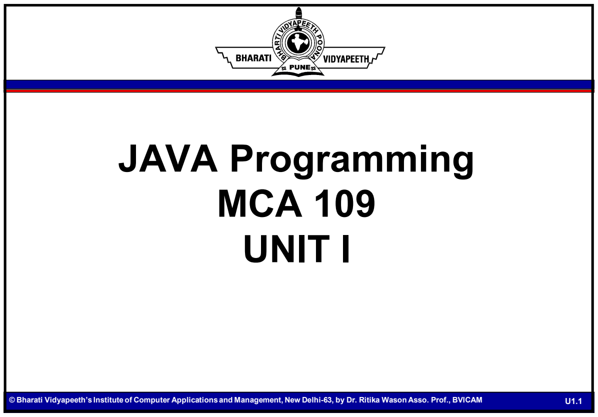
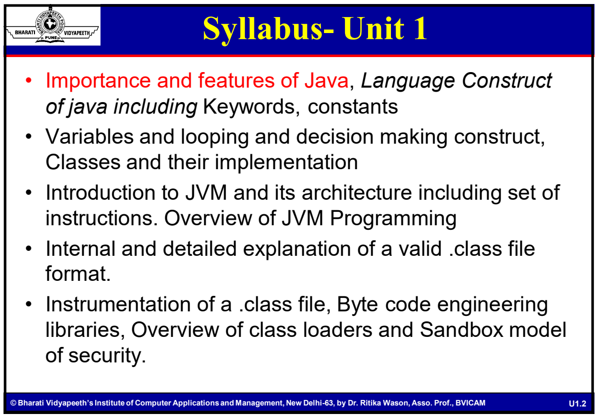
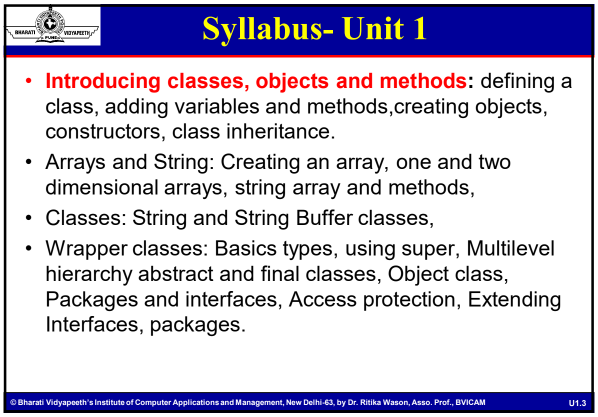


JAVA Programming
MCA 109
UNIT I



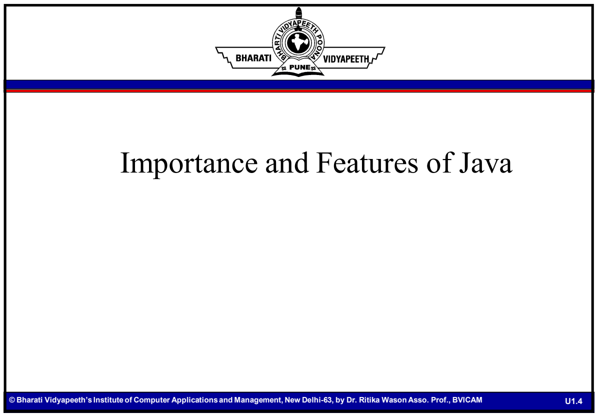
Syllabus- Unit 1

- Importance and features of Java, *Language Construct of java including* Keywords, constants
- Variables and looping and decision making construct, Classes and their implementation
- Introduction to JVM and its architecture including set of instructions. Overview of JVM Programming
- Internal and detailed explanation of a valid .class file format.
- Instrumentation of a .class file, Byte code engineering libraries, Overview of class loaders and Sandbox model of security.

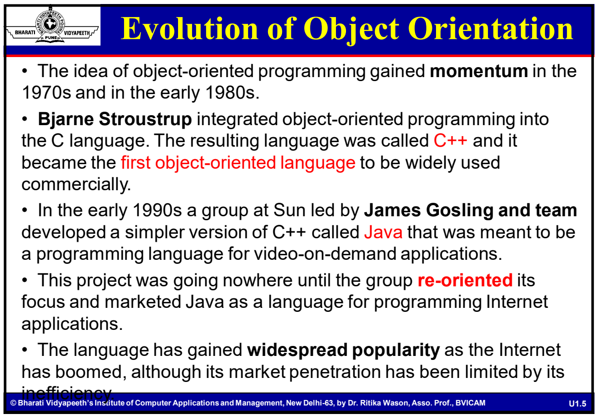


Syllabus- Unit 1

- **Introducing classes, objects and methods:** defining a class, adding variables and methods,creating objects, constructors, class inheritance.
- Arrays and String: Creating an array, one and two dimensional arrays, string array and methods,
- Classes: String and String Buffer classes,
- Wrapper classes: Basics types, using super, Multilevel hierarchy abstract and final classes, Object class, Packages and interfaces, Access protection, Extending Interfaces, packages.

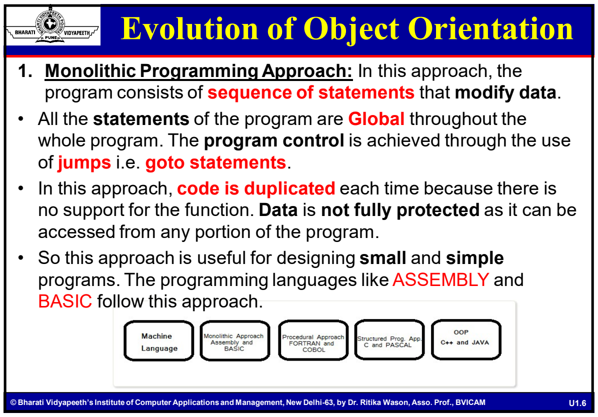


Importance and Features of Java



Evolution of Object Orientation

- The idea of object-oriented programming gained **momentum** in the 1970s and in the early 1980s.
- **Bjarne Stroustrup** integrated object-oriented programming into the C language. The resulting language was called C++ and it became the first object-oriented language to be widely used commercially.
- In the early 1990s a group at Sun led by **James Gosling and team** developed a simpler version of C++ called Java that was meant to be a programming language for video-on-demand applications.
- This project was going nowhere until the group **re-oriented** its focus and marketed Java as a language for programming Internet applications.
- The language has gained **widespread popularity** as the Internet has boomed, although its market penetration has been limited by its inefficiency.



Evolution of Object Orientation

1. **Monolithic Programming Approach:** In this approach, the program consists of **sequence of statements** that **modify data**.

- All the **statements** of the program are **Global** throughout the whole program. The **program control** is achieved through the use of **jumps** i.e. **goto statements**.
- In this approach, **code is duplicated** each time because there is no support for the function. **Data** is **not fully protected** as it can be accessed from any portion of the program.
- So this approach is useful for designing **small** and **simple** programs. The programming languages like ASSEMBLY and BASIC follow this approach.

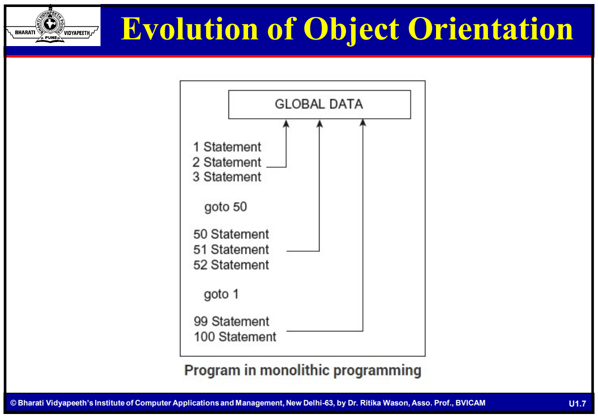
Machine Language

Monolithic Approach Assembly and BASIC

Procedural Approach FORTRAN and COBOL

Structured Prog. App. C and PASCAL

OOP C++ and JAVA



Evolution of Object Orientation

GLOBAL DATA

1 Statement
2 Statement
3 Statement

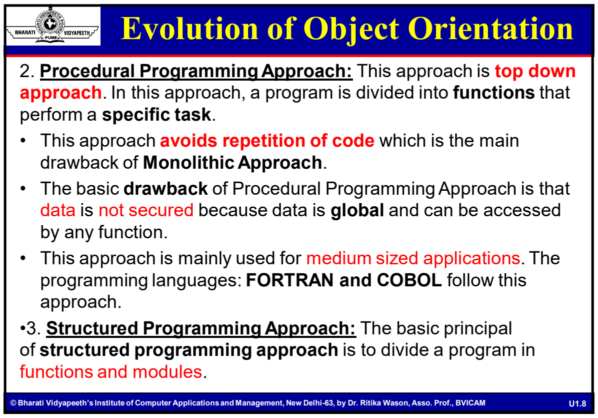
goto 50

50 Statement
51 Statement
52 Statement

goto 1

99 Statement
100 Statement

Program in monolithic programming

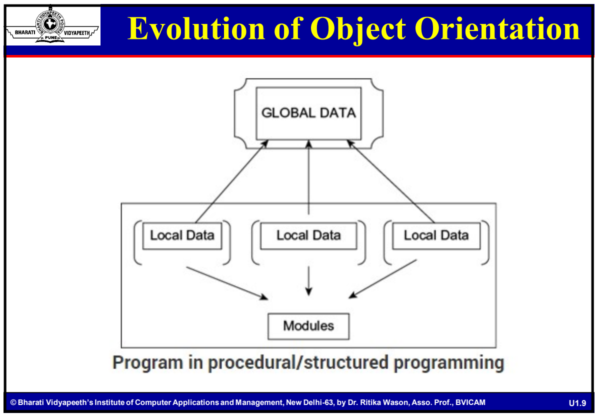


Evolution of Object Orientation

2. **Procedural Programming Approach:** This approach is **top down approach**. In this approach, a program is divided into **functions** that perform a **specific task**.

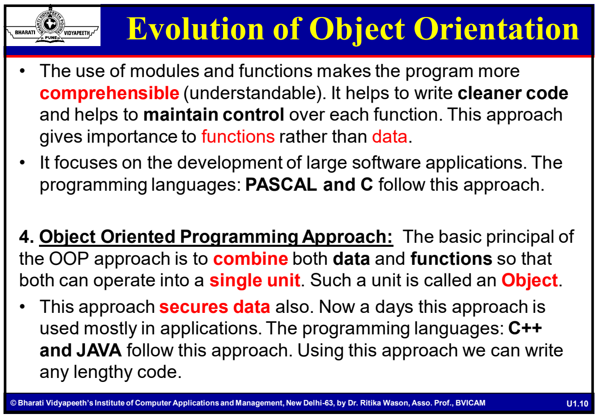
- This approach **avoids repetition of code** which is the main drawback of **Monolithic Approach**.
- The basic **drawback** of Procedural Programming Approach is that data is not secured because data is **global** and can be accessed by any function.
- This approach is mainly used for medium sized applications. The programming languages: **FORTRAN and COBOL** follow this approach.

•3. **Structured Programming Approach:** The basic principal of **structured programming approach** is to divide a program in functions and modules.



Evolution of Object Orientation

Program in procedural/structured programming

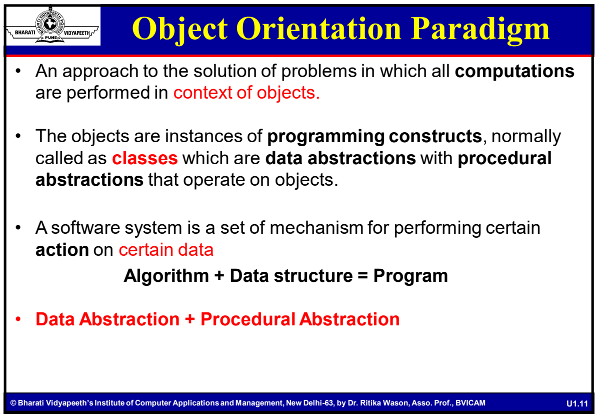


Evolution of Object Orientation

- The use of modules and functions makes the program more **comprehensible** (understandable). It helps to write **cleaner code** and helps to **maintain control** over each function. This approach gives importance to functions rather than data.
- It focuses on the development of large software applications. The programming languages: **PASCAL and C** follow this approach.

4. Object Oriented Programming Approach: The basic principal of the OOP approach is to **combine** both **data** and **functions** so that both can operate into a **single unit**. Such a unit is called an **Object**.

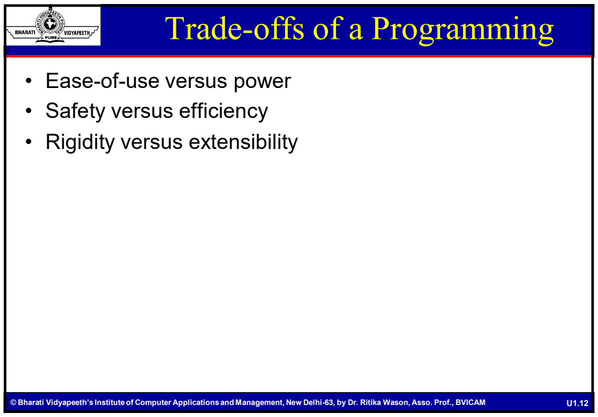
- This approach **secures data** also. Now a days this approach is used mostly in applications. The programming languages: **C++ and JAVA** follow this approach. Using this approach we can write any lengthy code.



Object Orientation Paradigm

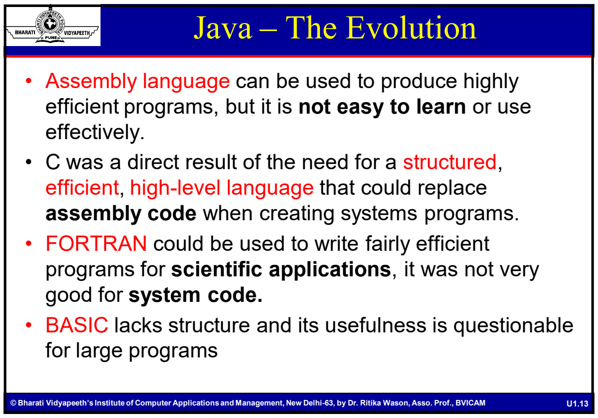
- An approach to the solution of problems in which all **computations** are performed in context of objects.
- The objects are instances of **programming constructs**, normally called as **classes** which are **data abstractions** with **procedural abstractions** that operate on objects.
- A software system is a set of mechanism for performing certain **action** on certain data

  Algorithm + Data structure = Program
- **Data Abstraction + Procedural Abstraction**



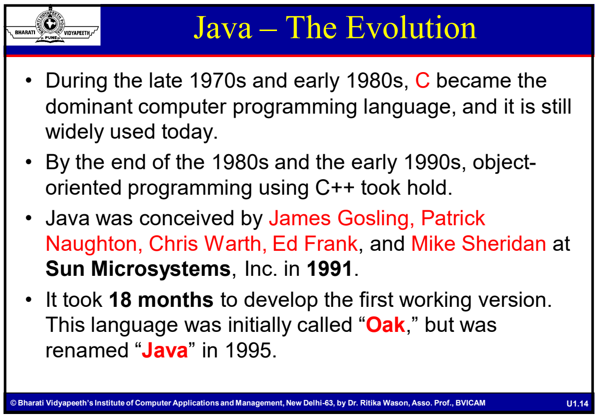
Trade-offs of a Programming

- Ease-of-use versus power
- Safety versus efficiency
- Rigidity versus extensibility



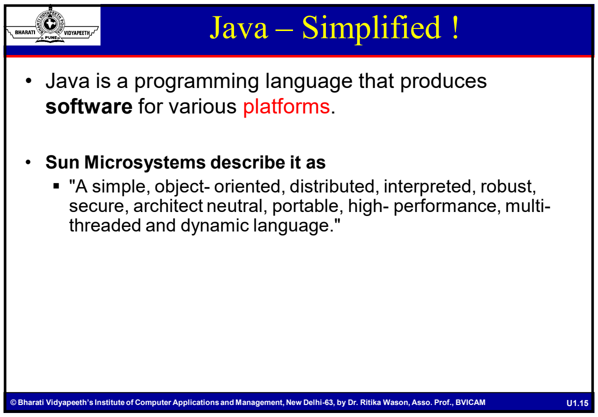
Java – The Evolution

- Assembly language can be used to produce highly efficient programs, but it is **not easy to learn** or use effectively.
- C was a direct result of the need for a structured, efficient, high-level language that could replace **assembly code** when creating systems programs.
- FORTRAN could be used to write fairly efficient programs for **scientific applications**, it was not very good for **system code.**
- BASIC lacks structure and its usefulness is questionable for large programs



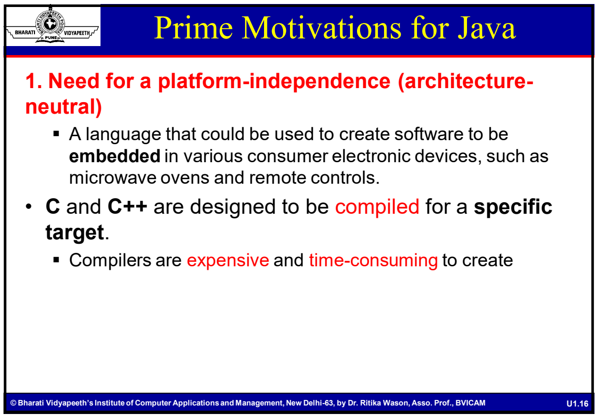
Java – The Evolution

- During the late 1970s and early 1980s, C became the dominant computer programming language, and it is still widely used today.
- By the end of the 1980s and the early 1990s, object-oriented programming using C++ took hold.
- Java was conceived 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.



