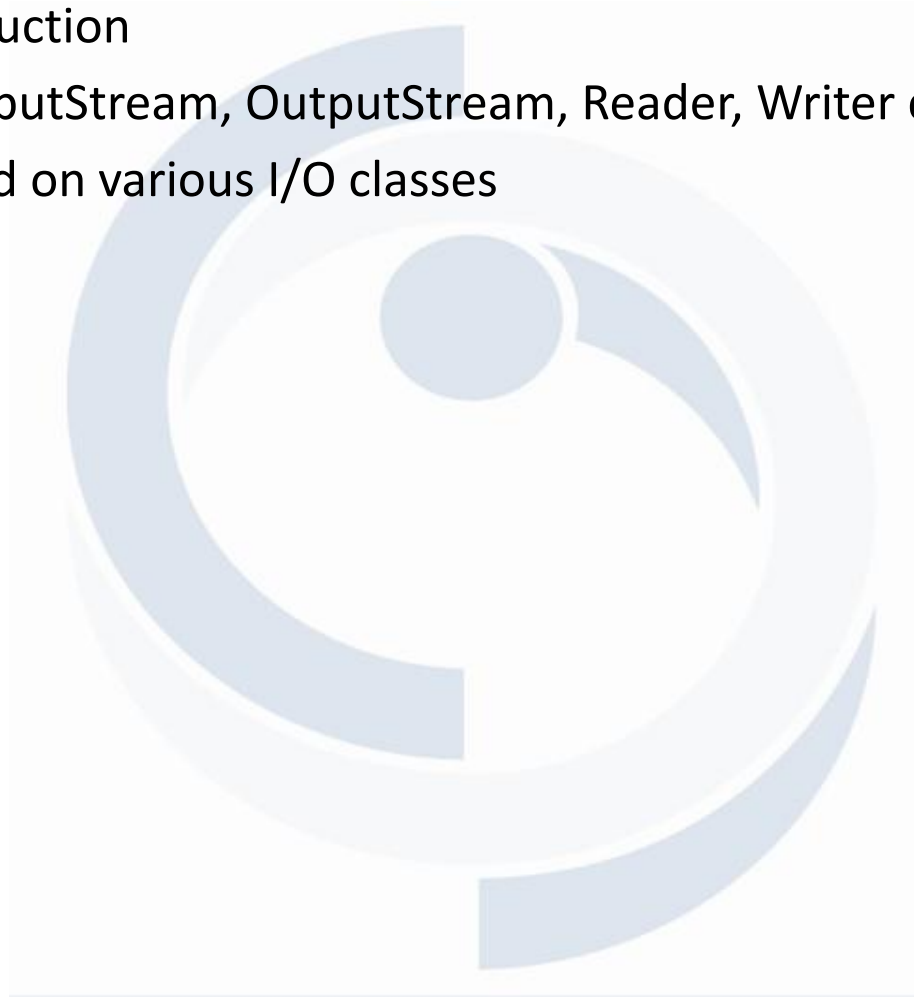
- Collection interface.
- Collection Framework Hierarchy
- Lists, Sets, Maps
- Vectors, Hashtables, Enumerations
- Iterators
- Arrays and Collections
- Comparable and Comparator



Day 8

File Handling - I/O

- Streams Introduction
- Serialization InputStream, OutputStream, Reader, Writer class
- Examples based on various I/O classes



Day 9

JDBC

- Introduction
- JDBC Drivers
- JDBC API



History of Java

- Sun Microsystems in 1991 commissioned “Green Project”
- Led by James Gosling
- Developed language ‘Oak’ -Later renamed to Java
- Along comes the Internet, and the need for a language that would work on all types of devices, “ a language without boundaries” is felt
- 1995, java emerges into limelight, OOPs the requirement of the situation
- Became popular with the rising popularity of ‘www’

Java characteristics

Simple:

- Syntax is based on the familiar 'C++' language
- Complex features like pointers were avoided making the Java programming simple

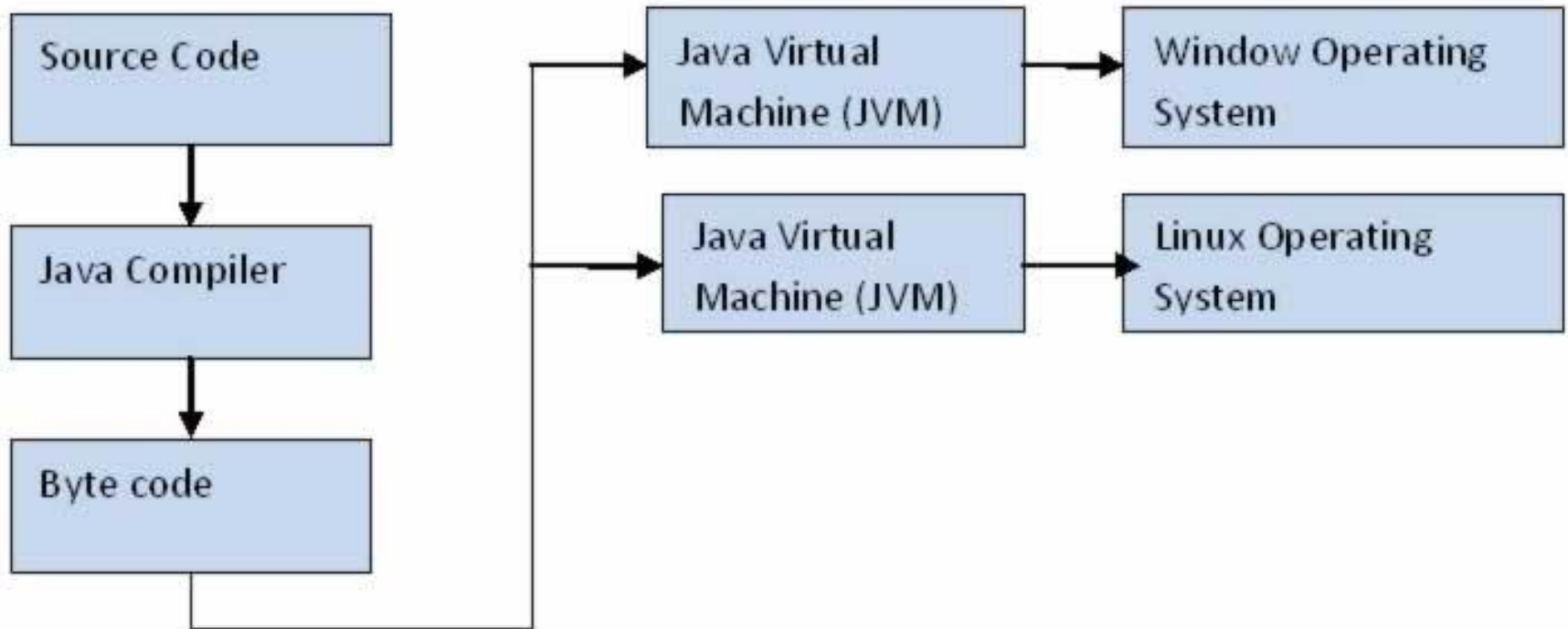
Object oriented:

- Programming is object oriented, comparable to C++
- Object oriented design is a technique that focuses design on data (i.e. objects) and on the interfaces to it

Architectural Neutral :

- The Java compiler generates an intermediate byte code which does not depend on the architecture of a computer i.e., whether it is an IBM PC, a Macintosh, or a Solaris system
- Java programs can be used on any machine irrespective of its architecture and hence the Java program is Architectural Neutral

Architecture Neutral



Java characteristics

Portable :

- Size of data types are always same irrespective of the system architecture. That is an int in Java is always 32 bit unlike in C/C++ where the size may be 16-bit or 32-bit depending on the compiler and machine
- Hence when ported on different machines, there does not occur any change in the values the data types can hold
- Java program run on any platform UNIX, Windows and the Macintosh systems

Distributed:

- Java applications are capable of accessing objects across the net, via URLs as easy as a local file system.
- We can create distributed applications in java.
- RMI and EJB are used for creating distributed applications.
- We may access files/object by calling the methods from any machine on the internet

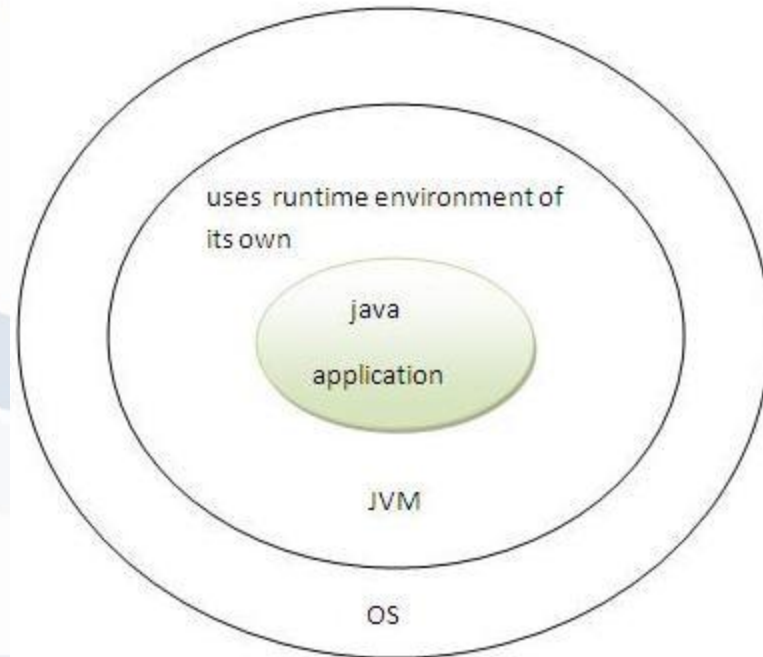
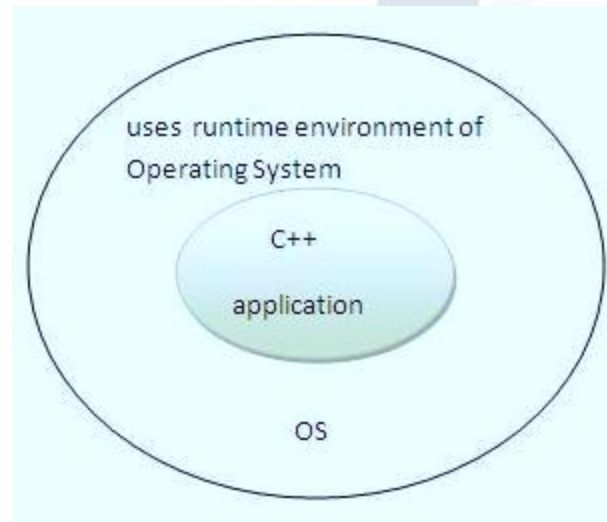
Java characteristics

Secure:

No explicit pointer Programs .

Runs inside virtual machine sandbox.

Byte code Verifier- checks the code fragments for illegal code that can violate access right to objects.



Java characteristics

Multithreaded:

- Multithreading is the ability for one program to do more than one task at once. Compared to other languages it is easy to implement multithreading in Java

Interpreted:

- The Java Interpreter can execute Java bytecode, directly on any machine to which the interpreter has been ported. Interpreted code is slower than compiled code

Java and OOPs

OOPS

- OOPs deals with the basic parts or **building blocks** of the problem
- Models the real world by **decomposing the problem into smaller discrete** pieces called objects
- The objects data becomes its **state** and its methods become its **behavior**
- Thus oops uses Objects, now having state and behavior to emulate real world objects. Programming using such objects is called OOPs
- Just as there is interaction between objects in real life, OOPs program interacts with the objects defined in the program

Java Architecture

Components of Java Architecture: JDK, JRE, JVM, JIT, Java API

Programming steps:

- Development: Writing a program in programming language (JDK)
- Compilation: compiling → converting programming language statements into machine language (binary) → debugging (JDK)
- Execution: Running the program, getting desired output (JRE)

JDK

- Java Development Kit (JDK), a set of programming tools for developing [Java](#) applications
- Facility to write and debug code

JRE

- The Java Runtime Environment (JRE), is part of the JDK.
- Provides the minimum requirements for executing a Java application; it consists of the Java Virtual Machine (JVM), core classes, and supporting files
- Can work independent of JDK

Java Architecture

JVM:

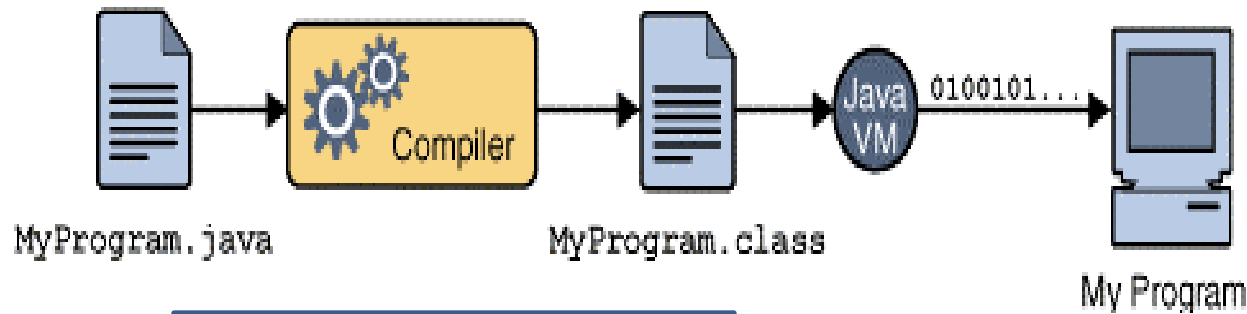
- A Java virtual machine is a virtual machine that can execute Java bytecode. It is the code execution component of the Java platform.

Java API:

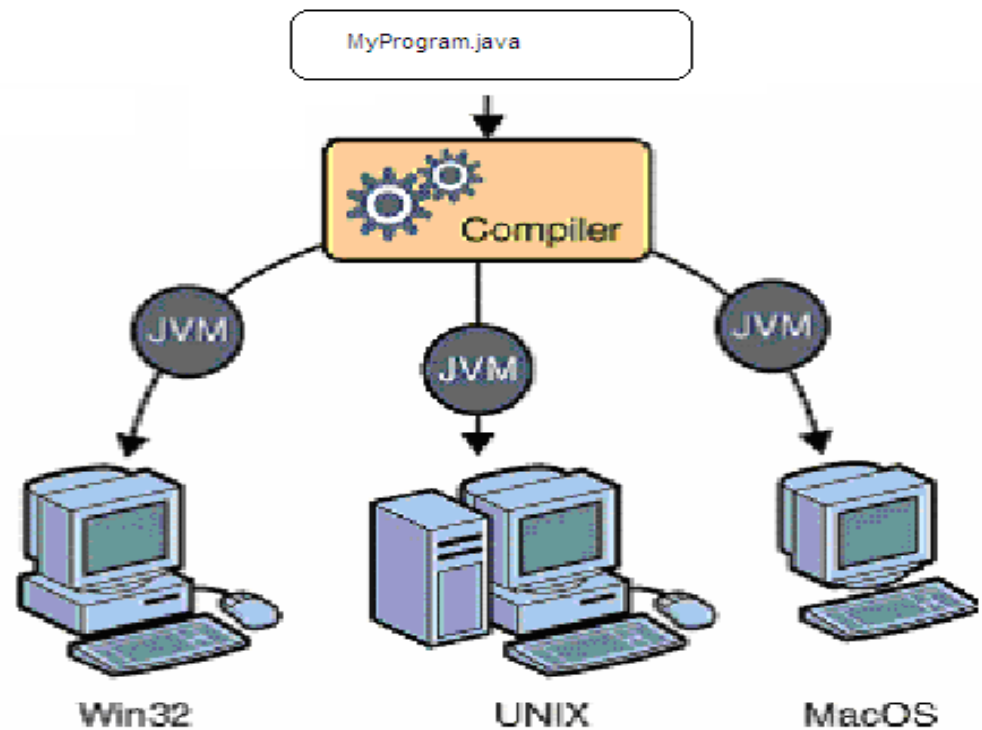
- An application programming interface (API) is a library of functions that Java provides for programmers for common tasks like file transfer, networking, and data structures.
- Java.io → System.out.print
- Java.lang → length method for arrays; exceptions