Java – Simplified !

- Java is a programming language that produces **software** for various platforms.
- **Sun Microsystems describe it as**
  - "A simple, object- oriented, distributed, interpreted, robust, secure, architect neutral, portable, high- performance, multi-threaded and dynamic language."

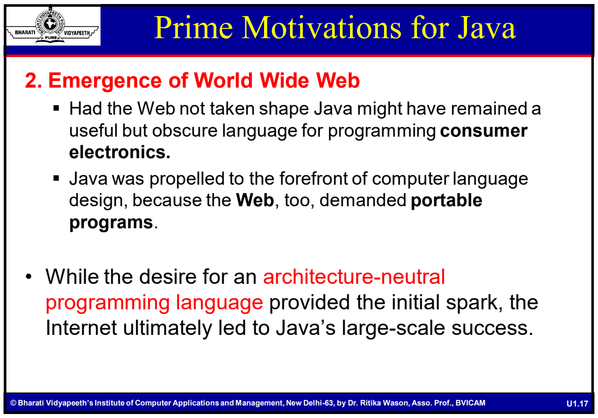


Prime Motivations for Java

1. Need for a platform-independence (architecture-neutral)

- A language that could be used to create software to be **embedded** in various consumer electronic devices, such as microwave ovens and remote controls.

- **C** and **C++** are designed to be compiled for a **specific target**.
  - Compilers are expensive and time-consuming to create

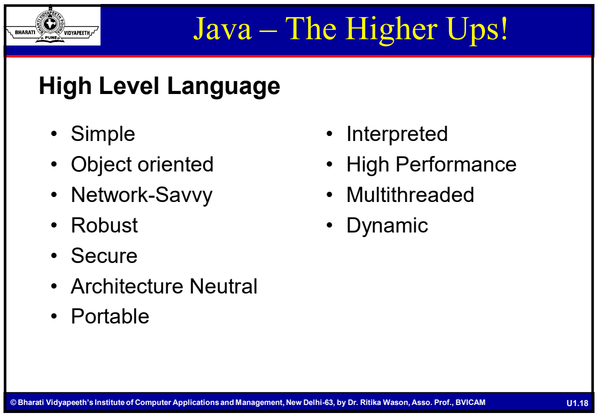


Prime Motivations for Java

2. Emergence of World Wide Web

- Had the Web not taken shape Java might have remained a useful but obscure language for programming **consumer electronics.**
- Java was propelled to the forefront of computer language design, because the **Web**, too, demanded **portable programs**.

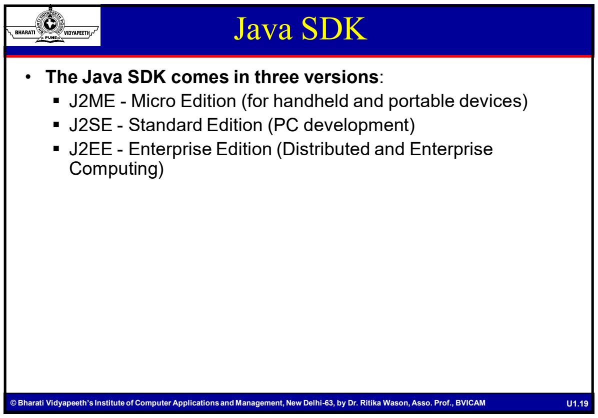
- While the desire for an architecture-neutral programming language provided the initial spark, the Internet ultimately led to Java's large-scale success.



Java – The Higher Ups!

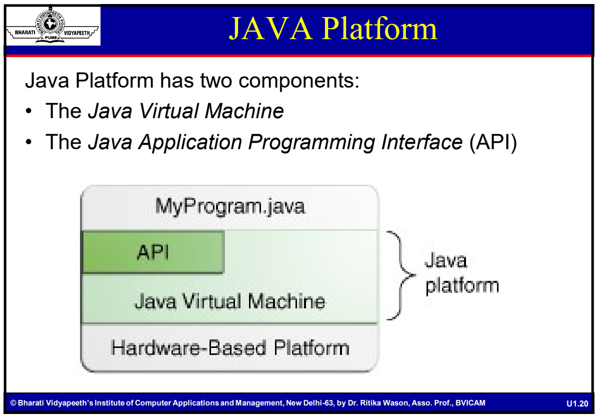
High Level Language

- Simple
- Object oriented
- Network-Savvy
- Robust
- Secure
- Architecture Neutral
- Portable
- Interpreted
- High Performance
- Multithreaded
- Dynamic



Java SDK

- **The Java SDK comes in three versions**:
  - J2ME - Micro Edition (for handheld and portable devices)
  - J2SE - Standard Edition (PC development)
  - J2EE - Enterprise Edition (Distributed and Enterprise Computing)



JAVA Platform

Java Platform has two components:

- The *Java Virtual Machine*
- The *Java Application Programming Interface* (API)

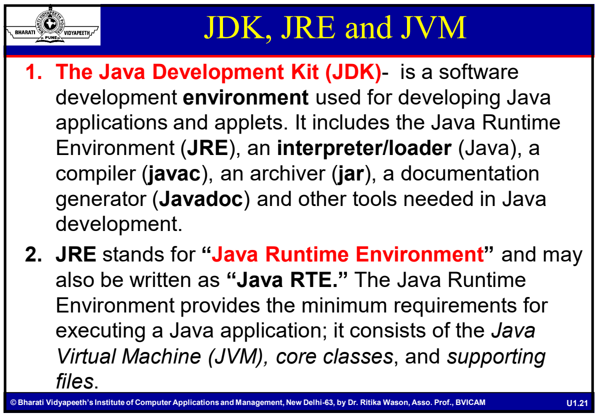
MyProgram.java

API

Java Virtual Machine

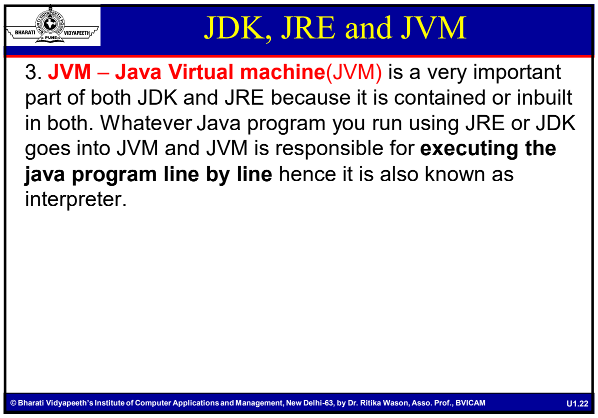
Hardware-Based Platform

Java platform



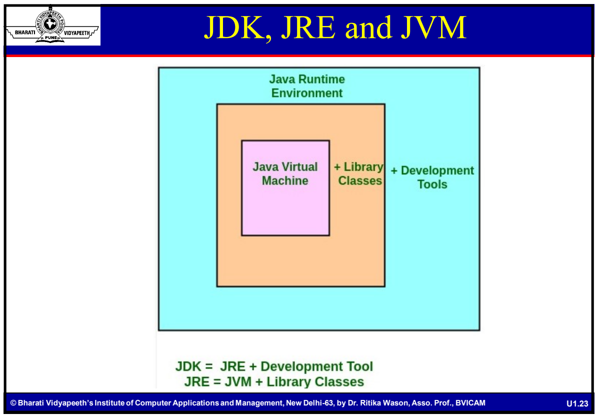
JDK, JRE and JVM

1. **The Java Development Kit (JDK)**- is a software development **environment** used for developing Java applications and applets. It includes the Java Runtime Environment (**JRE**), an **interpreter/loader** (Java), a compiler (**javac**), an archiver (**jar**), a documentation generator (**Javadoc**) and other tools needed in Java development.
2. **JRE** stands for **"Java Runtime Environment"** and may also be written as **"Java RTE."** The Java Runtime Environment provides the minimum requirements for executing a Java application; it consists of the *Java Virtual Machine (JVM)*, *core classes*, and *supporting files*.



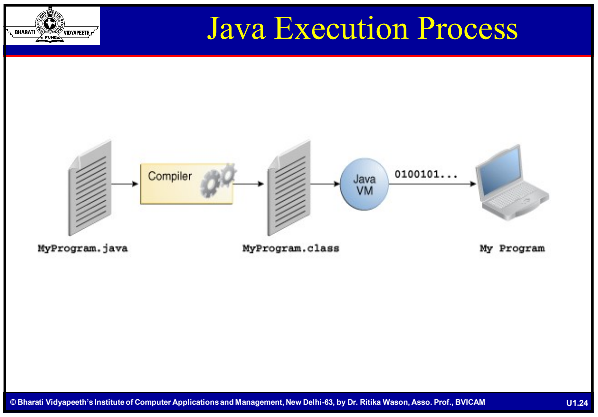
JDK, JRE and JVM

3. **JVM – Java Virtual machine**(JVM) is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for **executing the java program line by line** hence it is also known as interpreter.

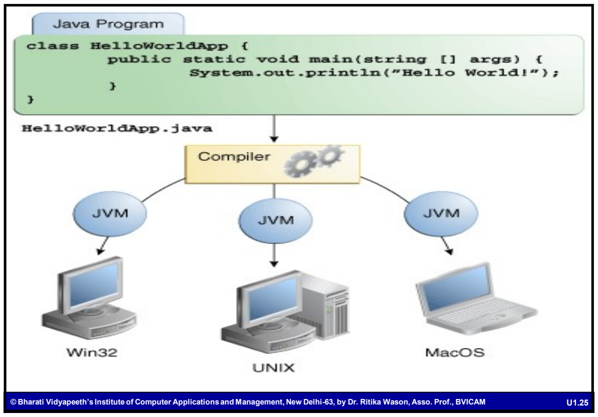


JDK, JRE and JVM

JDK = JRE + Development Tool
JRE = JVM + Library Classes



Java Execution Process



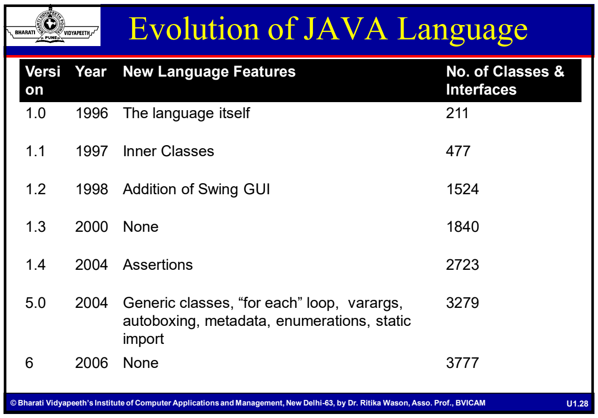
```
class HelloWorldApp {
        public static void main(string [] args) {
                System.out.println("Hello World!");
        }
}
```

Java's Magic – The Bytecode

- The key that allows Java to solve both the security and the portability problems is that the output of a Java compiler is not executable code. Rather, it is **bytecode**.
- *"Bytecode* is a highly **optimized set of instructions** designed to be executed by the Java run-time system, which is called the *Java Virtual Machine (JVM)"*.
- **JVM** is the **interpreter for bytecode.**
- Java code can run on any platform that has **JVM implemented.**
- JVM is default implemented in most of the OS by virtue of contract with Sun Microsystems.

Java's Magic – The Bytecode

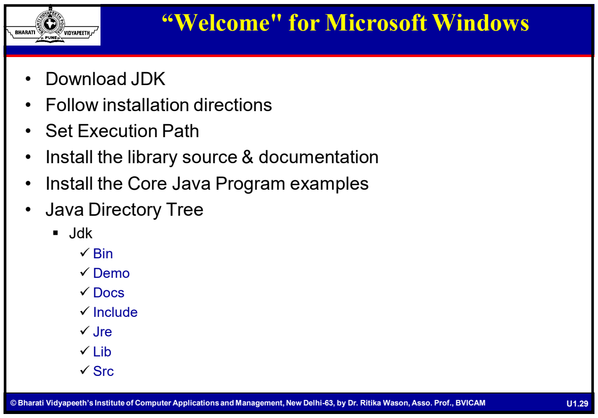
- JVM also helps to make Java **secure** as it contains the program and prevent it from generating side effects outside of the system.
- Java was designed as an **interpreted language**
- But it can also on-the-fly compile **bytecode** into **native code** in order to boost performance by **JIT.**
- JIT compiler compiles code *as it is needed*, during **execution.**



Evolution of JAVA Language

Version	Year	New Language Features	No. of Classes & Interfaces
1.0	1996	The language itself	211
1.1	1997	Inner Classes	477
1.2	1998	Addition of Swing GUI	1524
1.3	2000	None	1840
1.4	2004	Assertions	2723
5.0	2004	Generic classes, “for each” loop, varargs, autoboxing, metadata, enumerations, static import	3279
6	2006	None	3777

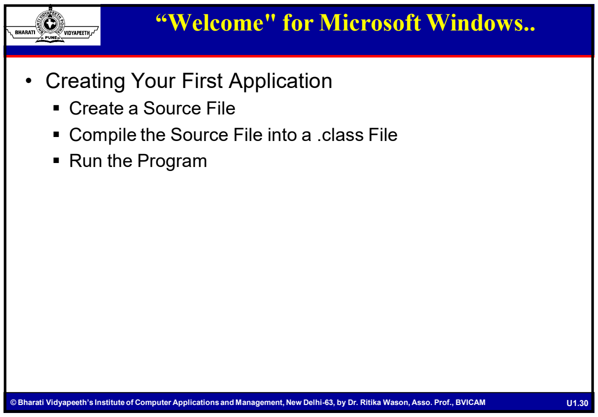
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.28



“Welcome" for Microsoft Windows

- Download JDK
- Follow installation directions
- Set Execution Path
- Install the library source & documentation
- Install the Core Java Program examples
- Java Directory Tree
  - Jdk
    - ✓ Bin
    - ✓ Demo
    - ✓ Docs
    - ✓ Include
    - ✓ Jre
    - ✓ Lib
    - ✓ Src

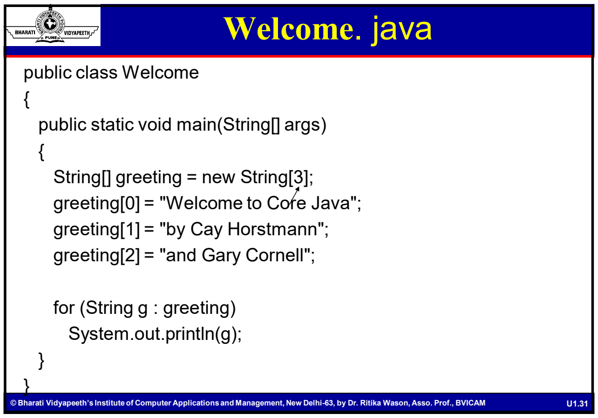
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.29



“Welcome" for Microsoft Windows..

- Creating Your First Application
  - Create a Source File
  - Compile the Source File into a .class File
  - Run the Program

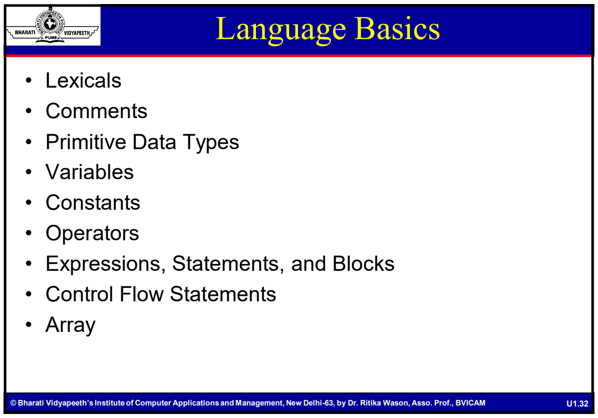
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.30



Welcome. java

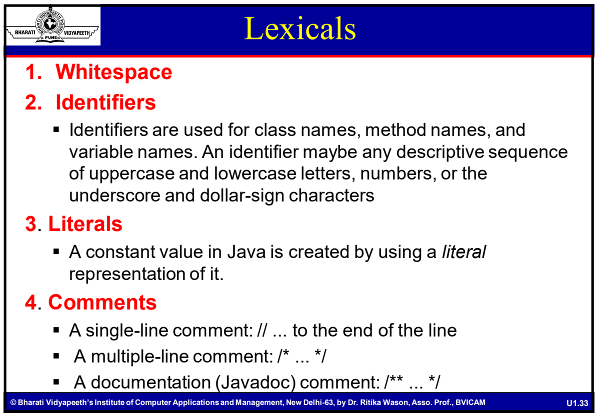
```
public class Welcome
{
  public static void main(String[] args)
  {
    String[] greeting = new String[3];
    greeting[0] = "Welcome to Core Java";
    greeting[1] = "by Cay Horstmann";
    greeting[2] = "and Gary Cornell";

    for (String g : greeting)
      System.out.println(g);
  }
}
```



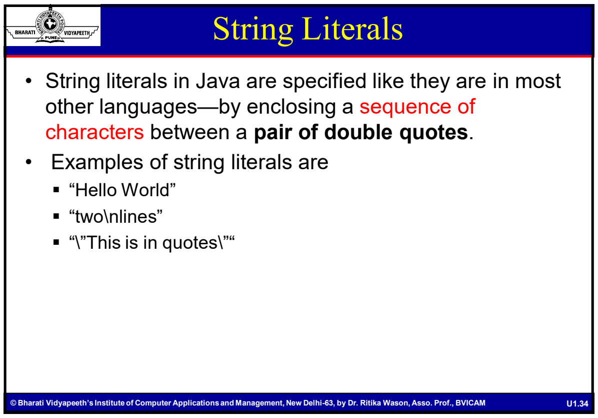
Language Basics

- Lexicals
- Comments
- Primitive Data Types
- Variables
- Constants
- Operators
- Expressions, Statements, and Blocks
- Control Flow Statements
- Array



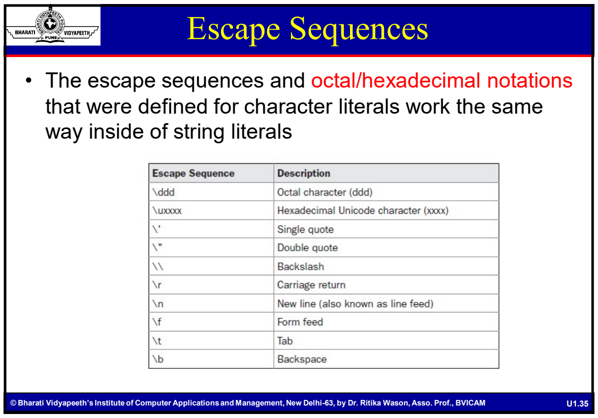
Lexicals

1. **Whitespace**
2. **Identifiers**
   - Identifiers are used for class names, method names, and variable names. An identifier maybe any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters
3. **Literals**
   - A constant value in Java is created by using a *literal* representation of it.
4. **Comments**
   - A single-line comment: // ... to the end of the line
   - A multiple-line comment: /* ... */
   - A documentation (Javadoc) comment: /** ... */



String Literals

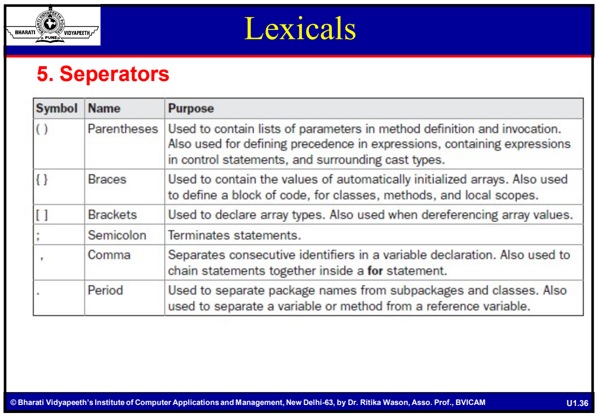
- String literals in Java are specified like they are in most other languages—by enclosing a sequence of characters between a **pair of double quotes**.
- Examples of string literals are
  - “Hello World”
  - “two\nlines”
  - “\”This is in quotes\”“



Escape Sequences

- The escape sequences and octal/hexadecimal notations that were defined for character literals work the same way inside of string literals

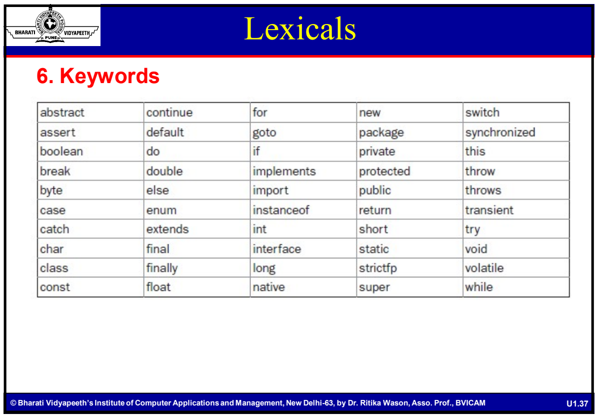
Escape Sequence	Description
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal Unicode character (xxxx)
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Backspace



Lexicals

5. Seperators

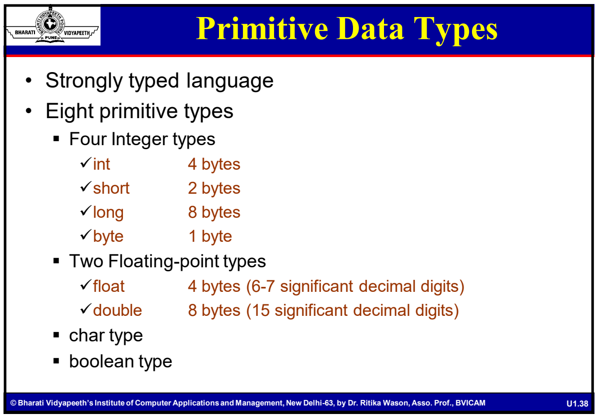
Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. 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.



Lexicals

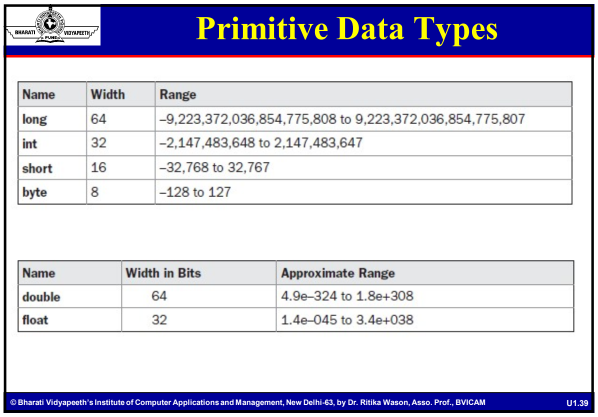
6. Keywords

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



Primitive Data Types

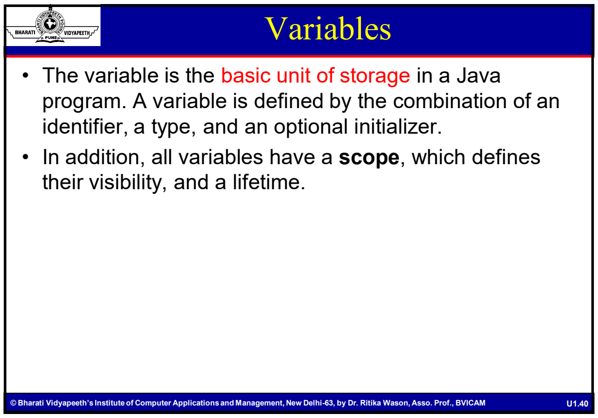
- Strongly typed language
- Eight primitive types
  - Four Integer types
    - ✓int 4 bytes
    - ✓short 2 bytes
    - ✓long 8 bytes
    - ✓byte 1 byte
  - Two Floating-point types
    - ✓float 4 bytes (6-7 significant decimal digits)
    - ✓double 8 bytes (15 significant decimal digits)
  - char type
  - boolean type



Primitive Data Types

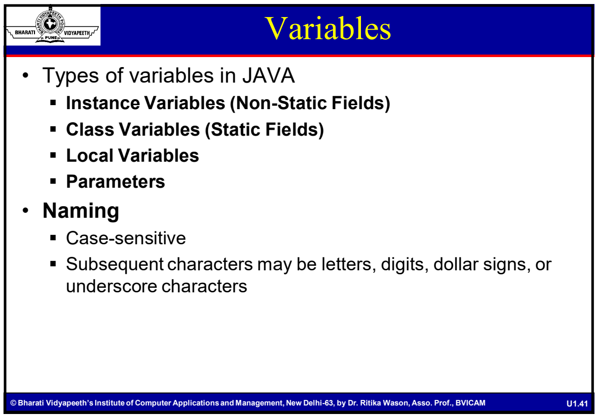
Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038



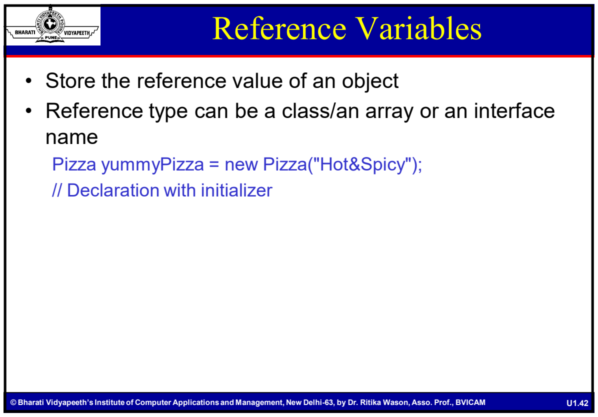
Variables

- The variable is the basic unit of storage in a Java program. 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.



Variables

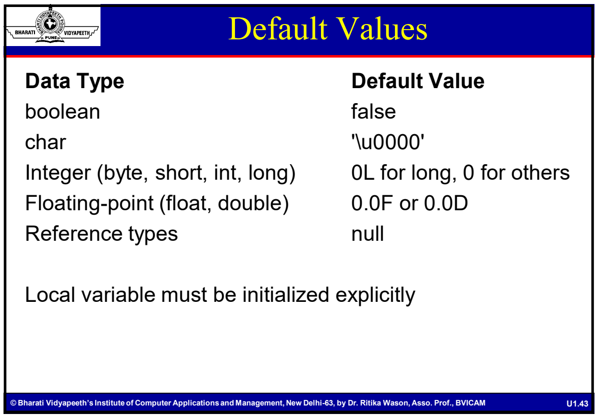
- Types of variables in JAVA
  - **Instance Variables (Non-Static Fields)**
  - **Class Variables (Static Fields)**
  - **Local Variables**
  - **Parameters**
- **Naming**
  - Case-sensitive
  - Subsequent characters may be letters, digits, dollar signs, or underscore characters



Reference Variables

- Store the reference value of an object
- Reference type can be a class/an array or an interface name

```
Pizza yummyPizza = new Pizza("Hot&Spicy");
// Declaration with initializer
```

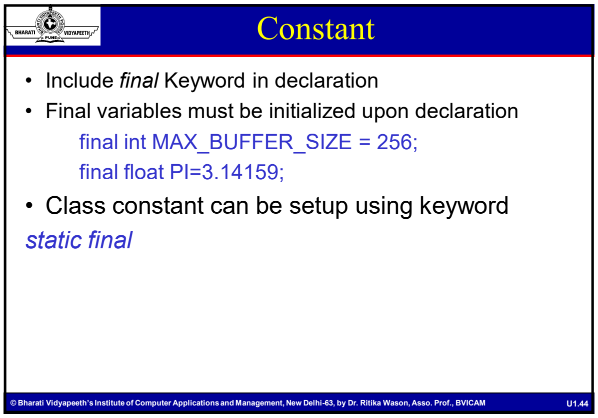


Default Values

Data Type	Default Value
boolean	false
char	'\u0000'
Integer (byte, short, int, long)	0L for long, 0 for others
Floating-point (float, double)	0.0F or 0.0D
Reference types	null

Local variable must be initialized explicitly

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.43



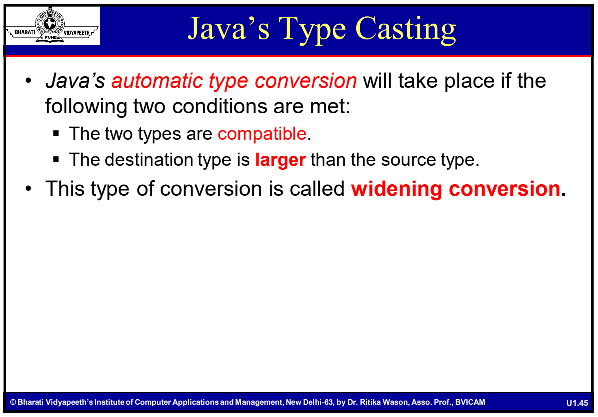
Constant

- Include *final* Keyword in declaration
- Final variables must be initialized upon declaration

```
final int MAX_BUFFER_SIZE = 256;
final float PI=3.14159;
```

- Class constant can be setup using keyword *static final*

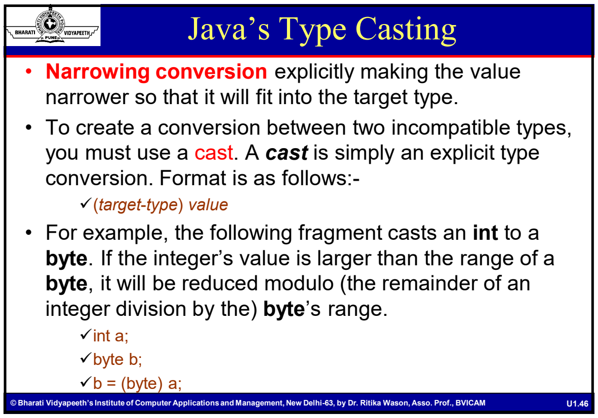
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.44



Java's Type Casting

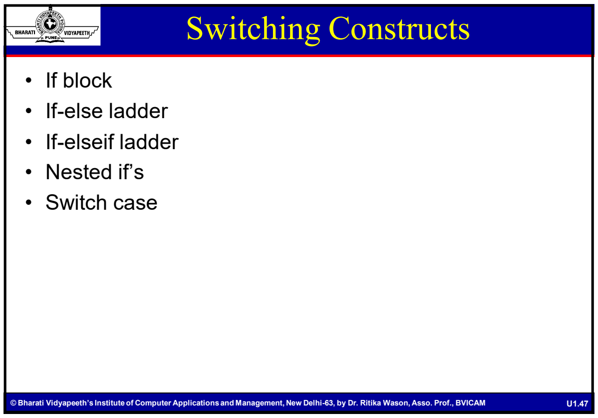
- *Java's automatic type conversion* will take place if the following two conditions are met:
  - The two types are compatible.
  - The destination type is **larger** than the source type.
- This type of conversion is called **widening conversion.**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.45



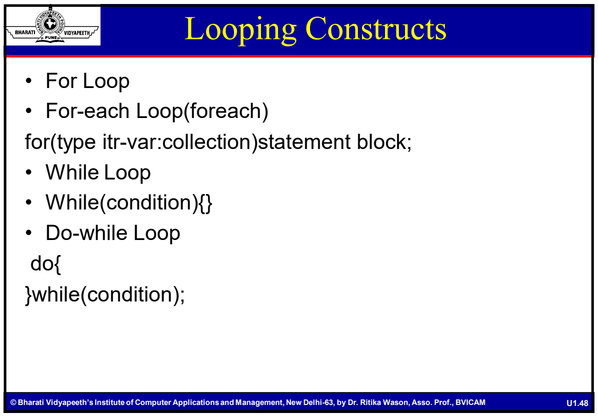
Java’s Type Casting

- **Narrowing conversion** explicitly making the value narrower so that it will fit into the target type.
- To create a conversion between two incompatible types, you must use a cast. A ***cast*** is simply an explicit type conversion. Format is as follows:-
  - ✓*(target-type) value*
- For example, the following fragment casts an **int** to a **byte**. If the integer’s value is larger than the range of a **byte**, it will be reduced modulo (the remainder of an integer division by the) **byte**’s range.
  - ✓int a;
  - ✓byte b;
  - ✓b = (byte) a;



Switching Constructs

- If block
- If-else ladder
- If-elseif ladder
- Nested if’s
- Switch case



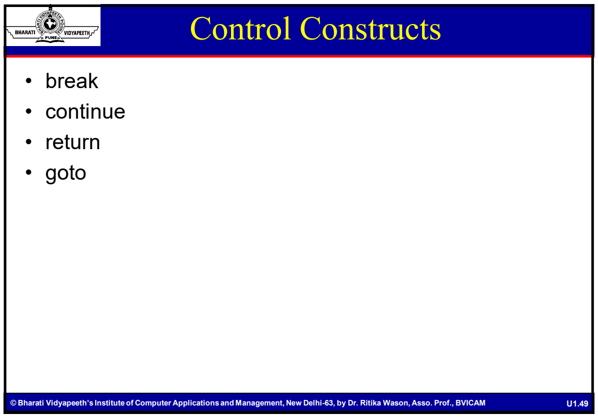
Looping Constructs

- For Loop
- For-each Loop(foreach)

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

- While Loop
- While(condition){}
- Do-while Loop

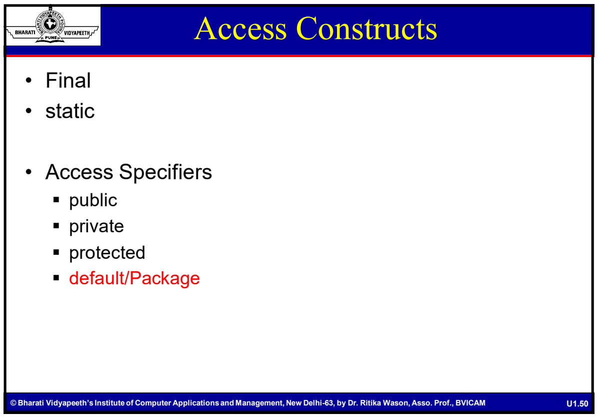
```
 do{
}while(condition);
```



Control Constructs

- break
- continue
- return
- goto

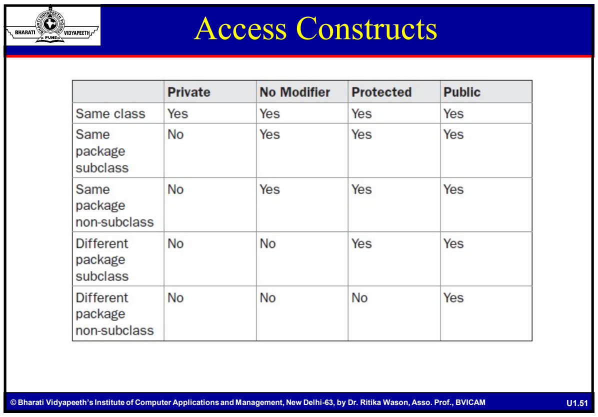
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.49