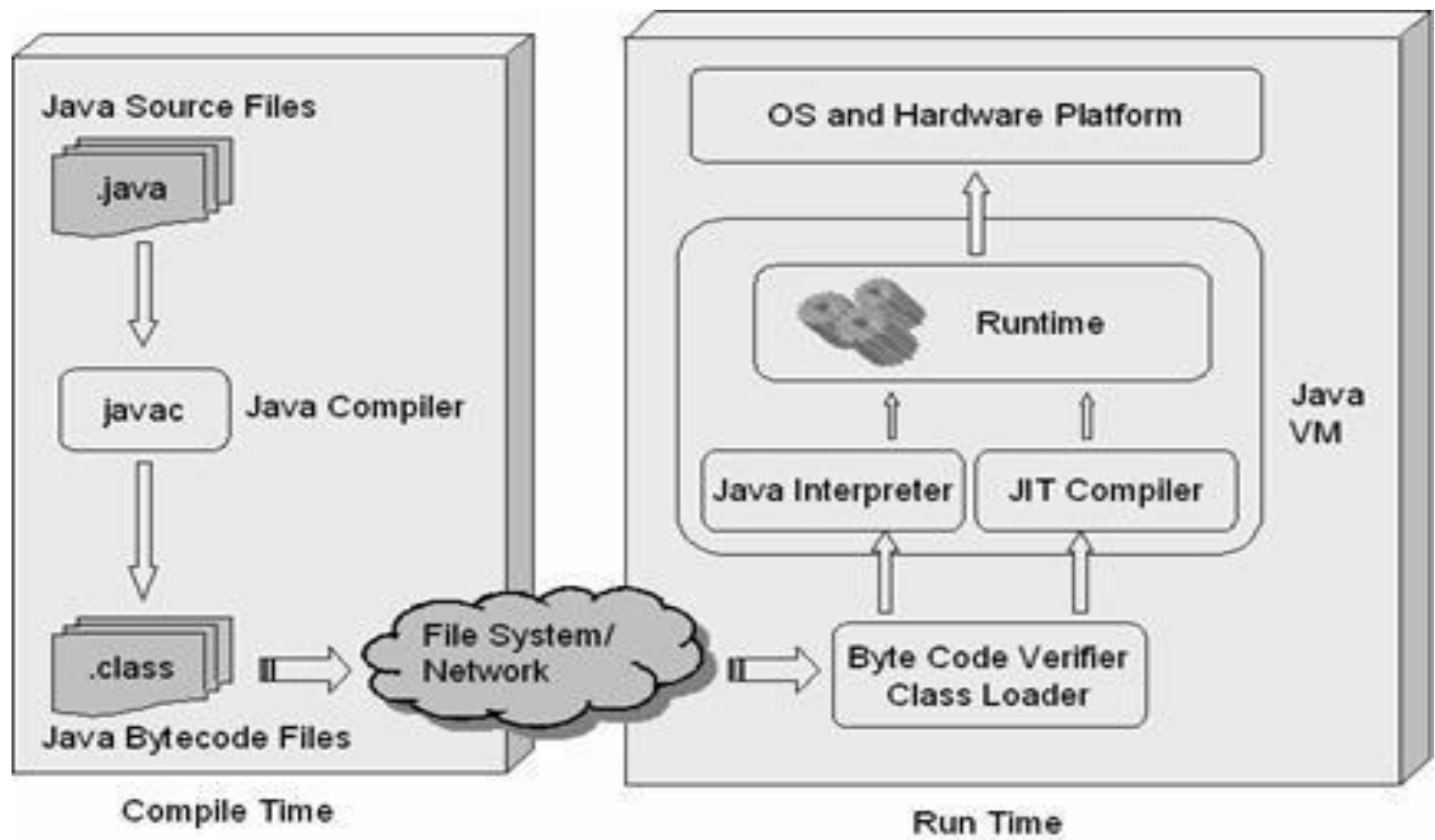


Java Architecture



How a program runs





Types of Java programs

Applet

- An applet runs within a web page on your browser

Application

- Application is a "full blown" Java application that runs on the desktop
- Locally installed VM is executed that loads/runs locally available Java classes

Path and Classpath

Path

- specifies a set of directories where executable programs are located.

Classpath

- An environment variable that tells the JVM where to look for user-defined classes and packages in Java programs
- It is the connection between the Java runtime and the filesystem
- It defines where the compiler and interpreter look for .class files to load

First Java Program: Hello World

```
public class HelloWorld {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        System.out.println("Hello World");  
    }  
}
```

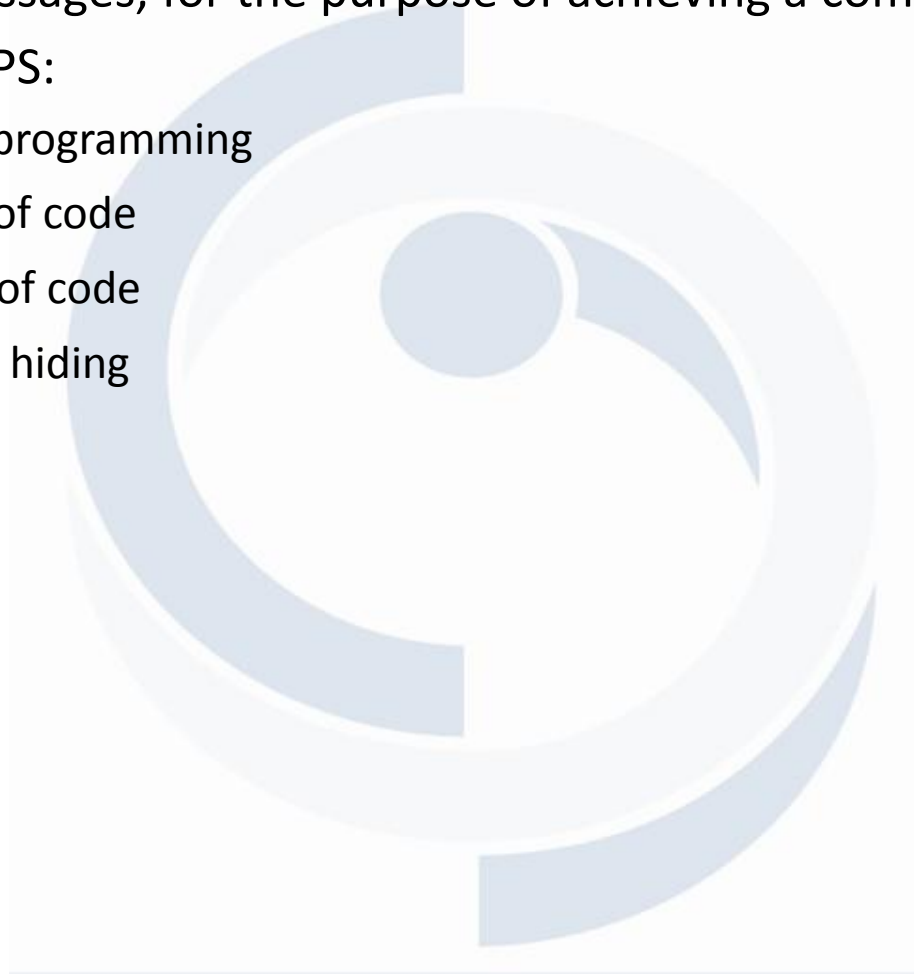
Set path, classpath

```
javac HelloWorld.java
```

```
java HelloWorld
```

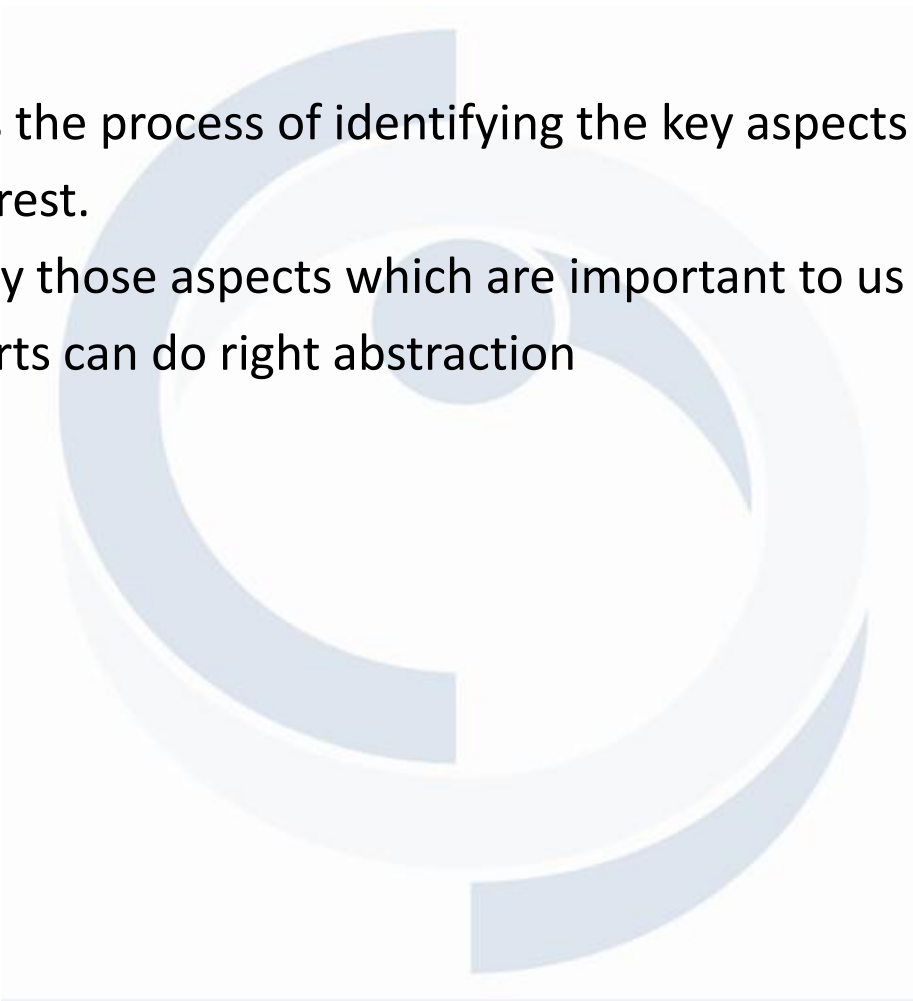
Object Model

- An Object-Oriented Program consists of a group of cooperating objects, exchanging messages, for the purpose of achieving a common objective.
- Benefits of OOPS:
 - Real-world programming
 - Reusability of code
 - Modularity of code
 - information hiding