Access Constructs

- Final
- static
- Access Specifiers
  - public
  - private
  - protected
  - default/Package

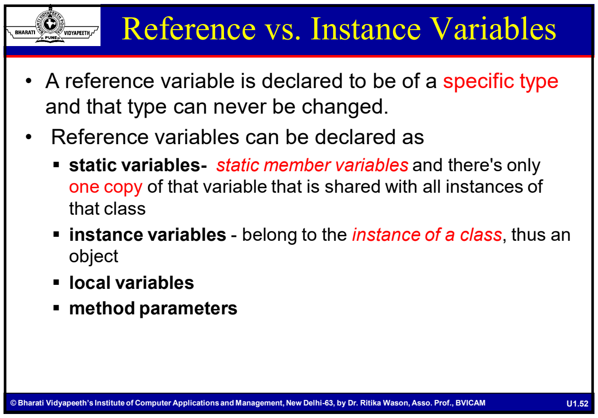
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.50



Access Constructs

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

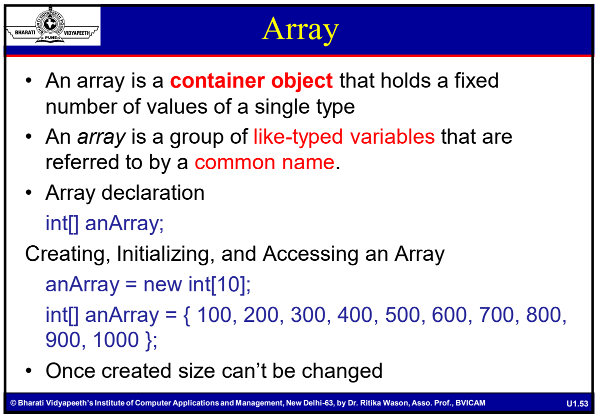
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.51



Reference vs. Instance Variables

- A reference variable is declared to be of a specific type and that type can never be changed.
- Reference variables can be declared as
  - **static variables-** *static member variables* and there's only one copy of that variable that is shared with all instances of that class
  - **instance variables** - belong to the *instance of a class*, thus an object
  - **local variables**
  - **method parameters**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.52



Array

- An array is a **container object** that holds a fixed number of values of a single type
- An *array* is a group of like-typed variables that are referred to by a common name.
- Array declaration

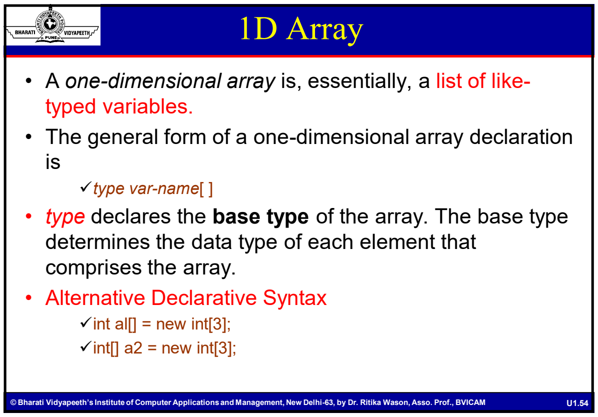
```
int[] anArray;
```

Creating, Initializing, and Accessing an Array

```
anArray = new int[10];
int[] anArray = { 100, 200, 300, 400, 500, 600, 700, 800,
900, 1000 };
```

- Once created size can't be changed

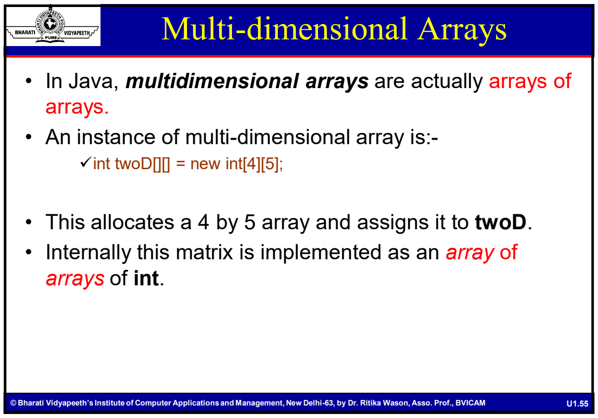
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.53



1D Array

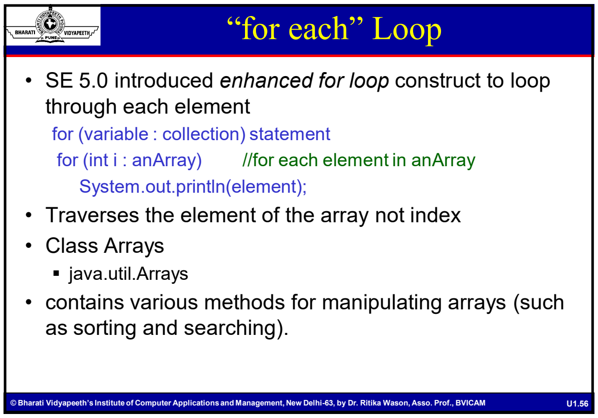
- A *one-dimensional array* is, essentially, a list of like-typed variables.
- The general form of a one-dimensional array declaration is
  - ✓*type var-name*[ ]
- *type* declares the **base type** of the array. The base type determines the data type of each element that comprises the array.
- Alternative Declarative Syntax
  - ✓int al[] = new int[3];
  - ✓int[] a2 = new int[3];

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.54



Multi-dimensional Arrays

- In Java, ***multidimensional arrays*** are actually arrays of arrays.
- An instance of multi-dimensional array is:-
  - ✓int twoD[][] = new int[4][5];
- This allocates a 4 by 5 array and assigns it to **twoD**.
- Internally this matrix is implemented as an *array of arrays* of **int**.

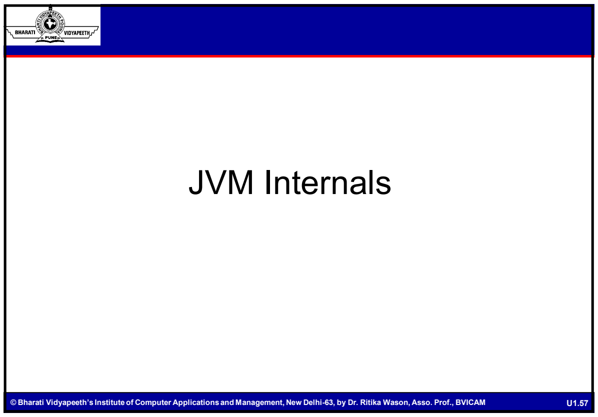


"for each" Loop

- SE 5.0 introduced *enhanced for loop* construct to loop through each element

```
 for (variable : collection) statement
  for (int i : anArray)       //for each element in anArray
      System.out.println(element);
```

- Traverses the element of the array not index
- Class Arrays
  - java.util.Arrays
- contains various methods for manipulating arrays (such as sorting and searching).

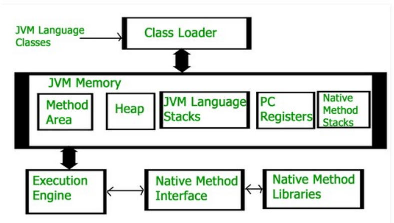


JVM Internals

JVM Architecture

- JVM(Java Virtual Machine) acts as a run-time engine to run Java applications.
- JVM is the one that actually calls the **main** method present in a java code.
- JVM is a part of JRE(Java Runtime Environment).
- When we compile a *.java* file, *.class* files(contains byte-code) with the same class names present in *.java* file are generated by the Java compiler. This *.class* file goes into various steps when we run it. These steps together describe the whole JVM.

JVM Architecture



JVM Language Classes

Class Loader

JVM Memory

Method Area

Heap

JVM Language Stacks

PC Registers

Native Method Stacks

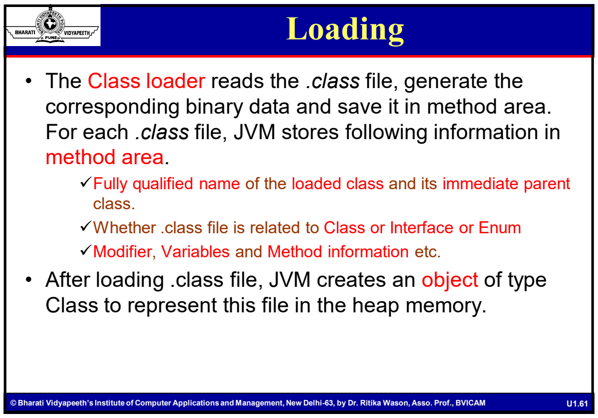
Execution Engine

Native Method Interface

Native Method Libraries

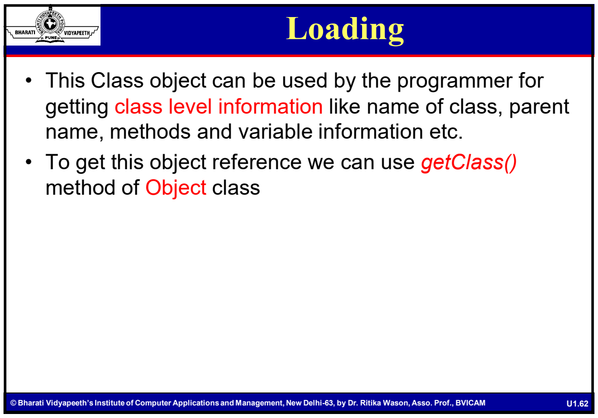
Class Loader Subsystem

- It is mainly responsible for three activities.
  - ✓Loading
  - ✓Linking
  - ✓Initialization



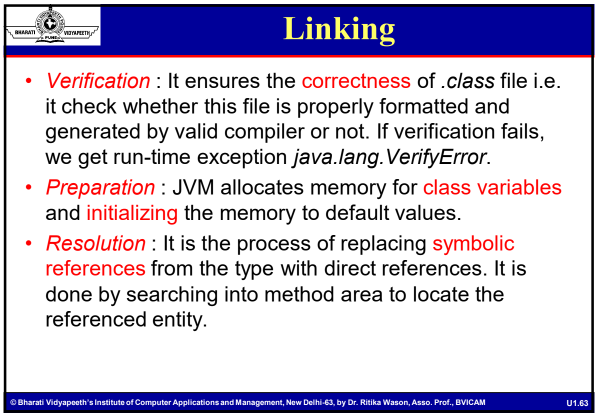
Loading

- The Class loader reads the *.class* file, generate the corresponding binary data and save it in method area. For each *.class* file, JVM stores following information in method area.
    - ✓Fully qualified name of the loaded class and its immediate parent class.
    - ✓Whether .class file is related to Class or Interface or Enum
    - ✓Modifier, Variables and Method information etc.
- After loading .class file, JVM creates an object of type Class to represent this file in the heap memory.



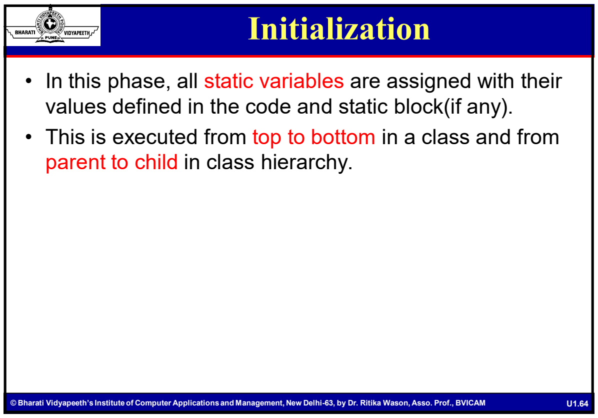
Loading

- This Class object can be used by the programmer for getting class level information like name of class, parent name, methods and variable information etc.
- To get this object reference we can use *getClass()* method of Object class



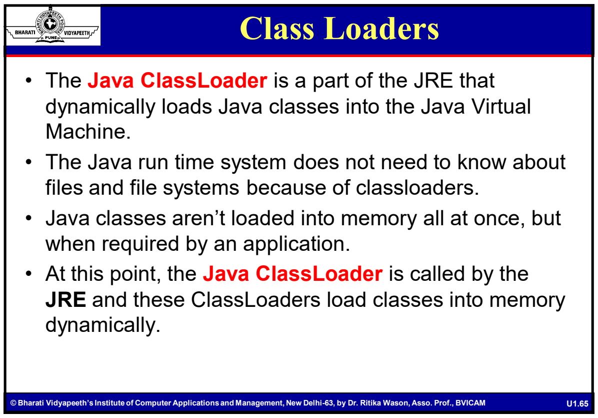
Linking

- *Verification* : It ensures the correctness of *.class* file i.e. it check whether this file is properly formatted and generated by valid compiler or not. If verification fails, we get run-time exception *java.lang.VerifyError*.
- *Preparation* : JVM allocates memory for class variables and initializing the memory to default values.
- *Resolution* : It is the process of replacing symbolic references from the type with direct references. It is done by searching into method area to locate the referenced entity.



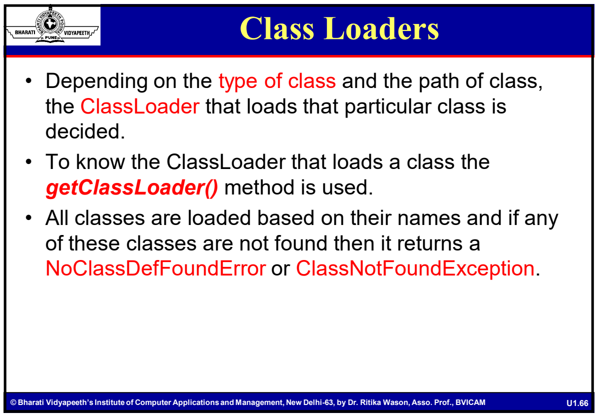
Initialization

- In this phase, all static variables are assigned with their values defined in the code and static block(if any).
- This is executed from top to bottom in a class and from parent to child in class hierarchy.



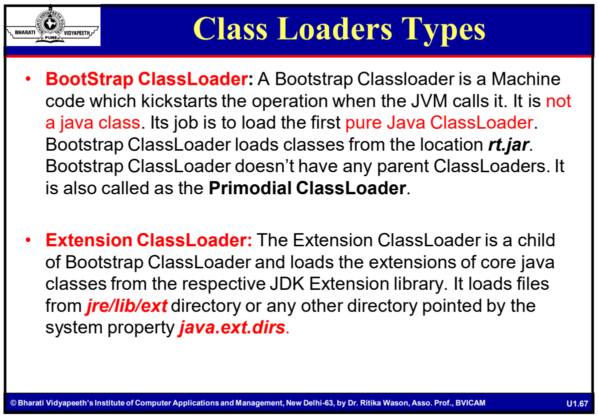
Class Loaders

- The **Java ClassLoader** is a part of the JRE that dynamically loads Java classes into the Java Virtual Machine.
- The Java run time system does not need to know about files and file systems because of classloaders.
- Java classes aren't loaded into memory all at once, but when required by an application.
- At this point, the **Java ClassLoader** is called by the **JRE** and these ClassLoaders load classes into memory dynamically.



Class Loaders

- Depending on the type of class and the path of class, the ClassLoader that loads that particular class is decided.
- To know the ClassLoader that loads a class the ***getClassLoader()*** method is used.
- All classes are loaded based on their names and if any of these classes are not found then it returns a NoClassDefFoundError or ClassNotFoundException.

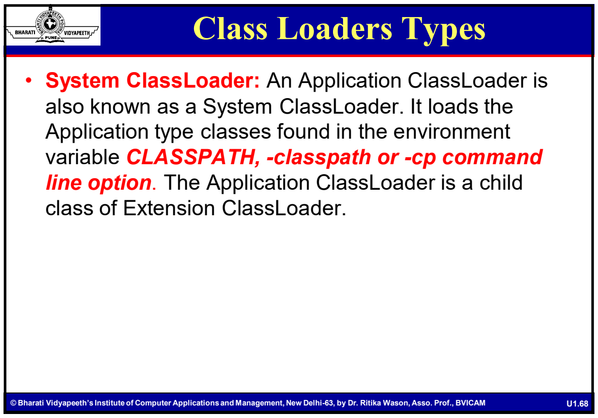


Class Loaders Types

- **BootStrap ClassLoader:** A Bootstrap Classloader is a Machine code which kickstarts the operation when the JVM calls it. It is not a java class. Its job is to load the first pure Java ClassLoader. Bootstrap ClassLoader loads classes from the location ***rt.jar***. Bootstrap ClassLoader doesn't have any parent ClassLoaders. It is also called as the **Primodial ClassLoader**.

- **Extension ClassLoader:** The Extension ClassLoader is a child of Bootstrap ClassLoader and loads the extensions of core java classes from the respective JDK Extension library. It loads files from ***jre/lib/ext*** directory or any other directory pointed by the system property ***java.ext.dirs***.