Four Pillars of OOPs

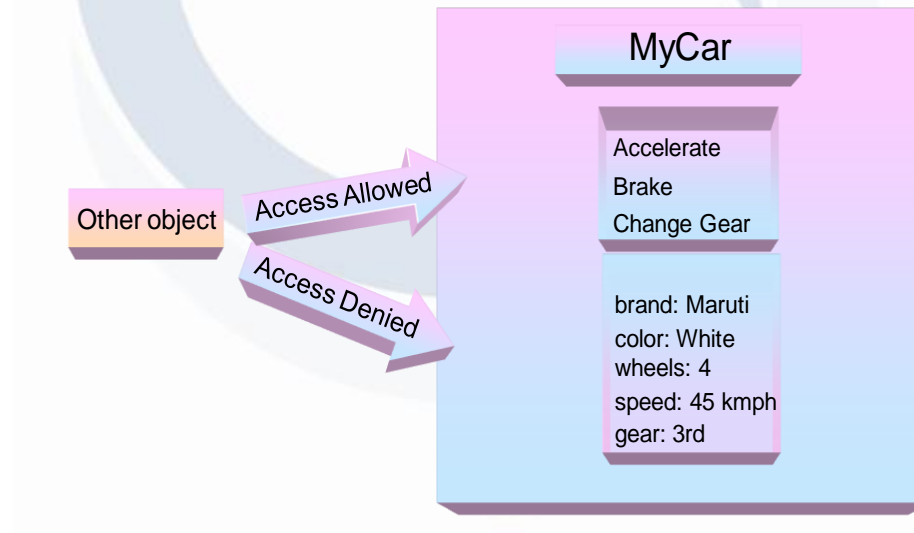
1. Abstraction

- Abstraction is the process of identifying the key aspects of an entity and ignoring the rest.
 - We select only those aspects which are important to us
 - Domain Experts can do right abstraction
- 

Four Pillars of OOPs

2. Encapsulation

- Encapsulation serves to separate interface of an abstraction and its implementation
- Encapsulation ensures that data within an object is protected; it can be accessed only by its methods
- Encapsulation is the ability of an object to place a boundary around its properties (i.e. data) and methods (i.e. operations)



Four Pillars of OOPs

3. Inheritance

- Classification helps in **handling complexity**.
- Inheritance is a property of a class hierarchy whereby each subclass inherits attributes and methods of its super-class
- **Superclass** and **Subclass**
- The **subclass** can have **additional** specific **attributes** and **methods**.
- Enables you to add new features and functionality to an existing class without modifying the existing class.
- **IS-A relationship**
- Eg Employee: Manager, Secretary, Developer, Accountant

Four Pillars of OOPs

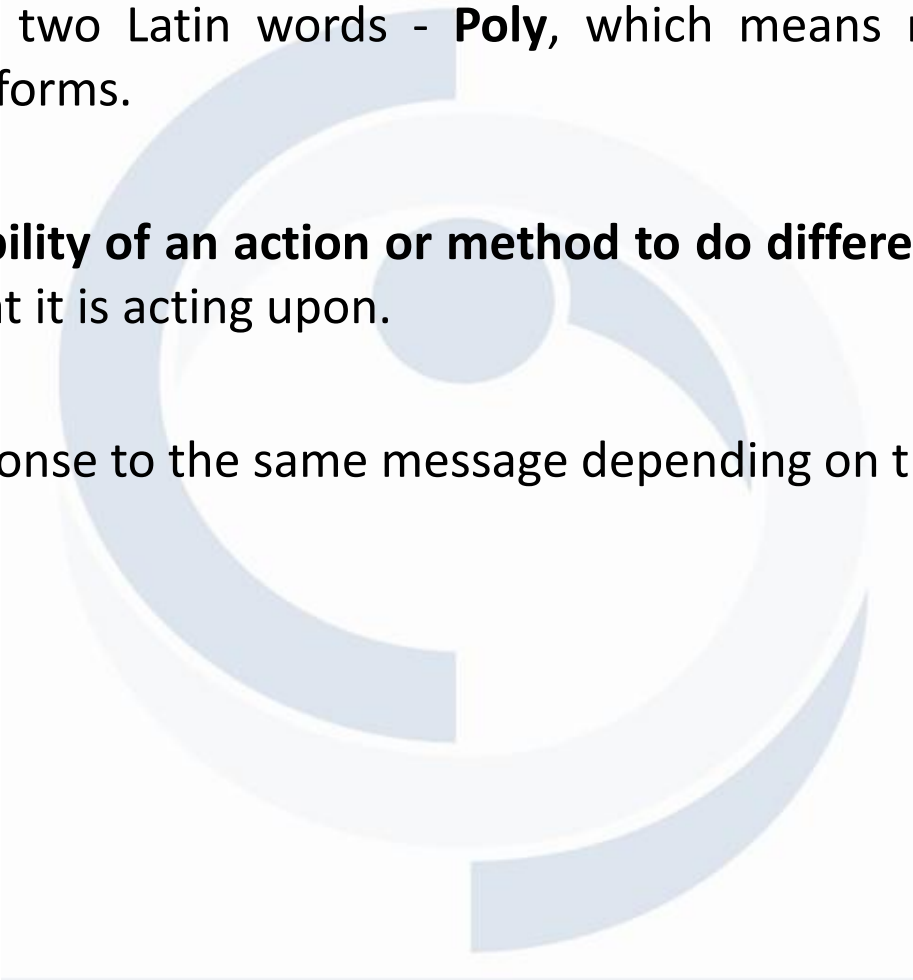
Types of inheritance

- *Single inheritance*
- *Multilevel inheritance*
- *Multiple Inheritance*

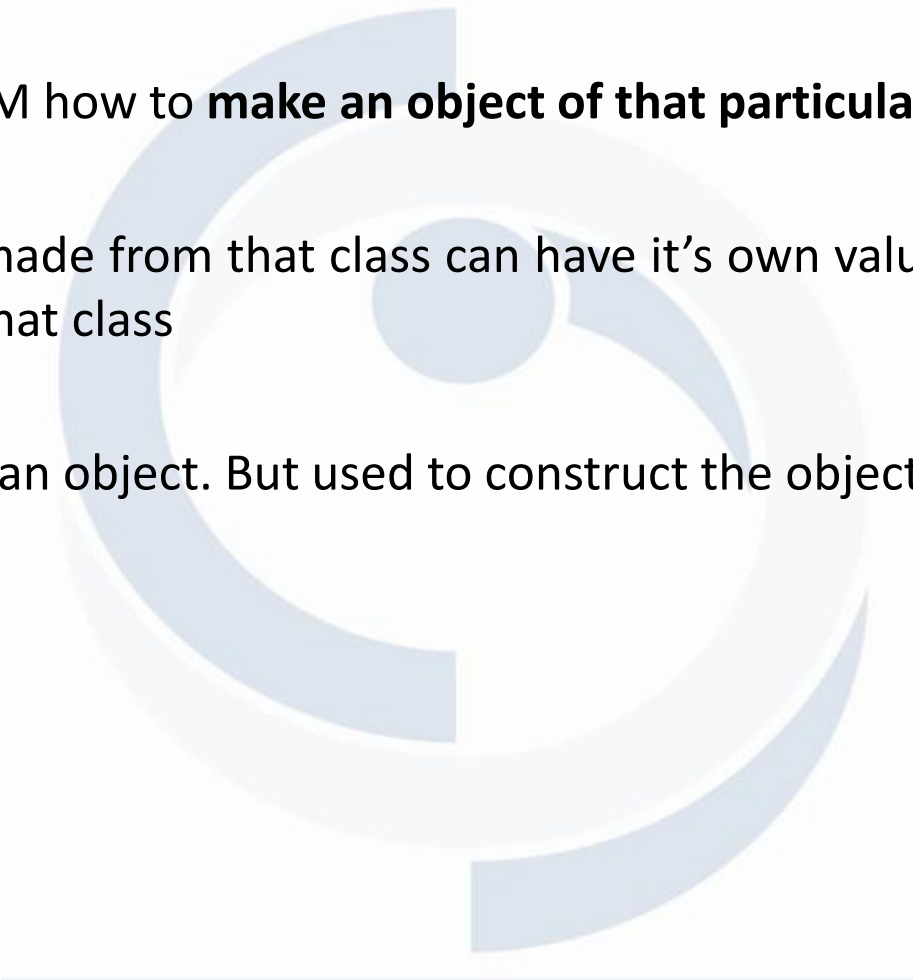


Four Pillars of OOPs

4. Polymorphism

- Derived from two Latin words - **Poly**, which means many, and **morph**, which means forms.
 - It is the **capability of an action or method to do different things based on the object** that it is acting upon.
 - Different response to the same message depending on the object type
- 

Classes v/s Objects

- A class is a **blue print** or **template** for an object.
 - It tells the JVM how to **make an object of that particular type**.
 - Each object made from that class can have it's own values for the instance variables of that class
 - A class is not an object. But used to construct the object.
- 

Java Basics

Variables

- A variable is the name that refers to a memory location where some data value is stored
- Each variable that is used in a program must be declared
- 2 types of variables: Primitive and Reference

Variable naming conventions

- A variable name must be unique
- A variable name must begin with a letter, an underscore (_), or the dollar symbol (\$), followed by a sequence of letters or digits (0 to 9), '\$', or '_' .
- A variable name should not start with a digit
- A variable name should not contain embedded white spaces
- A variable name should not consist of a keyword
- A variable name in Java is case sensitive

Java Basics - continued

- It must not be a keyword, boolean literal or reserved word null.
- It must be unique within its scope.
- Must have a type
- Must have a name (Identifier)
 - Starts with a letter, underscore (_), or dollar sign (\$)
 - Cannot be a reserved word (public, void, static, int, ...)

– `int total = 100;`

Keywords

- Reserved words in Java.
- You cannot use any of the following as identifiers in your programs.
- The keywords **const** and **goto** are reserved, even though they are not currently used.
- **true, false, and null** might seem like keywords, but they are actually literals; you cannot use them as identifiers in your programs.
- enum - added in 5.0
- const - not used

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

Data Types: Primitive

- There are eight primitive data types in Java

Type Name	Size (in bits)
Integrals:	
byte	8
short	16
int	32
long	64
Floating Points:	
float	32
double	64
Characters:	
char	16
Booleans:	
boolean	N/A

Data Types: Primitive

Type Name	Size	Range	Default Values
Integers			
<i>byte</i>	8 bits	-128 to 127 (-2^7 to 2^7-1)	0
<i>short</i>	16 bits	-32,768 to 32,767 (-2^{15} to $2^{15}-1$)	0
<i>int</i>	32 bits	-2,147,483,648 to 2,147,483,647 (-2^{31} to $2^{31}-1$)	0
<i>long</i>	64 bits	$\sim -9.2 \times 10^{15}$ to $\sim 9.2 \times 10^{15}$ (-2^{63} to $2^{63}-1$)	0
Floating-Point Numbers			
<i>float</i>	32 bits	$\sim -3.4 \times 10^{38}$ to $\sim 3.4 \times 10^{38}$	0.0f
<i>double</i>	64 bits	$\sim -1.8 \times 10^{308}$ to $\sim 1.8 \times 10^{308}$	0.0d
Character			
<i>char</i>	16 bits	0 to 65,535	'\u0000'
Boolean			
<i>boolean</i>	N/A	false or true	false

Literals

- **String literals**

Ex : “Core Java Training”

- **Boolean literals**

Ex: true, false

- **Character literals**

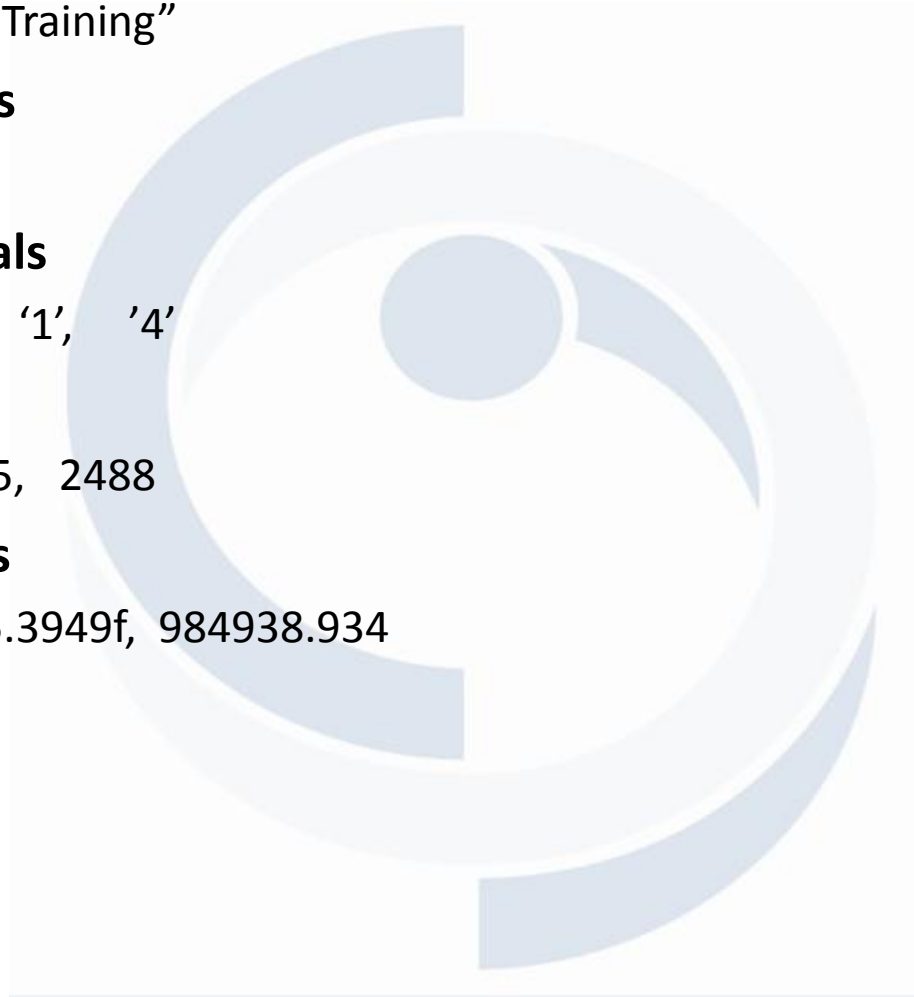
Ex : 'a', 'b', '1', '4'