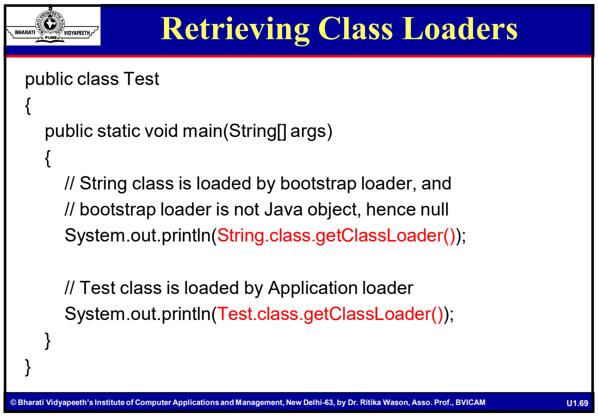
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.67



Class Loaders Types

- **System ClassLoader:** An Application ClassLoader is also known as a System ClassLoader. It loads the Application type classes found in the environment variable ***CLASSPATH, -classpath or -cp command line option***. The Application ClassLoader is a child class of Extension ClassLoader.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.68

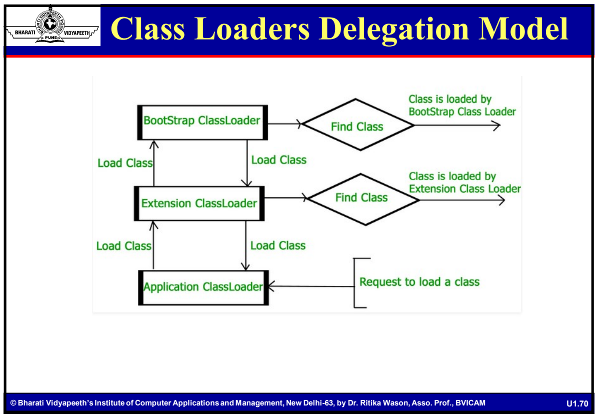


Retrieving Class Loaders

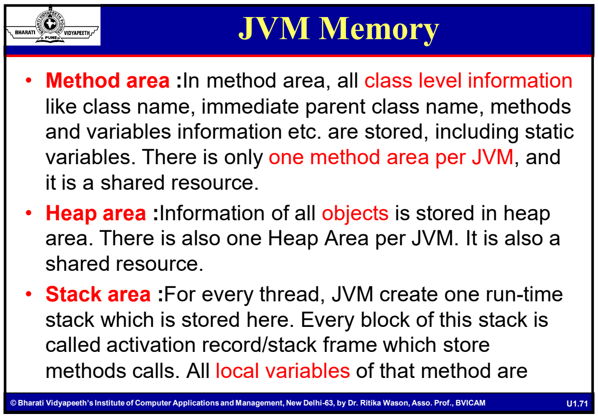
```
public class Test
{
    public static void main(String[] args)
    {
        // String class is loaded by bootstrap loader, and
        // bootstrap loader is not Java object, hence null
        System.out.println(String.class.getClassLoader());

        // Test class is loaded by Application loader
        System.out.println(Test.class.getClassLoader());
    }
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.69

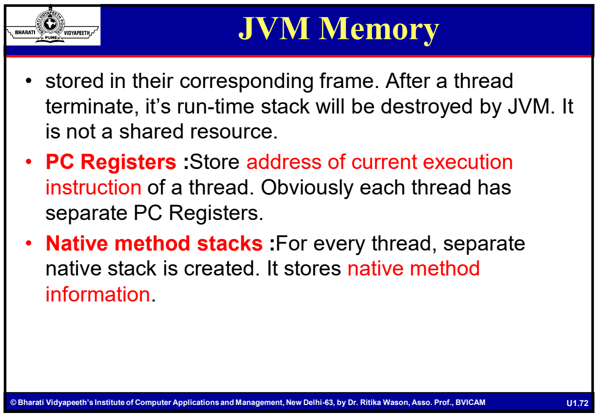


Class Loaders Delegation Model



JVM Memory

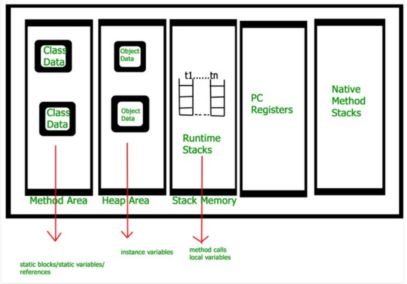
- **Method area** :In method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource.
- **Heap area** :Information of all objects is stored in heap area. There is also one Heap Area per JVM. It is also a shared resource.
- **Stack area** :For every thread, JVM create one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which store methods calls. All local variables of that method are



JVM Memory

- stored in their corresponding frame. After a thread terminate, it's run-time stack will be destroyed by JVM. It is not a shared resource.
- **PC Registers** :Store address of current execution instruction of a thread. Obviously each thread has separate PC Registers.
- **Native method stacks** :For every thread, separate native stack is created. It stores native method information.

JVM Memory



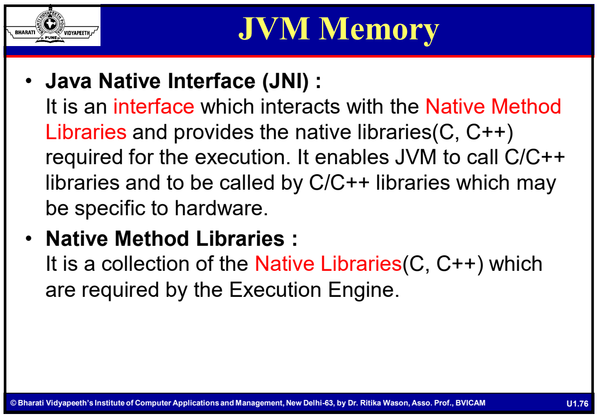
Execution Engine

- Execution engine execute the *.class* (bytecode). It reads the byte-code line by line, use data and information present in various memory area and execute instructions. It can be classified in three parts :-
- *Interpreter* : It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.
- *Just-In-Time Compiler(JIT)* : It is used to increase efficiency of interpreter.It compiles the entire bytecode and changes it to native code so whenever interpreter see repeated method

JVM Memory

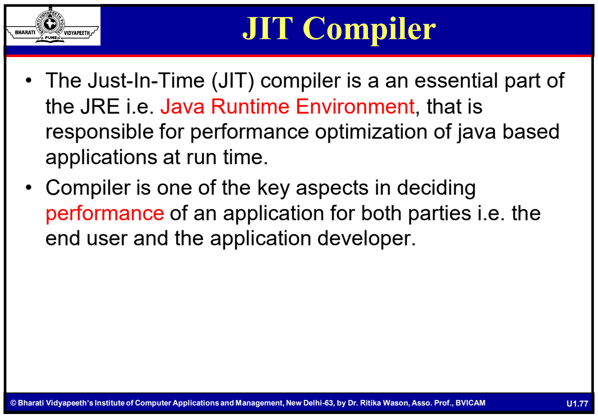
calls,JIT provide direct native code for that part so re-interpretation is not required,thus efficiency is improved.

- *Garbage Collector* : It destroy un-referenced objects.For more on Garbage Collector,refer Garbage Collector.



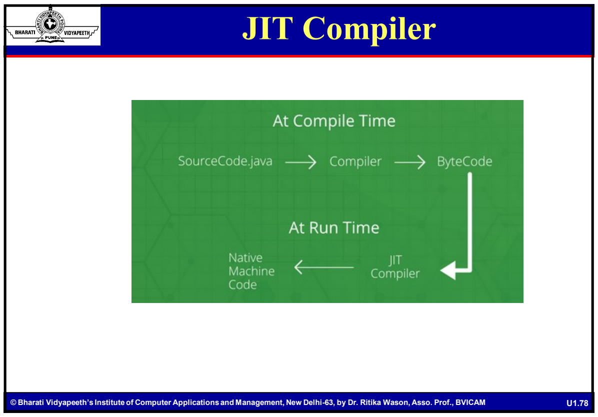
JVM Memory

- **Java Native Interface (JNI) :**
  It is an interface which interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution. It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.
- **Native Method Libraries :**
  It is a collection of the Native Libraries(C, C++) which are required by the Execution Engine.

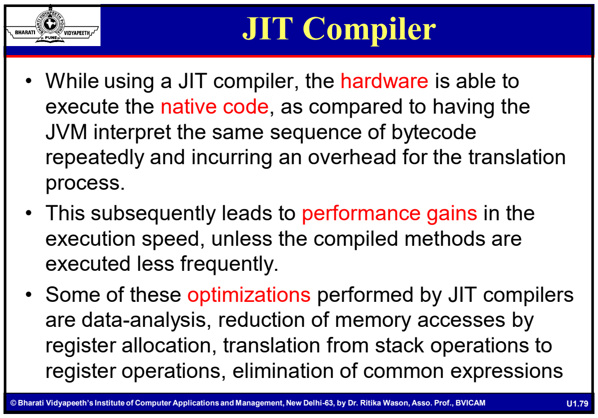


JIT Compiler

- The Just-In-Time (JIT) compiler is a an essential part of the JRE i.e. Java Runtime Environment, that is responsible for performance optimization of java based applications at run time.
- Compiler is one of the key aspects in deciding performance of an application for both parties i.e. the end user and the application developer.

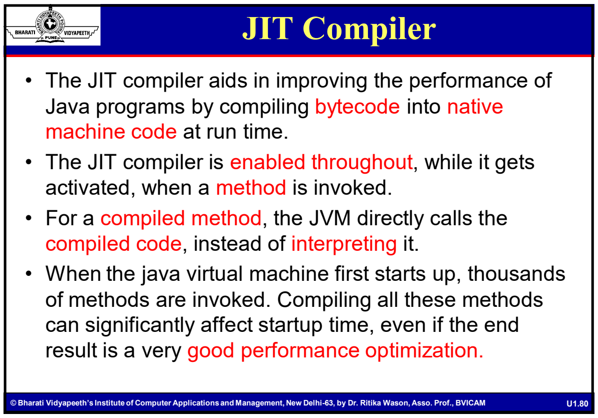


JIT Compiler



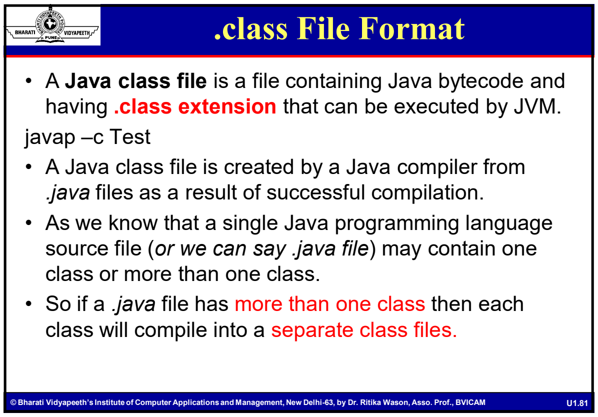
JIT Compiler

- While using a JIT compiler, the hardware is able to execute the native code, as compared to having the JVM interpret the same sequence of bytecode repeatedly and incurring an overhead for the translation process.
- This subsequently leads to performance gains in the execution speed, unless the compiled methods are executed less frequently.
- Some of these optimizations performed by JIT compilers are data-analysis, reduction of memory accesses by register allocation, translation from stack operations to register operations, elimination of common expressions



JIT Compiler

- The JIT compiler aids in improving the performance of Java programs by compiling bytecode into native machine code at run time.
- The JIT compiler is enabled throughout, while it gets activated, when a method is invoked.
- For a compiled method, the JVM directly calls the compiled code, instead of interpreting it.
- When the java virtual machine first starts up, thousands of methods are invoked. Compiling all these methods can significantly affect startup time, even if the end result is a very good performance optimization.

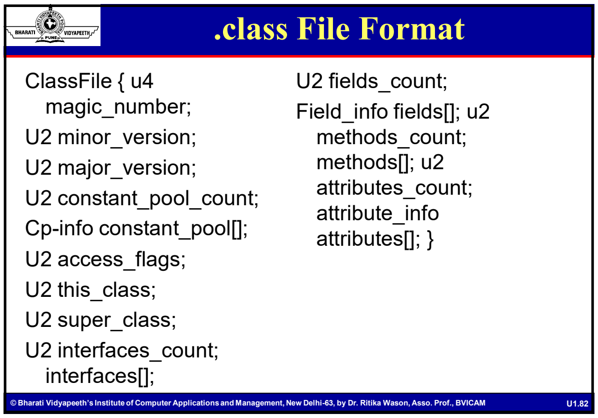


.class File Format

- A **Java class file** is a file containing Java bytecode and having **.class extension** that can be executed by JVM.

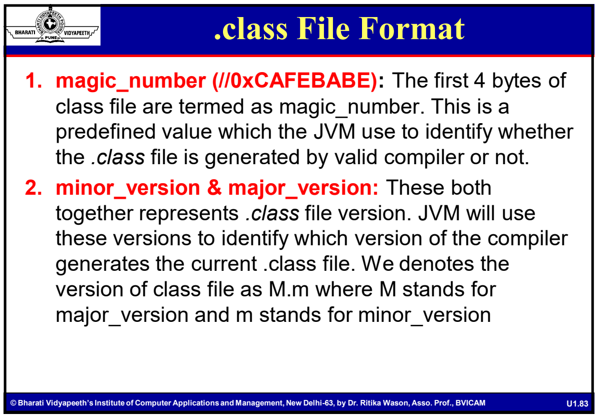
javap –c Test

- A Java class file is created by a Java compiler from *.java* files as a result of successful compilation.
- As we know that a single Java programming language source file (*or we can say .java file*) may contain one class or more than one class.
- So if a *.java* file has more than one class then each class will compile into a separate class files.



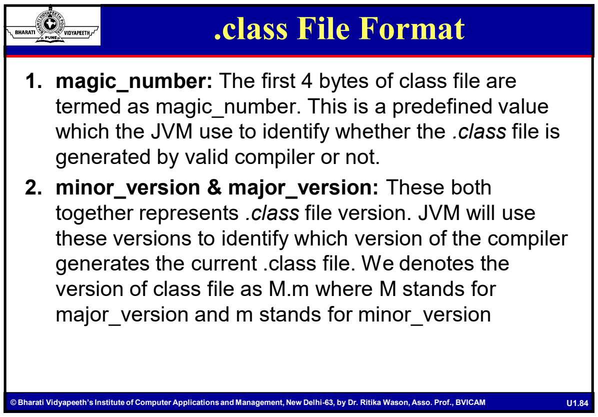
.class File Format

ClassFile { u4
magic_number;
U2 minor_version;
U2 major_version;
U2 constant_pool_count;
Cp-info constant_pool[];
U2 access_flags;
U2 this_class;
U2 super_class;
U2 interfaces_count;
interfaces[];
U2 fields_count;
Field_info fields[]; u2
methods_count;
methods[]; u2
attributes_count;
attribute_info
attributes[]; }



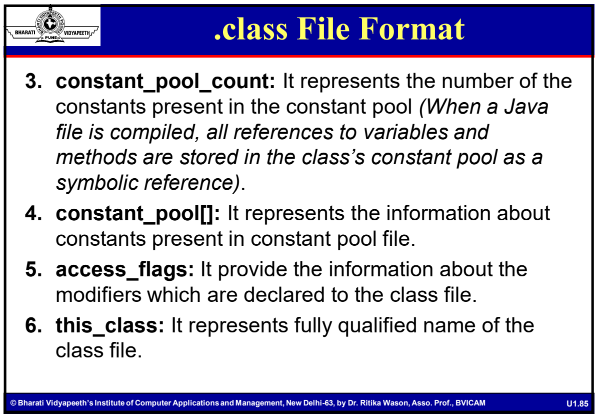
.class File Format

1. **magic_number (//0xCAFEBABE):** The first 4 bytes of class file are termed as magic_number. This is a predefined value which the JVM use to identify whether the *.class* file is generated by valid compiler or not.
2. **minor_version & major_version:** These both together represents *.class* file version. JVM will use these versions to identify which version of the compiler generates the current .class file. We denotes the version of class file as M.m where M stands for major_version and m stands for minor_version



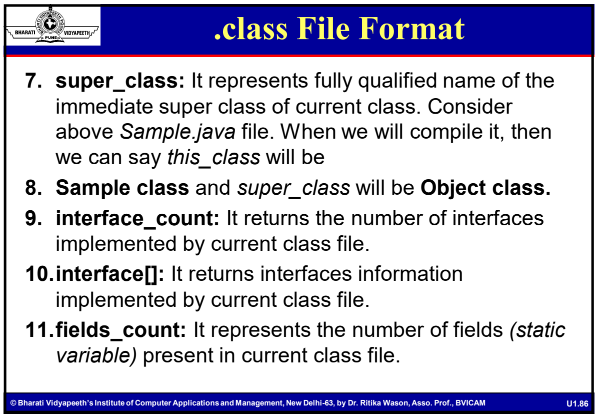
.class File Format

1. **magic_number:** The first 4 bytes of class file are termed as magic_number. This is a predefined value which the JVM use to identify whether the *.class* file is generated by valid compiler or not.
2. **minor_version & major_version:** These both together represents *.class* file version. JVM will use these versions to identify which version of the compiler generates the current .class file. We denotes the version of class file as M.m where M stands for major_version and m stands for minor_version



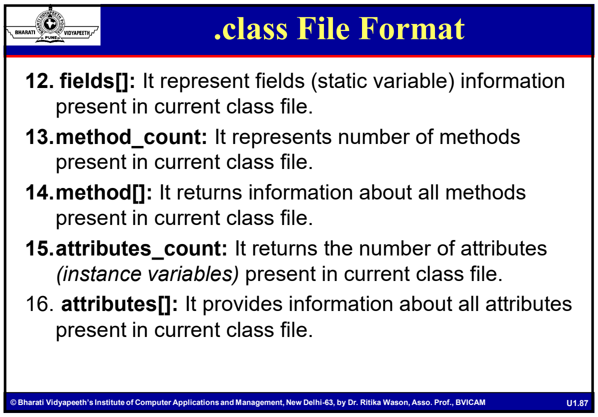
.class File Format

3. **constant_pool_count:** It represents the number of the constants present in the constant pool *(When a Java file is compiled, all references to variables and methods are stored in the class's constant pool as a symbolic reference)*.
4. **constant_pool[]:** It represents the information about constants present in constant pool file.
5. **access_flags:** It provide the information about the modifiers which are declared to the class file.
6. **this_class:** It represents fully qualified name of the class file.



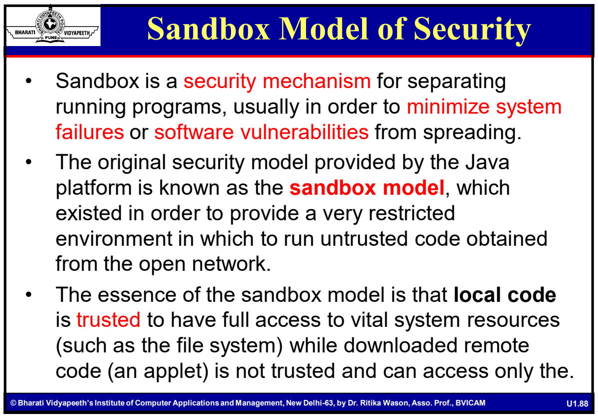
.class File Format

7. **super_class:** It represents fully qualified name of the immediate super class of current class. Consider above *Sample.java* file. When we will compile it, then we can say *this_class* will be
8. **Sample class** and *super_class* will be **Object class.**
9. **interface_count:** It returns the number of interfaces implemented by current class file.
10. **interface[]:** It returns interfaces information implemented by current class file.
11. **fields_count:** It represents the number of fields *(static variable)* present in current class file.



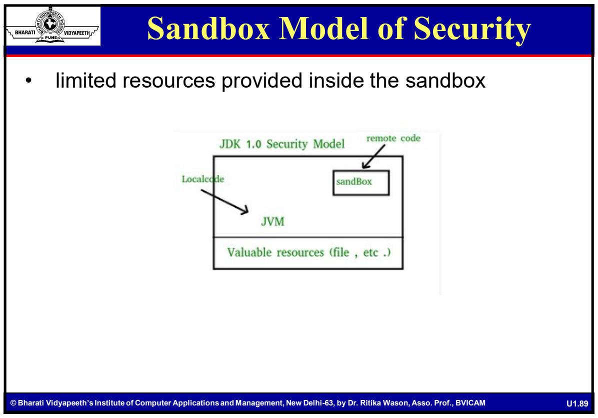
.class File Format

12. **fields[]:** It represent fields (static variable) information present in current class file.
13. **method_count:** It represents number of methods present in current class file.
14. **method[]:** It returns information about all methods present in current class file.
15. **attributes_count:** It returns the number of attributes *(instance variables)* present in current class file.
16. **attributes[]:** It provides information about all attributes present in current class file.



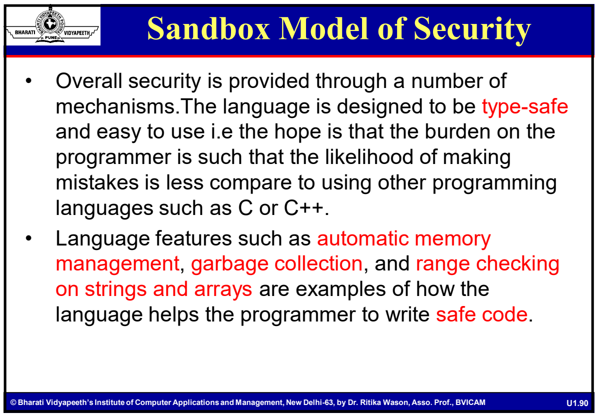
Sandbox Model of Security

- Sandbox is a security mechanism for separating running programs, usually in order to minimize system failures or software vulnerabilities from spreading.
- The original security model provided by the Java platform is known as the **sandbox model**, which existed in order to provide a very restricted environment in which to run untrusted code obtained from the open network.
- The essence of the sandbox model is that **local code** is trusted to have full access to vital system resources (such as the file system) while downloaded remote code (an applet) is not trusted and can access only the.



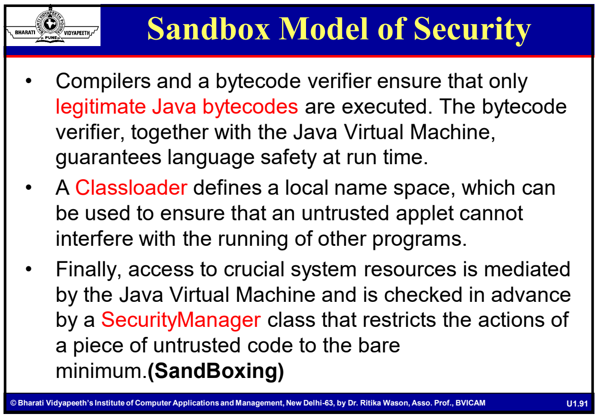
Sandbox Model of Security

- limited resources provided inside the sandbox



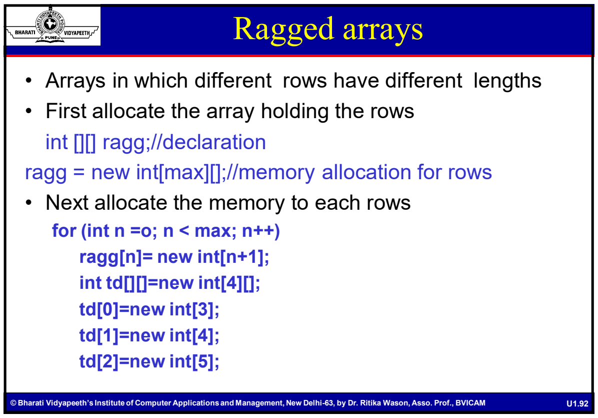
Sandbox Model of Security

- Overall security is provided through a number of mechanisms.The language is designed to be type-safe and easy to use i.e the hope is that the burden on the programmer is such that the likelihood of making mistakes is less compare to using other programming languages such as C or C++.
- Language features such as automatic memory management, garbage collection, and range checking on strings and arrays are examples of how the language helps the programmer to write safe code.



Sandbox Model of Security

- Compilers and a bytecode verifier ensure that only legitimate Java bytecodes are executed. The bytecode verifier, together with the Java Virtual Machine, guarantees language safety at run time.
- A Classloader defines a local name space, which can be used to ensure that an untrusted applet cannot interfere with the running of other programs.
- Finally, access to crucial system resources is mediated by the Java Virtual Machine and is checked in advance by a SecurityManager class that restricts the actions of a piece of untrusted code to the bare minimum.**(SandBoxing)**



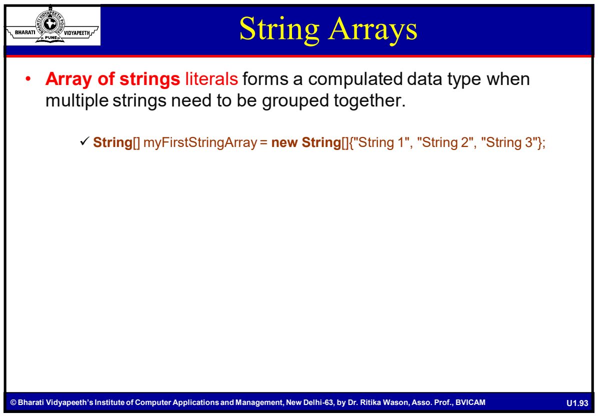
Ragged arrays

- Arrays in which different rows have different lengths
- First allocate the array holding the rows