- **Integer literals**

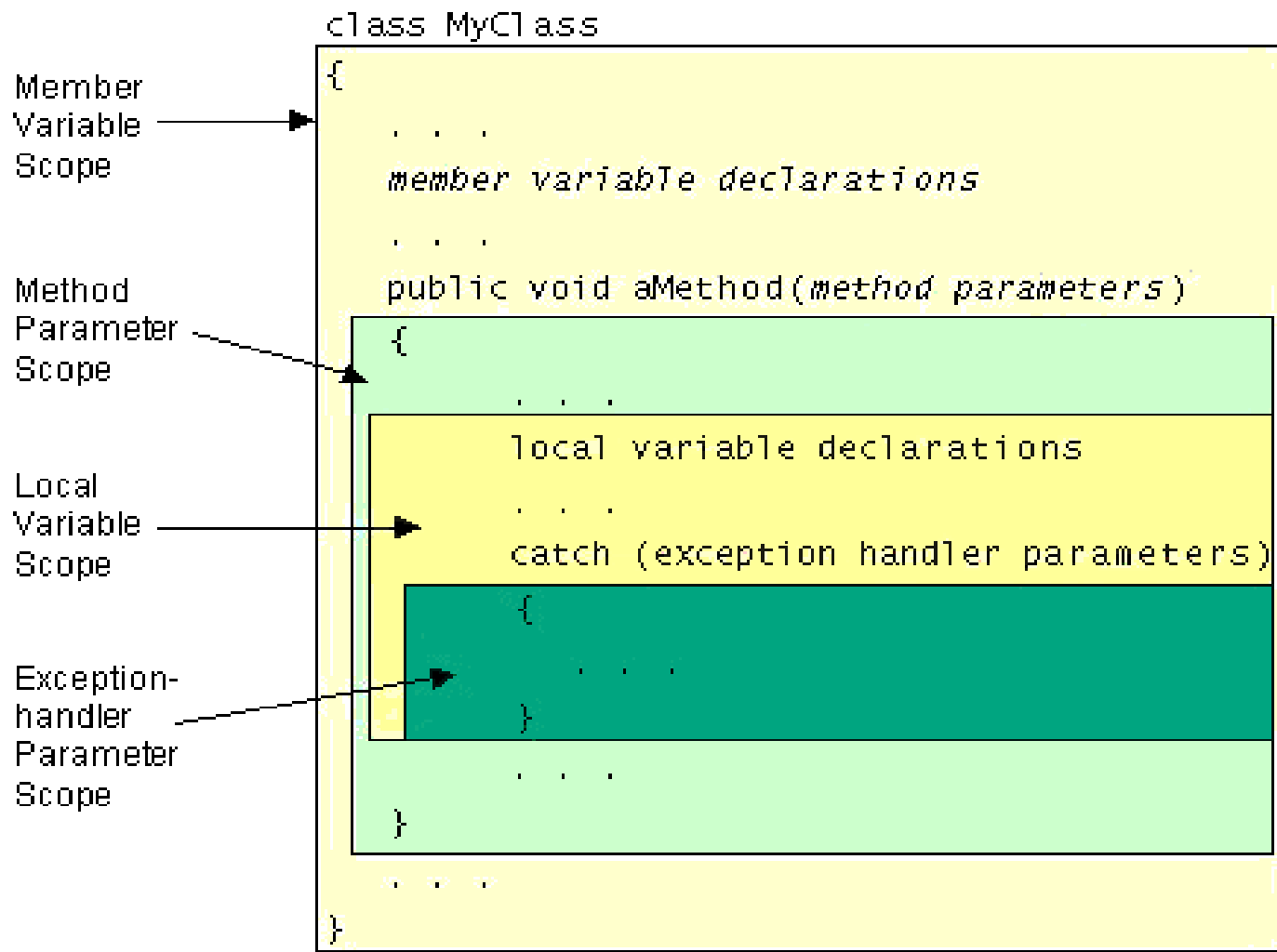
Ex: 123, 5985, 2488

- **Floating literals**

Ex: 343.244, 5.3949f, 984938.934

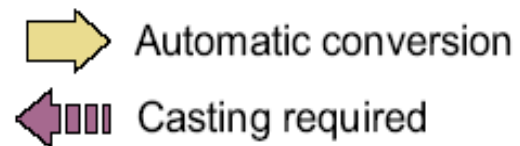
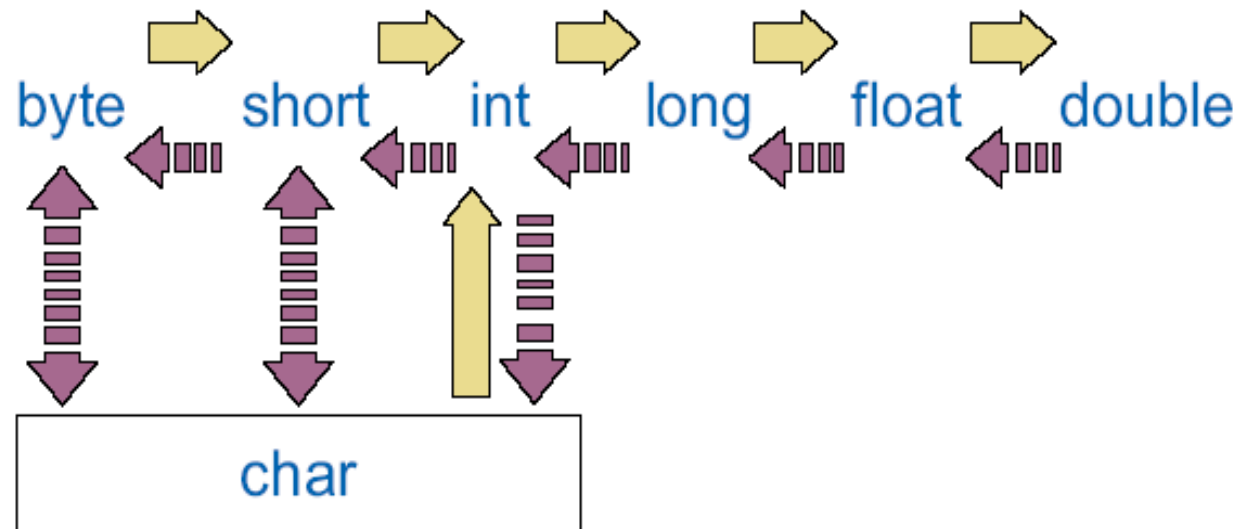


Scope of variables



Type conversion of Primitive Data Types

- Java implicitly cast to bigger data types
- When placing larger to smaller types, you must use explicit casting to mention type name to which you are converting



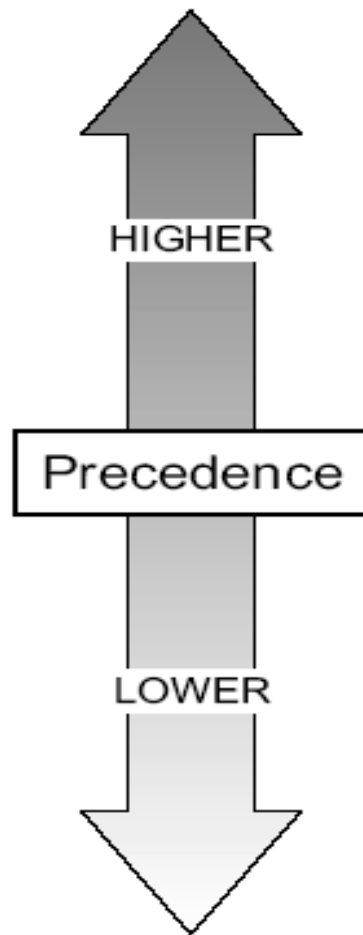
Automatic casting and conversion of primitives

When considering two operands in arithmetic expression:

- If either type is double, other is cast to a double
- If either type is float, other is cast to float.
- If either type is long, other is cast to long.
- Otherwise both the operands are converted to int.

```
int i, j;  
long x;  
float f;  
double d;  
// ...  
i = j; // exact match, no conversion needed  
x = i; // OK  
i = x; // NO, information could be lost  
i = (int) x; // allowed, explicit cast  
d = i; // OK  
f = d; // NO
```

Operator Priority or Precedence



Postfix	<code>[] .(params) ++ --</code>
Unary	<code>++ -- + - ~ !</code>
Creation and Cast	<code>new (type)</code>
Multiplicative	<code>* / %</code>
Additive	<code>+ -</code>
Shift	<code><< >> >>></code>
Relational	<code>< > <= >= instanceof</code>
Equality	<code>== !=</code>
Bitwise AND	<code>&</code>
Bitwise XOR	<code>^</code>
Bitwise OR	<code> </code>
Logical AND	<code>&&</code>
Logical OR	<code> </code>
Conditional	<code>?:</code>
Assignment	<code>= *= /= %= += -= >>= <<= >>>= &= ^= =</code>

Increment Operator ++

++ adds 1 to a variable

- It can be used as a statement by itself, or within an expression
- It can be put *before* or *after* a variable
- If before a variable (**preincrement**), it means to add one to the variable, then use the result

```
int x=5;
```

```
int y=0;
```

```
y=++x;    // y=6 x=6
```

- If put after a variable (**postincrement**), it means to use the current value of the variable, then add one to the variable

```
int x=5;
```

```
int y=0;
```

```
y=x++;    // y=5 x=6
```

Decrement Operator --

-- subtracts 1 from a variable

- It can be used as a statement by itself, or within an expression
- It can be put *before* or *after* a variable
- If before a variable (**predecrement**), it means to subtract one from the variable, then use the result

```
int x=5;
```

```
int y=0;
```

```
y=--x; // y=5 x=5
```

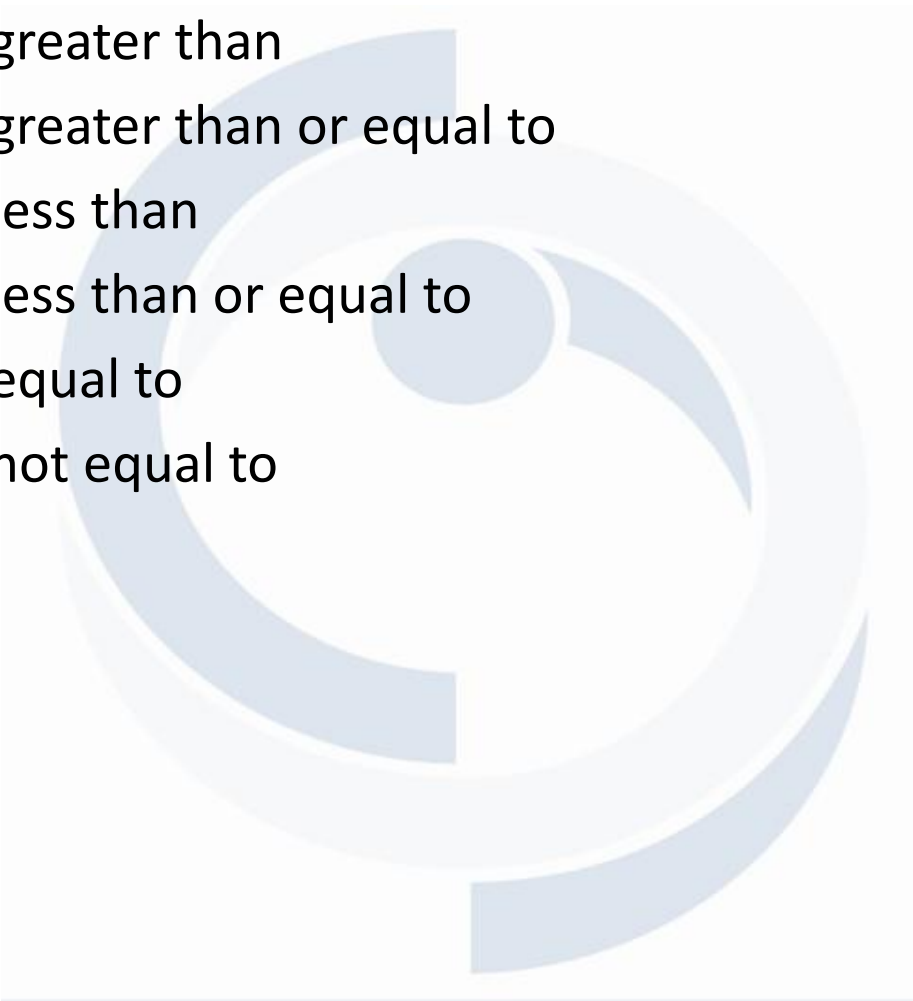
- If put after a variable (**postdecrement**), it means to use the current value of the variable, then subtract one from the variable

```
int x=5;
```

```
int y=0;
```

```
y=x--; // y=5 x=4
```

Relational operators



>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to

Logical Operators

- Boolean expressions can use the following *logical operators*:

! Logical NOT

& & Logical AND

| | Logical OR

- They all take boolean operands and produce boolean results
- Logical NOT is a unary operator (it operates on one operand)
- Logical AND and logical OR are binary operators (each operates on two operands)

Comments

- Comments are used to tell you what something does in English.
- Used to disable parts of your program if you need to remove them temporarily.
- **// single line comment**
- **/* */ block comment**



Java Statements

Assignment statement

Assign the value to variable

Form : <variable> = <expression>

```
int x =5;
```

```
int y=0;
```

```
y*=x; // y=y*x;
```