```
  int [][] ragg;//declaration
ragg = new int[max][];//memory allocation for rows
```

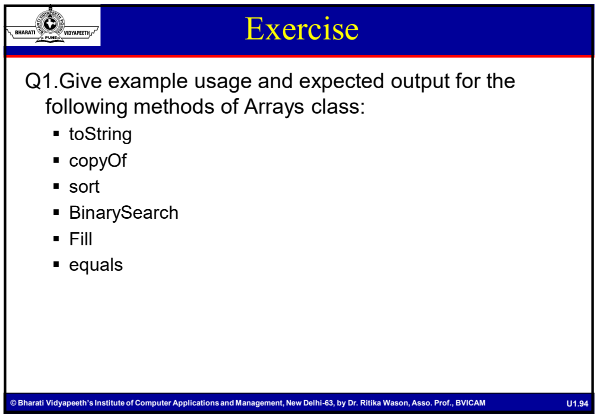
- Next allocate the memory to each rows

```
for (int n =o; n < max; n++)
    ragg[n]= new int[n+1];
    int td[][]=new int[4][];
    td[0]=new int[3];
    td[1]=new int[4];
    td[2]=new int[5];
```



String Arrays

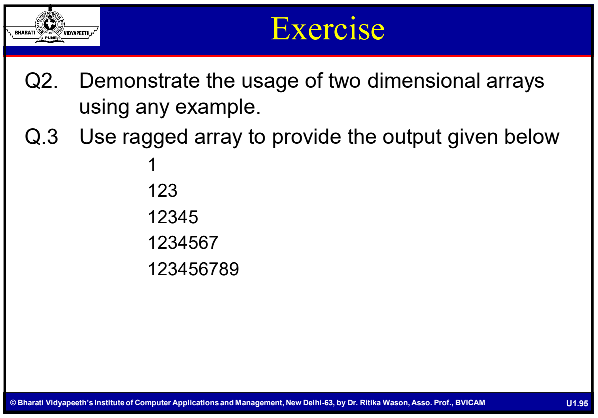
- **Array of strings** literals forms a computated data type when multiple strings need to be grouped together.
  - ✓ **String**[] myFirstStringArray = **new String**[]{"String 1", "String 2", "String 3"};



Exercise

Q1.Give example usage and expected output for the following methods of Arrays class:

- toString
- copyOf
- sort
- BinarySearch
- Fill
- equals

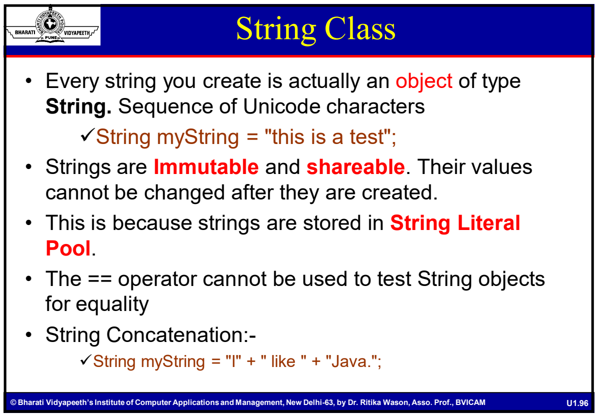


Exercise

Q2. Demonstrate the usage of two dimensional arrays using any example.

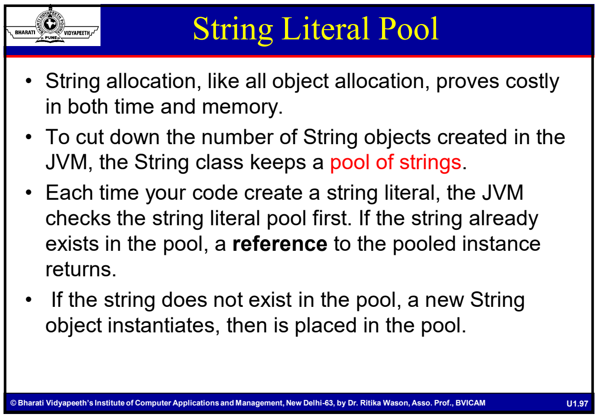
Q.3 Use ragged array to provide the output given below

```
1
123
12345
1234567
123456789
```



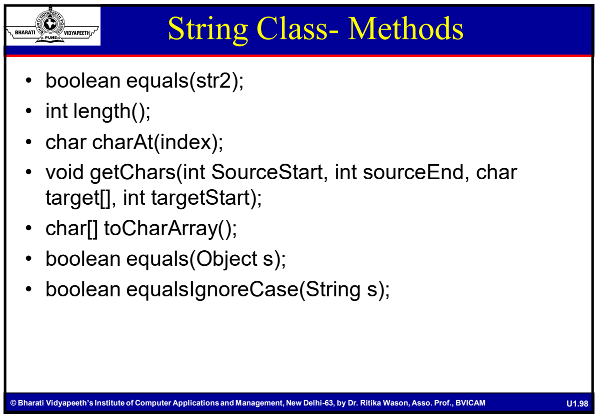
String Class

- Every string you create is actually an object of type **String.** Sequence of Unicode characters
  - ✓String myString = "this is a test";
- Strings are **Immutable** and **shareable**. Their values cannot be changed after they are created.
- This is because strings are stored in **String Literal Pool**.
- The == operator cannot be used to test String objects for equality
- String Concatenation:-
  - ✓String myString = "I" + " like " + "Java.";



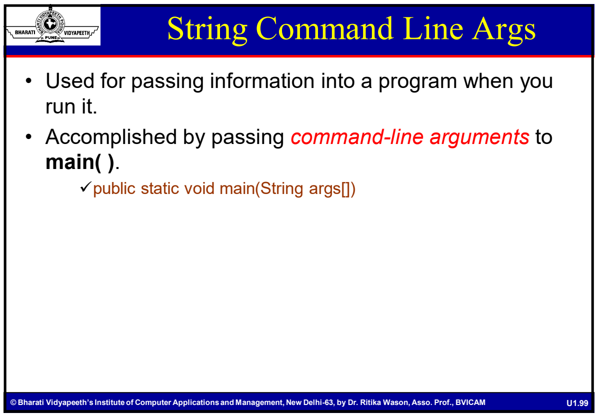
String Literal Pool

- String allocation, like all object allocation, proves costly in both time and memory.
- To cut down the number of String objects created in the JVM, the String class keeps a pool of strings.
- Each time your code create a string literal, the JVM checks the string literal pool first. If the string already exists in the pool, a **reference** to the pooled instance returns.
- If the string does not exist in the pool, a new String object instantiates, then is placed in the pool.



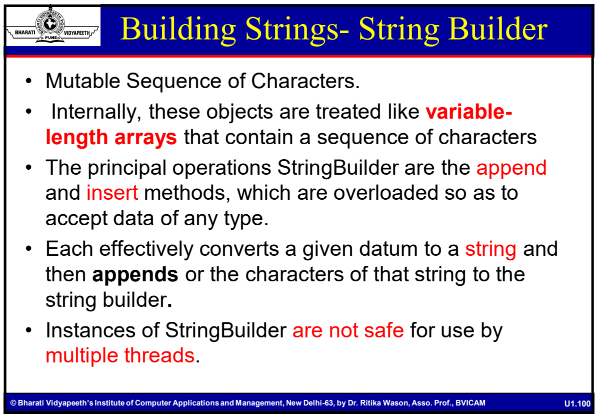
String Class- Methods

- boolean equals(str2);
- int length();
- char charAt(index);
- void getChars(int SourceStart, int sourceEnd, char target[], int targetStart);
- char[] toCharArray();
- boolean equals(Object s);
- boolean equalsIgnoreCase(String s);



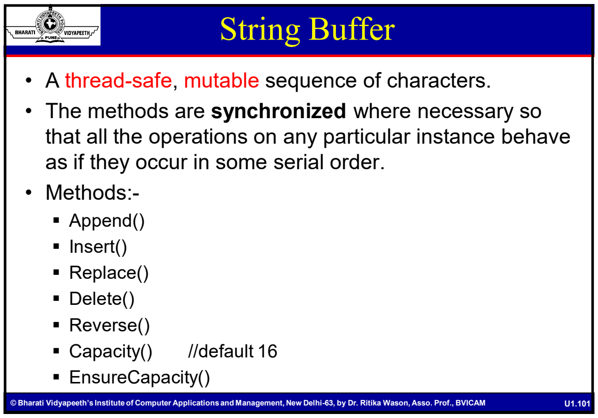
String Command Line Args

- Used for passing information into a program when you run it.
- Accomplished by passing *command-line arguments* to **main( )**.
  - ✓public static void main(String args[])



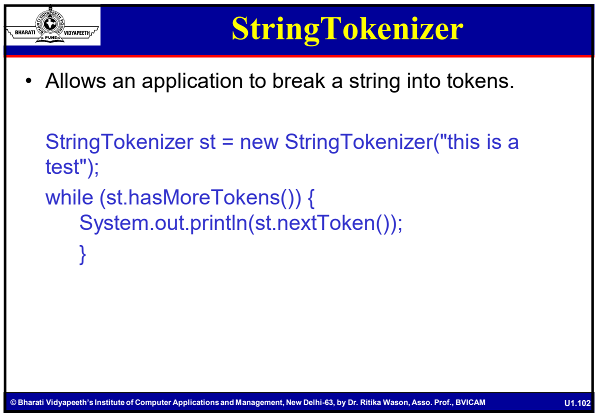
Building Strings- String Builder

- Mutable Sequence of Characters.
- Internally, these objects are treated like **variable-length arrays** that contain a sequence of characters
- The principal operations StringBuilder are the append and insert methods, which are overloaded so as to accept data of any type.
- Each effectively converts a given datum to a string and then **appends** or the characters of that string to the string builder**.**
- Instances of StringBuilder are not safe for use by multiple threads.



String Buffer

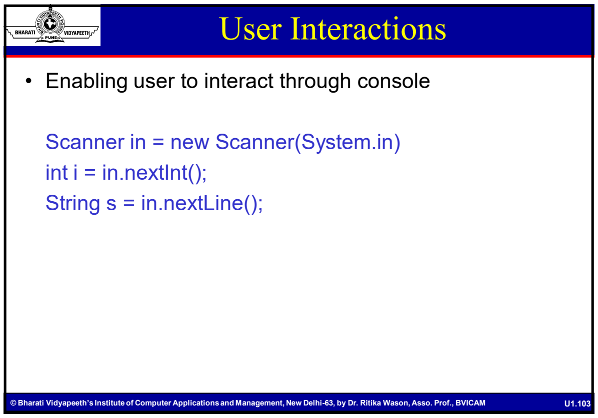
- A thread-safe, mutable sequence of characters.
- The methods are **synchronized** where necessary so that all the operations on any particular instance behave as if they occur in some serial order.
- Methods:-
  - Append()
  - Insert()
  - Replace()
  - Delete()
  - Reverse()
  - Capacity() //default 16
  - EnsureCapacity()



StringTokenizer

- Allows an application to break a string into tokens.

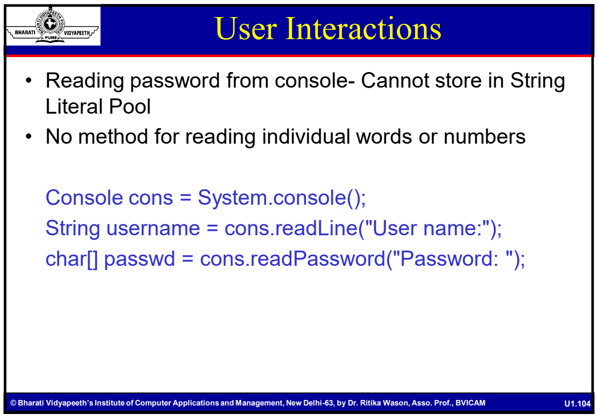
```
StringTokenizer st = new StringTokenizer("this is a
test");
while (st.hasMoreTokens()) {
     System.out.println(st.nextToken());
     }
```



User Interactions

- Enabling user to interact through console

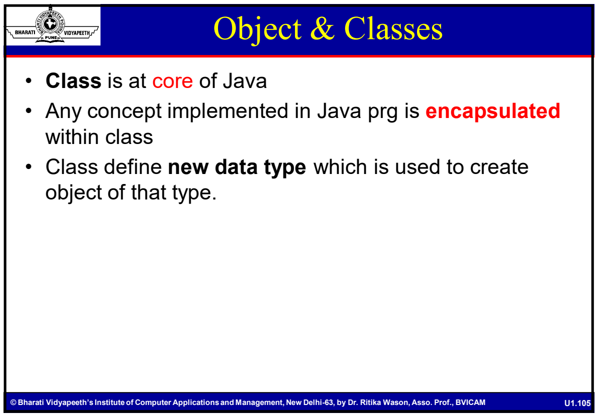
```
Scanner in = new Scanner(System.in)
int i = in.nextInt();
String s = in.nextLine();
```



User Interactions

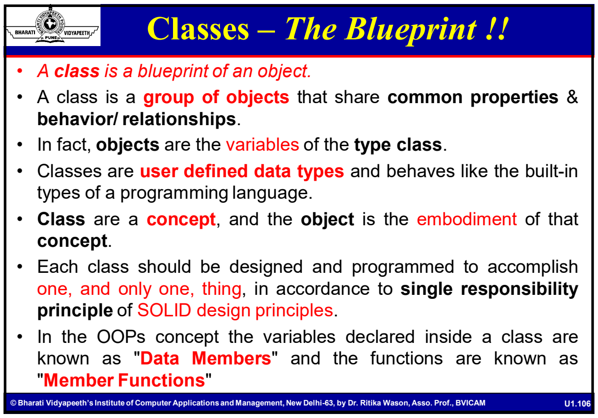
- Reading password from console- Cannot store in String Literal Pool
- No method for reading individual words or numbers