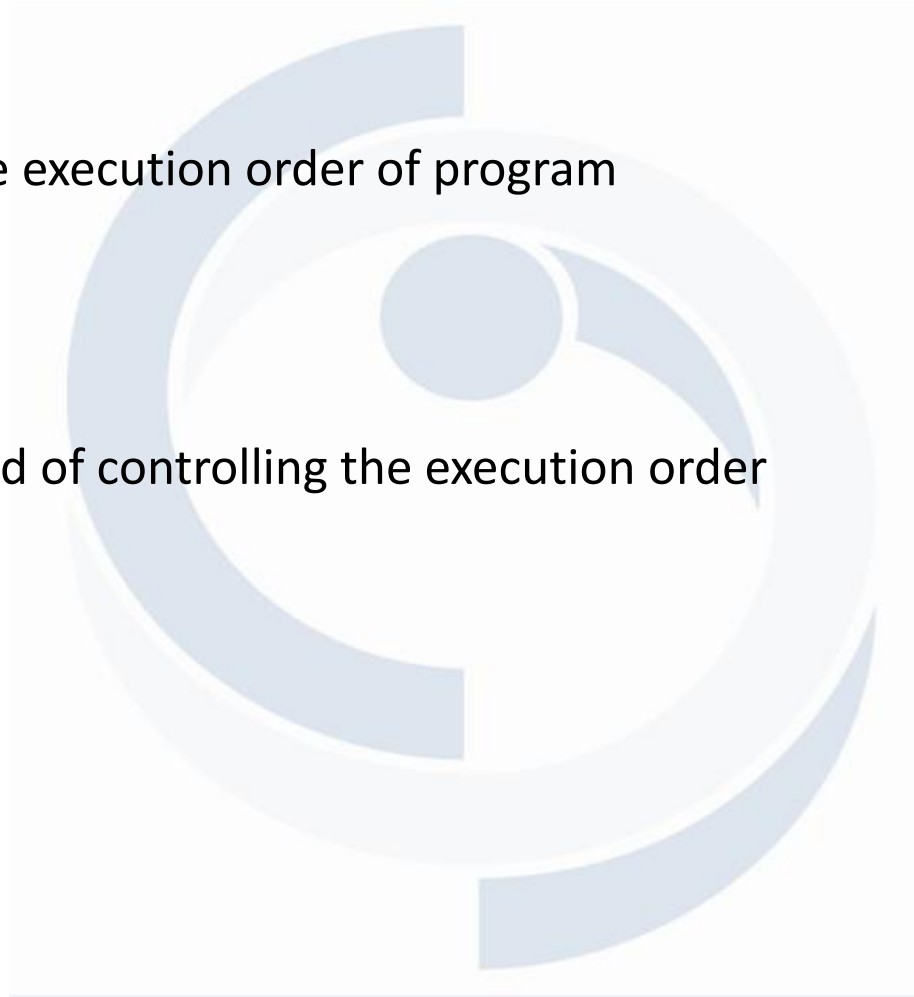


Java Statements

Control flow statements

To change the execution order of program

As the method of controlling the execution order



The if statement


```
public class WhatIf
{
    public static void main( String[] args )
    {
        int people = 20;
        int cats = 30;
        int dogs = 15;
        if ( people < cats )
            { System.out.println( "Too many cats! The world is doomed!" ); }
        if ( people > cats )
            { System.out.println( "Not many cats! The world
            is saved!" ); }
        if ( people < dogs )
            { System.out.println( "The world is drooled on!");    }
    }
}
```

The else-if statement

```
public class ElseAndIf
{
    public static void main( String[] args )
    {
        int people = 30;
        int cars = 40;
        int buses = 15;
        if ( cars > people )
            { System.out.println( "We should take the cars." ); }
        else if ( cars < people )
            { System.out.println( "We should not take the cars." ); }
        else
            { System.out.println( "We can't decide." ); }
    }
}
```

Nested if statement

```
if (<cond. expr.>
{
    if (<cond. expr.>
    {
        // ...
        <statement>
    }
}
```



The *switch* statement

The syntax is:

```
switch (expression) {  
    case value1 :  
        statements ;  
        break ;  
    case value2 :  
        statements ;  
        break ;  
        ...(more cases)...  
    default :  
        statements ;  
        break ;  
}
```



```
switch (cardValue) {
```

```
    case 1:
```

```
        System.out.print("Ace");
```

```
        break;
```

```
    case 11:
```

```
        System.out.print("Jack");
```

```
        break;
```

```
    case 12:
```

```
        System.out.print("Queen");
```

```
        break;
```

```
    case 13:
```

```
        System.out.print("King");
```

```
        break;
```

```
    default:
```

```
        System.out.print(cardValue);
```

```
        break;
```

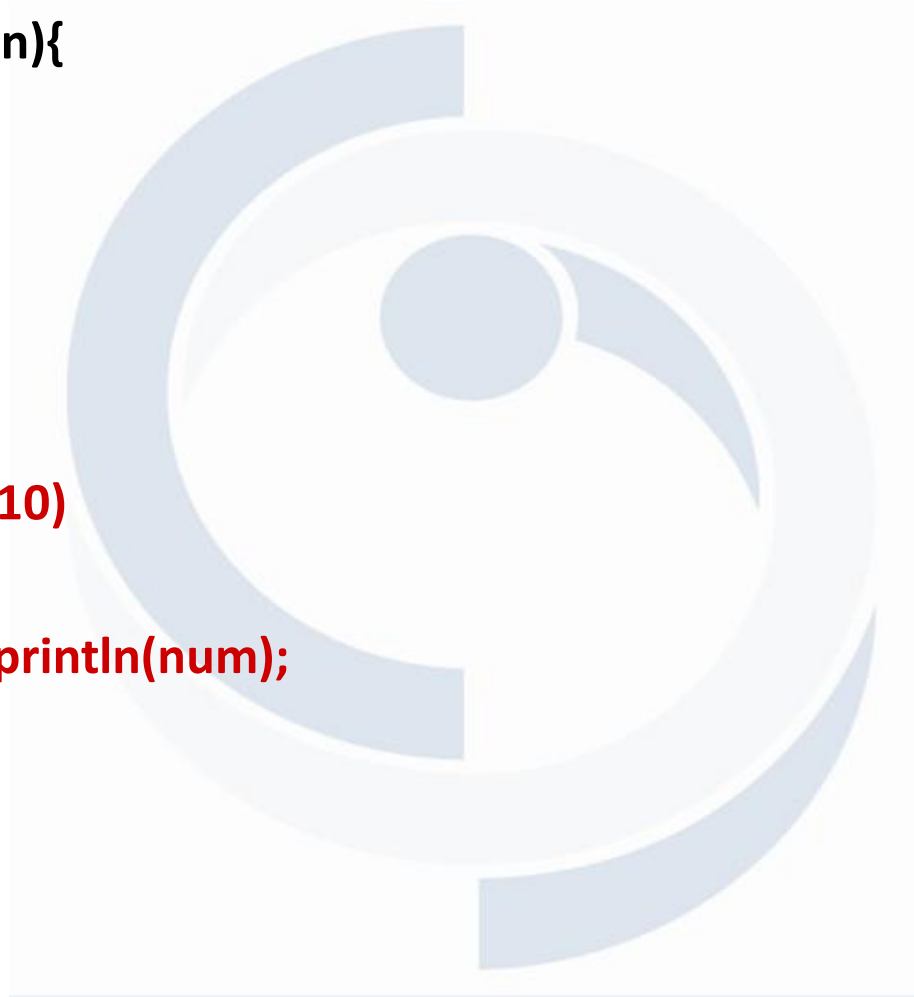
```
}
```

while loop

- General form

```
while(condition){  
  
    //statement  
}
```

```
int num=1;  
while(num <=10)  
{  
    System.out.println(num);  
    num++;  
}
```

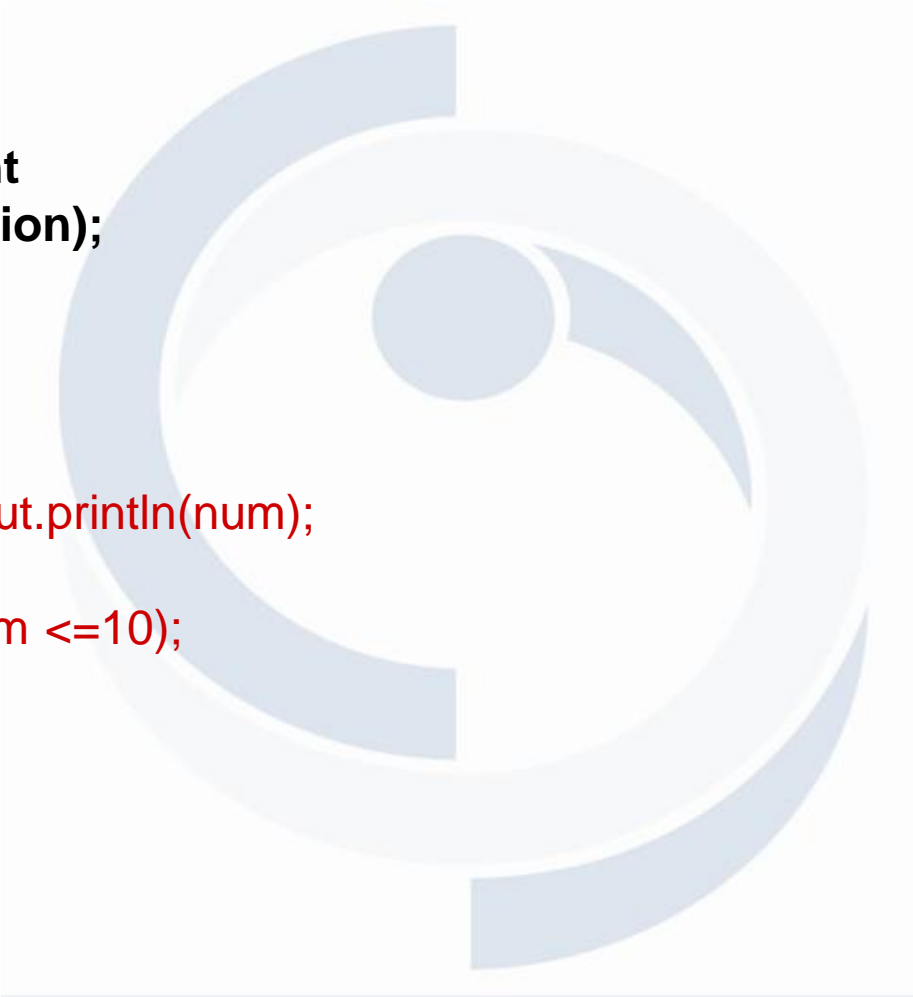


The *do... while* loop

General form

```
do{  
    //statement  
}while(condition);
```

```
int num=1;  
do  
{  
    System.out.println(num);  
    num++;  
}while((num <=10);
```



The *for* loop

General form

```
for(initialization ; condition; increment/ decrement){
```

```
    //statement
```

```
}
```

```
for (int i=1;i<=10;i++)  
{  
    System.out.println(i);  
}
```

Leaving a loop

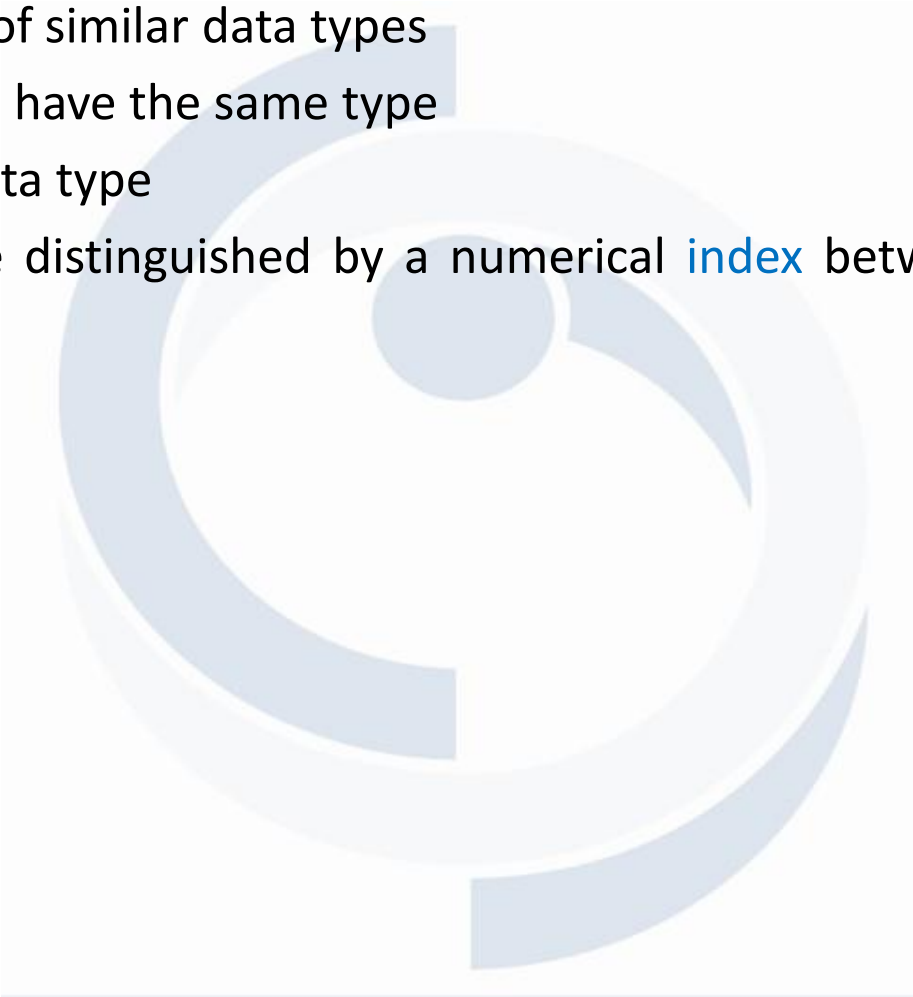
- Break leaves the loop
- Continue ends the current pass through the loop

```
for(int i=0; i<10; i = i + 1) {  
    for(int k=0; k<20; k = k + 1) {  
        // ...  
        if (...) break;  
        // ...  
        if (...) continue;  
        // ...  
    } // end inner loop  
    System.out.println("After inner loop");  
} // end outer loop
```

● Ends inner loop

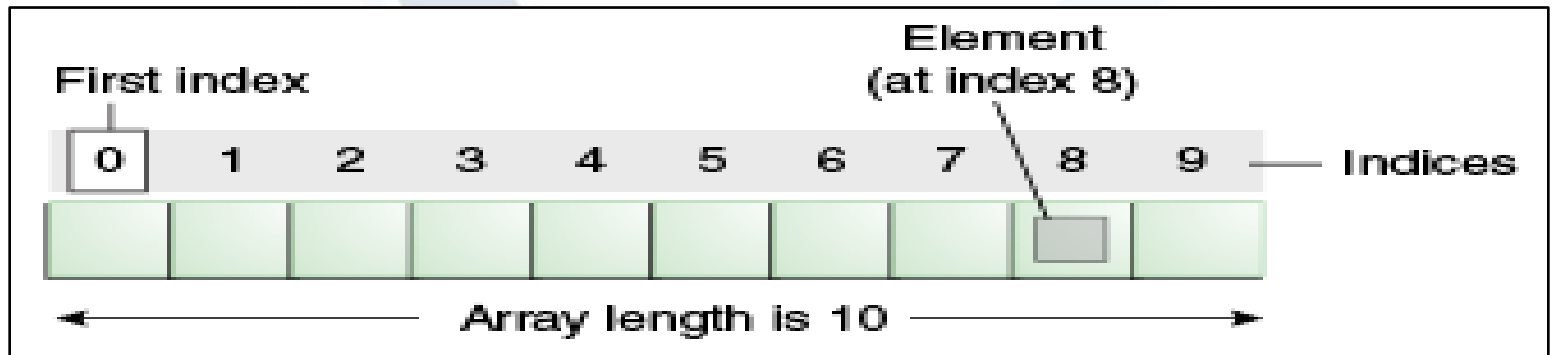
● Ends this pass
of inner loop

Arrays

- Array is group of similar data types
 - All values must have the same type
 - It is indexed data type
 - The values are distinguished by a numerical **index** between 0 and array size minus 1
- 

Arrays

- The length of an array is established when the array is created.
- After creation, its length is fixed
- Each item in an array is called an *element*, and each element is accessed by its numerical *index*
- Numbering begins with 0. The 9th element, for example, would therefore be accessed at index 8



Arrays

	0	1	2	3	4	5	6	7	8	9
myArray	12	43	6	83	14	-57	109	12	0	6

Examples:

```
x = myArray[1];    // sets x to 43
```

```
myArray[4] = 99;    // replaces 14 with 99
```

```
m = 5;
```

```
y = myArray[m];    // sets y to -57
```

```
z = myArray[myArray[9]]; // sets z to 109
```

Array declaration and creation(definition)

- Arrays *are* objects
- Creating arrays is like creating other objects:
 - the *declaration* only provides type information
 - the **new** *definition* actually allocates space
 - declaration and definition may be separate or combined

- declaration:

```
int num[ ]; or int []num; // declaration
```

- definition:

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

- Declaration and definition

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

- Initialization int num[]={ 1,2,3,4,5,6};

Length of an Array

- Arrays are objects
- Every array has an instance constant, `length`, that tells how large the array is
- Example:

```
for (int i = 0; i < num.length; i++)  
    System.out.println(num[i]);
```
- Use of `length` is always preferred over using a constant such as 5
- Strings have a `length()` *method*!

Two dimensional Array

- The elements of an array can be arrays
 - Declaration: `int table[][]`;
 - Definition: `table = new int[3][2]`;
 - Combined: `int table[][] = new int[3][2]`;
- The first index (3) is called the **row** index; the second index (2) is the **column** index
- An array like this is called a **two-dimensional array**

**`int table[][] = new int[3][2];` or,
`int table[][] = { {1, 2}, {3, 6}, {7, 8} };`**

	0	1
0	1	2
1	3	6
2	7	8

Two dimensional Array

- `int table[][] = new int[3][2];`
- The length of this array is the number of *rows*: `table.length` is 3
- Each row contains an array
- To get the number of *columns*, pick a row and ask for its length:
`table[0].length` is 2

Usage of Loops

- Use the **for** loop if you know ahead of time how many times you want to go through the loop
Example: Print a 12-month calendar
- Use the **while** loop in almost all other cases
Example: Repeat Scott's problem until you get to 1
- Use the **do-while** loop if you must go through the loop at least once before it makes sense to do the test
Example: Ask for the password until user gets it right
- The **if-else** statement chooses one of two statements, based on the value of a **boolean** expression
- The **switch** statement chooses one of several statements, based on the value on an integer (**int**, **byte**, **short**) or a **char** expression