```
Console cons = System.console();
String username = cons.readLine("User name:");
char[] passwd = cons.readPassword("Password: ");
```



Object & Classes

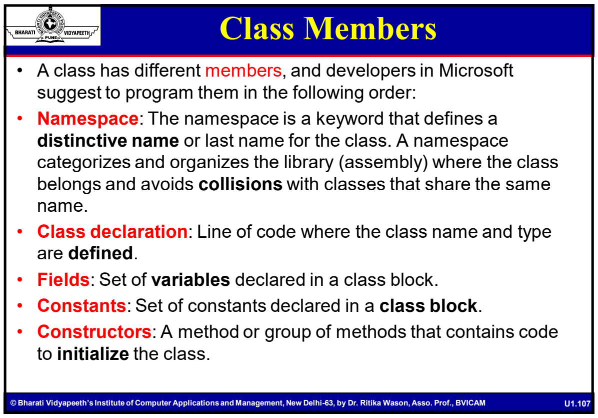
- **Class** is at core of Java
- Any concept implemented in Java prg is **encapsulated** within class
- Class define **new data type** which is used to create object of that type.



Classes – *The Blueprint !!*

- *A **class** is a blueprint of an object.*
- A class is a **group of objects** that share **common properties** & **behavior/ relationships**.
- In fact, **objects** are the variables of the **type class**.
- Classes are **user defined data types** and behaves like the built-in types of a programming language.
- **Class** are a **concept**, and the **object** is the embodiment of that **concept**.
- Each class should be designed and programmed to accomplish one, and only one, thing, in accordance to **single responsibility principle** of SOLID design principles.
- In the OOPs concept the variables declared inside a class are known as "**Data Members**" and the functions are known as "**Member Functions**"

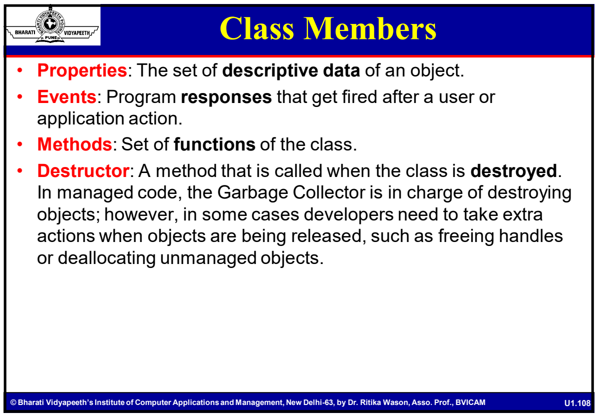
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.106



Class Members

- A class has different members, and developers in Microsoft suggest to program them in the following order:
- **Namespace**: The namespace is a keyword that defines a **distinctive name** or last name for the class. A namespace categorizes and organizes the library (assembly) where the class belongs and avoids **collisions** with classes that share the same name.
- **Class declaration**: Line of code where the class name and type are **defined**.
- **Fields**: Set of **variables** declared in a class block.
- **Constants**: Set of constants declared in a **class block**.
- **Constructors**: A method or group of methods that contains code to **initialize** the class.

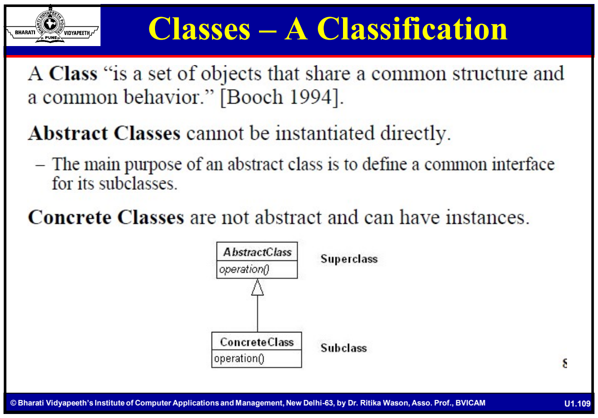
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.107



Class Members

- **Properties**: The set of **descriptive data** of an object.
- **Events**: Program **responses** that get fired after a user or application action.
- **Methods**: Set of **functions** of the class.
- **Destructor**: A method that is called when the class is **destroyed**. In managed code, the Garbage Collector is in charge of destroying objects; however, in some cases developers need to take extra actions when objects are being released, such as freeing handles or deallocating unmanaged objects.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.108



Classes – A Classification

A **Class** "is a set of objects that share a common structure and a common behavior." [Booch 1994].

Abstract Classes cannot be instantiated directly.

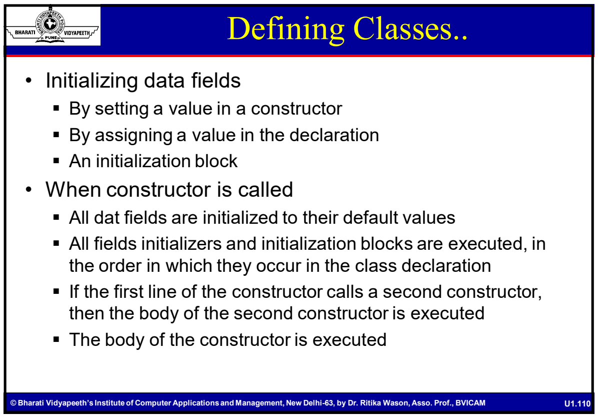
– The main purpose of an abstract class is to define a common interface for its subclasses.

Concrete Classes are not abstract and can have instances.

AbstractClass
operation()
Superclass

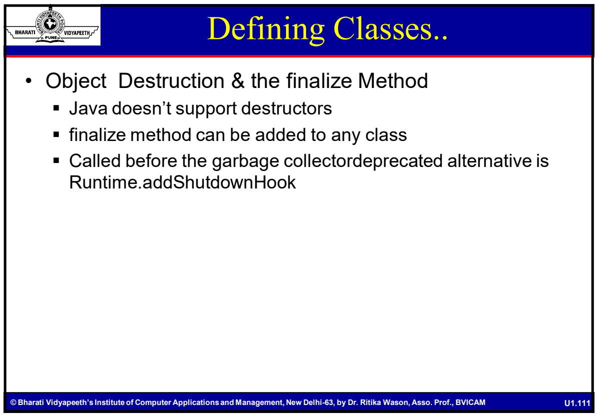
ConcreteClass
operation()
Subclass

8



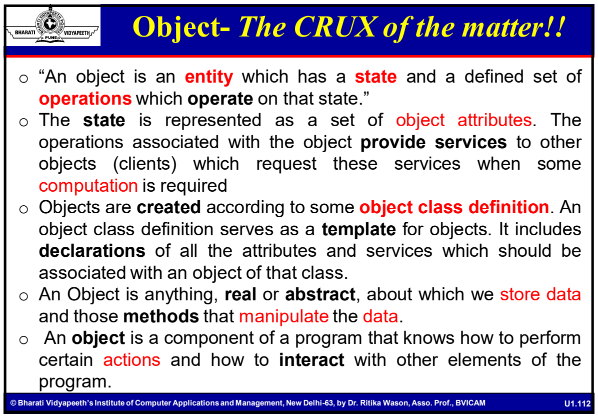
Defining Classes..

- Initializing data fields
  - By setting a value in a constructor
  - By assigning a value in the declaration
  - An initialization block
- When constructor is called
  - All dat fields are initialized to their default values
  - All fields initializers and initialization blocks are executed, in the order in which they occur in the class declaration
  - If the first line of the constructor calls a second constructor, then the body of the second constructor is executed
  - The body of the constructor is executed



Defining Classes..

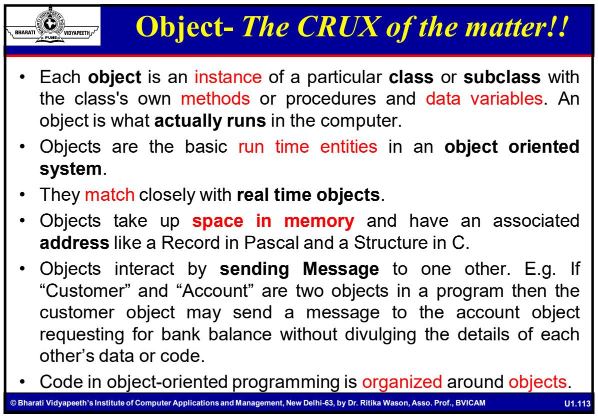
- Object Destruction & the finalize Method
  - Java doesn't support destructors
  - finalize method can be added to any class
  - Called before the garbage collectordeprecated alternative is Runtime.addShutdownHook



Object- *The CRUX of the matter!!*

- "An object is an **entity** which has a **state** and a defined set of **operations** which **operate** on that state."
- The **state** is represented as a set of object attributes. The operations associated with the object **provide services** to other objects (clients) which request these services when some computation is required
- Objects are **created** according to some **object class definition**. An object class definition serves as a **template** for objects. It includes **declarations** of all the attributes and services which should be associated with an object of that class.
- An Object is anything, **real** or **abstract**, about which we store data and those **methods** that manipulate the data.
- An **object** is a component of a program that knows how to perform certain actions and how to **interact** with other elements of the program.

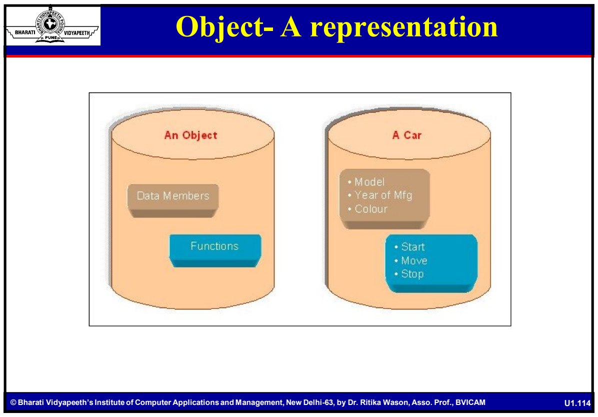
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.112



Object- *The CRUX of the matter!!*

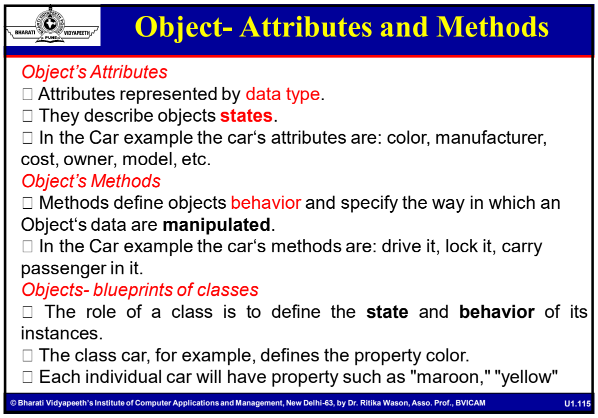
- Each **object** is an instance of a particular **class** or **subclass** with the class's own methods or procedures and data variables. An object is what **actually runs** in the computer.
- Objects are the basic run time entities in an **object oriented system**.
- They match closely with **real time objects**.
- Objects take up **space in memory** and have an associated **address** like a Record in Pascal and a Structure in C.
- Objects interact by **sending Message** to one other. E.g. If "Customer" and "Account" are two objects in a program then the customer object may send a message to the account object requesting for bank balance without divulging the details of each other's data or code.
- Code in object-oriented programming is organized around objects.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.113



Object- A representation

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.114



Object- Attributes and Methods

Object's Attributes

□ Attributes represented by data type.

□ They describe objects **states**.

□ In the Car example the car's attributes are: color, manufacturer, cost, owner, model, etc.

Object's Methods

□ Methods define objects behavior and specify the way in which an Object's data are **manipulated**.

□ In the Car example the car's methods are: drive it, lock it, carry passenger in it.

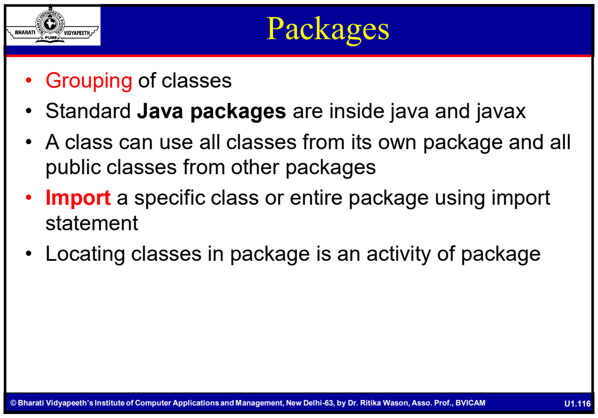
Objects- blueprints of classes

□ The role of a class is to define the **state** and **behavior** of its instances.

□ The class car, for example, defines the property color.

□ Each individual car will have property such as "maroon," "yellow"

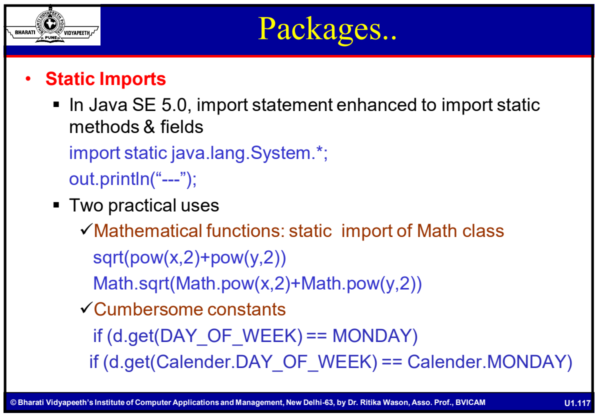
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.115



Packages

- Grouping of classes
- Standard **Java packages** are inside java and javax
- A class can use all classes from its own package and all public classes from other packages
- **Import** a specific class or entire package using import statement
- Locating classes in package is an activity of package

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.116



Packages..

- **Static Imports**
  - In Java SE 5.0, import statement enhanced to import static methods & fields

    import static java.lang.System.*;

    out.println("---");
  - Two practical uses
    - ✓Mathematical functions: static import of Math class

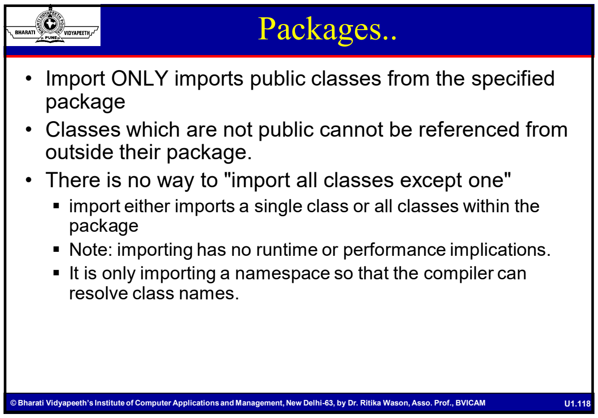
      sqrt(pow(x,2)+pow(y,2))

      Math.sqrt(Math.pow(x,2)+Math.pow(y,2))
    - ✓Cumbersome constants

      if (d.get(DAY_OF_WEEK) == MONDAY)

      if (d.get(Calender.DAY_OF_WEEK) == Calender.MONDAY)

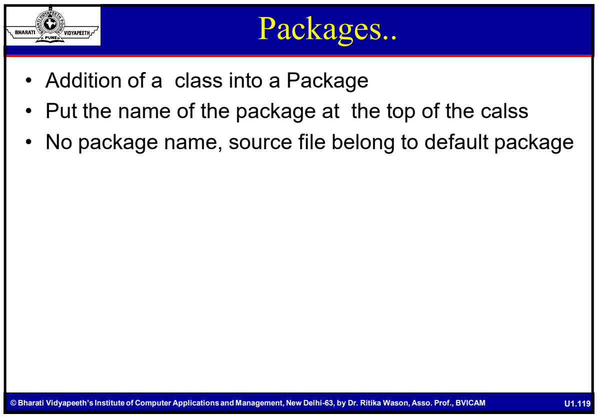
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.117



Packages..

- Import ONLY imports public classes from the specified package
- Classes which are not public cannot be referenced from outside their package.
- There is no way to "import all classes except one"
  - import either imports a single class or all classes within the package
  - Note: importing has no runtime or performance implications.
  - It is only importing a namespace so that the compiler can resolve class names.

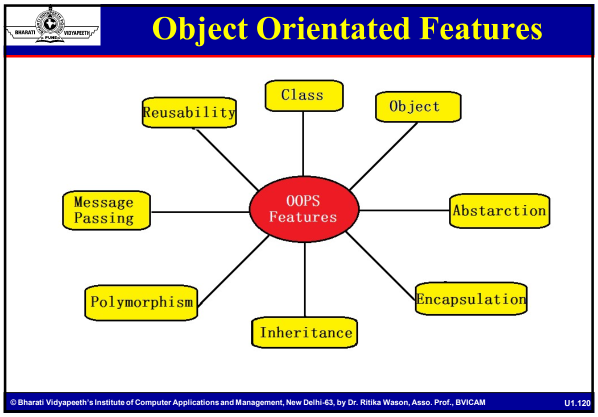
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.118



Packages..

- Addition of a class into a Package
- Put the name of the package at the top of the calss
- No package name, source file belong to default package

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.119



Object Orientated Features

OOPS Features

Reusability

Class

Object

Message Passing

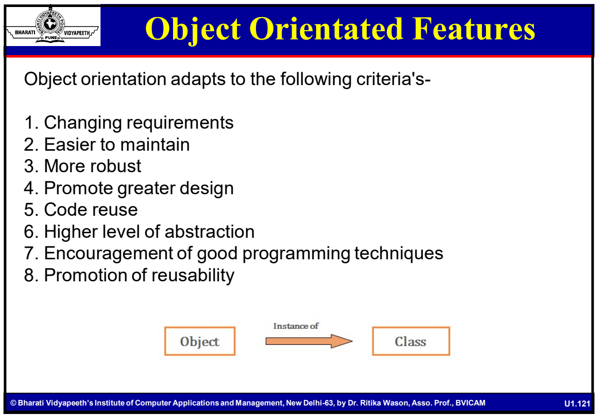
Abstarction

Polymorphism

Inheritance

Encapsulation

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.120



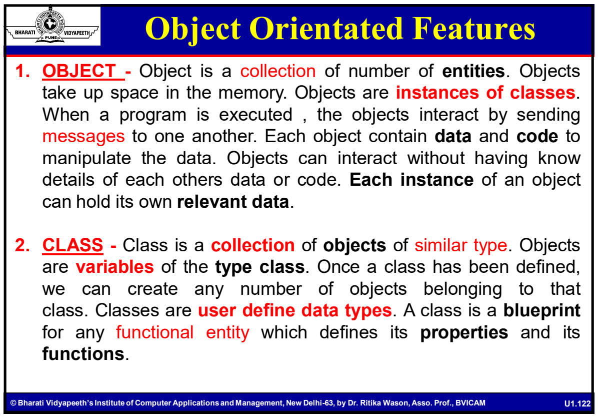
Object Orientated Features

Object orientation adapts to the following criteria's-

1. Changing requirements
2. Easier to maintain
3. More robust
4. Promote greater design
5. Code reuse
6. Higher level of abstraction
7. Encouragement of good programming techniques
8. Promotion of reusability

Object → Instance of → Class

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.121

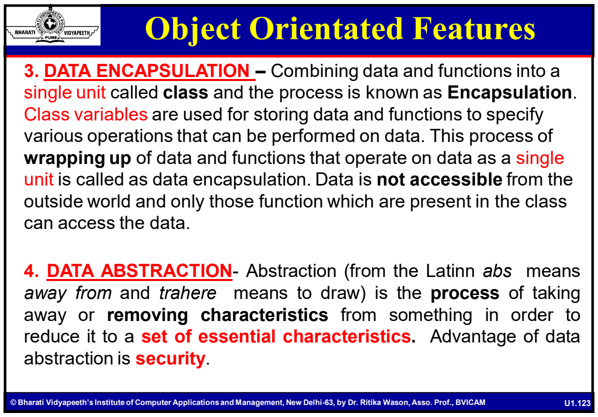


Object Orientated Features

1. **OBJECT** - Object is a collection of number of **entities**. Objects take up space in the memory. Objects are **instances of classes**. When a program is executed , the objects interact by sending messages to one another. Each object contain **data** and **code** to manipulate the data. Objects can interact without having know details of each others data or code. **Each instance** of an object can hold its own **relevant data**.

2. **CLASS** - Class is a **collection** of **objects** of similar type. Objects are **variables** of the **type class**. Once a class has been defined, we can create any number of objects belonging to that class. Classes are **user define data types**. A class is a **blueprint** for any functional entity which defines its **properties** and its **functions**.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.122

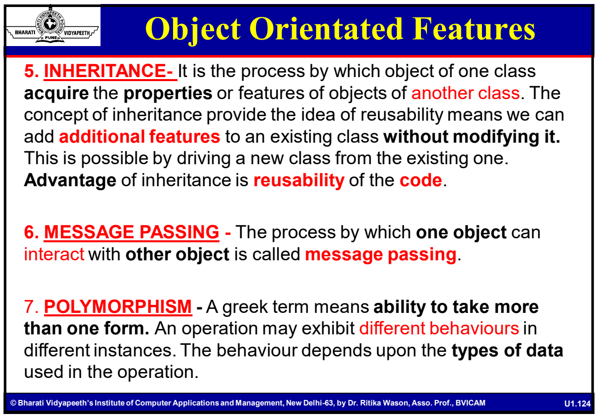


Object Orientated Features

3. DATA ENCAPSULATION – Combining data and functions into a single unit called **class** and the process is known as **Encapsulation**. Class variables are used for storing data and functions to specify various operations that can be performed on data. This process of **wrapping up** of data and functions that operate on data as a single unit is called as data encapsulation. Data is **not accessible** from the outside world and only those function which are present in the class can access the data.

4. DATA ABSTRACTION- Abstraction (from the Latinn *abs* means *away from* and *trahere* means to draw) is the **process** of taking away or **removing characteristics** from something in order to reduce it to a **set of essential characteristics**. Advantage of data abstraction is **security**.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.123

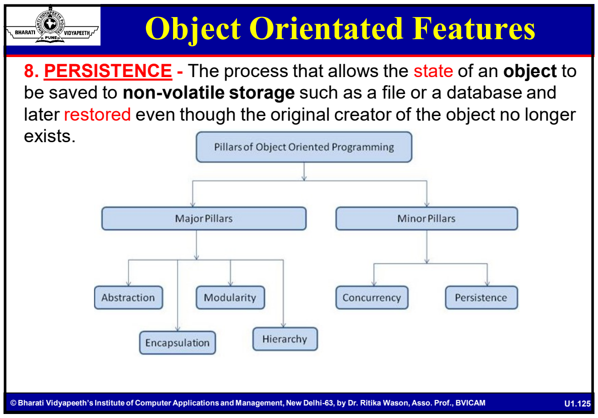


Object Orientated Features

5. INHERITANCE- It is the process by which object of one class **acquire** the **properties** or features of objects of another class. The concept of inheritance provide the idea of reusability means we can add **additional features** to an existing class **without modifying it.** This is possible by driving a new class from the existing one. **Advantage** of inheritance is **reusability** of the **code**.

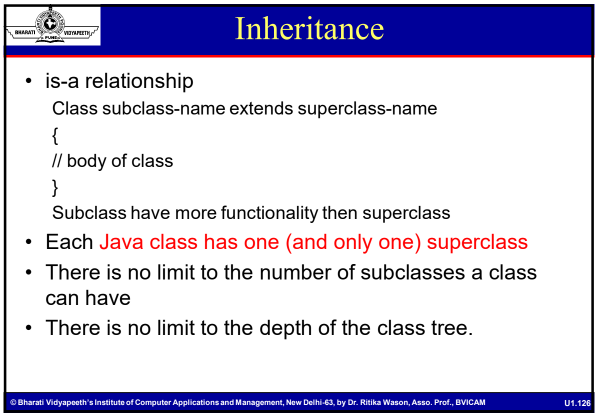
6. MESSAGE PASSING - The process by which **one object** can interact with **other object** is called **message passing**.

7. **POLYMORPHISM -** A greek term means **ability to take more than one form.** An operation may exhibit different behaviours in different instances. The behaviour depends upon the **types of data** used in the operation.



Object Orientated Features

8. PERSISTENCE - The process that allows the state of an **object** to be saved to **non-volatile storage** such as a file or a database and later restored even though the original creator of the object no longer exists.



Inheritance

- is-a relationship

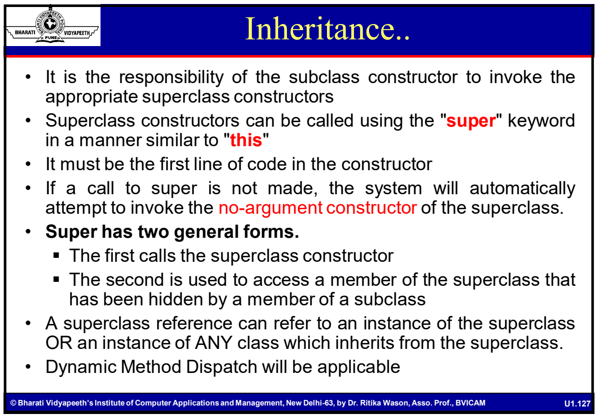
  Class subclass-name extends superclass-name

  {

  // body of class

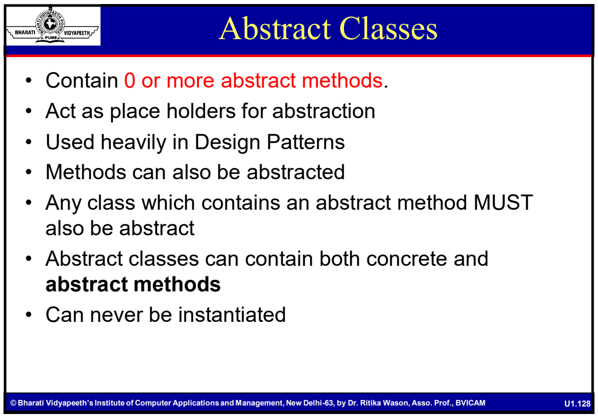
  }

  Subclass have more functionality then superclass
- Each Java class has one (and only one) superclass
- There is no limit to the number of subclasses a class can have
- There is no limit to the depth of the class tree.



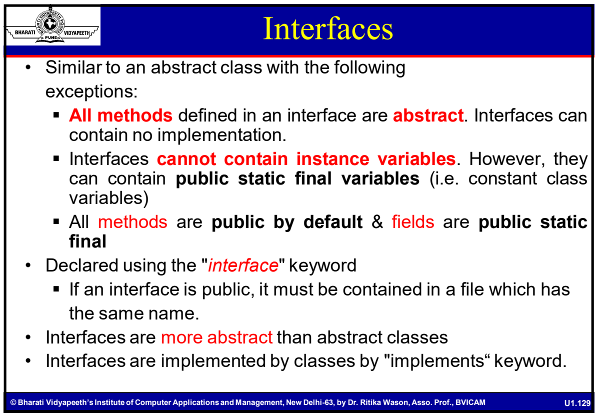
Inheritance..

- It is the responsibility of the subclass constructor to invoke the appropriate superclass constructors
- Superclass constructors can be called using the "**super**" keyword in a manner similar to "**this**"
- It must be the first line of code in the constructor
- If a call to super is not made, the system will automatically attempt to invoke the no-argument constructor of the superclass.
- **Super has two general forms.**
  - The first calls the superclass constructor
  - The second is used to access a member of the superclass that has been hidden by a member of a subclass
- A superclass reference can refer to an instance of the superclass OR an instance of ANY class which inherits from the superclass.
- Dynamic Method Dispatch will be applicable



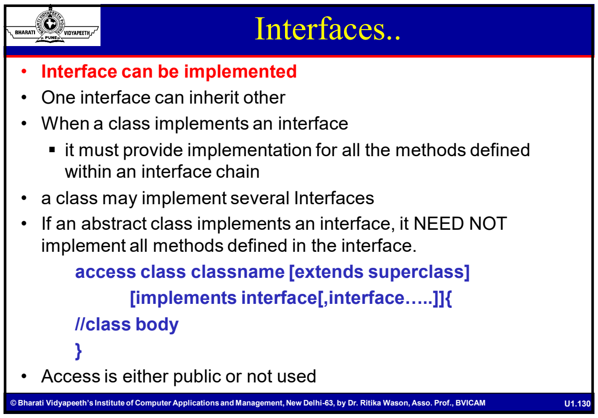
Abstract Classes

- Contain 0 or more abstract methods.
- Act as place holders for abstraction
- Used heavily in Design Patterns
- Methods can also be abstracted
- Any class which contains an abstract method MUST also be abstract
- Abstract classes can contain both concrete and **abstract methods**
- Can never be instantiated



Interfaces

- Similar to an abstract class with the following exceptions:
  - **All methods** defined in an interface are **abstract**. Interfaces can contain no implementation.
  - Interfaces **cannot contain instance variables**. However, they can contain **public static final variables** (i.e. constant class variables)
  - All methods are **public by default** & fields are **public static final**
- Declared using the "*interface*" keyword
  - If an interface is public, it must be contained in a file which has the same name.
- Interfaces are more abstract than abstract classes
- Interfaces are implemented by classes by "implements" keyword.

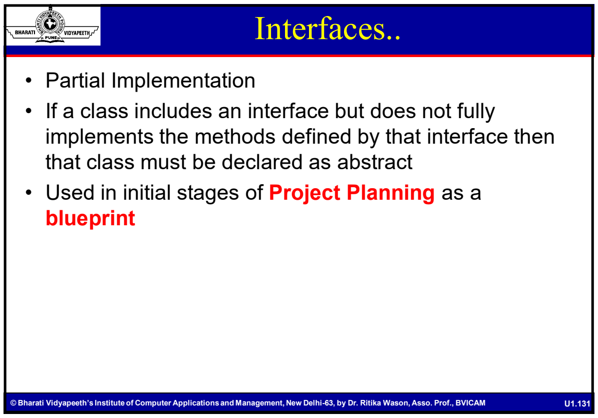


Interfaces..

- **Interface can be implemented**
- One interface can inherit other
- When a class implements an interface
  - it must provide implementation for all the methods defined within an interface chain
- a class may implement several Interfaces
- If an abstract class implements an interface, it NEED NOT implement all methods defined in the interface.

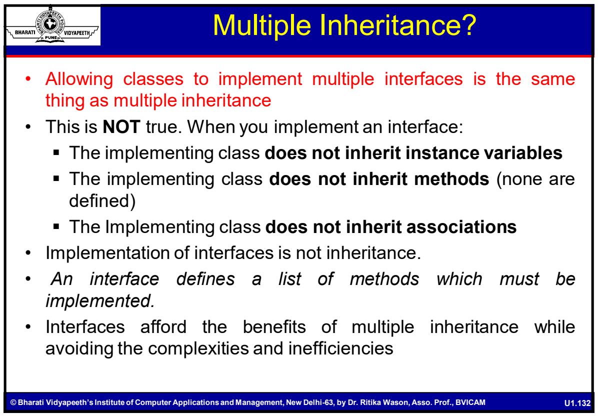
```
access class classname [extends superclass]
        [implements interface[,interface…..]]{
//class body
}
```

- Access is either public or not used



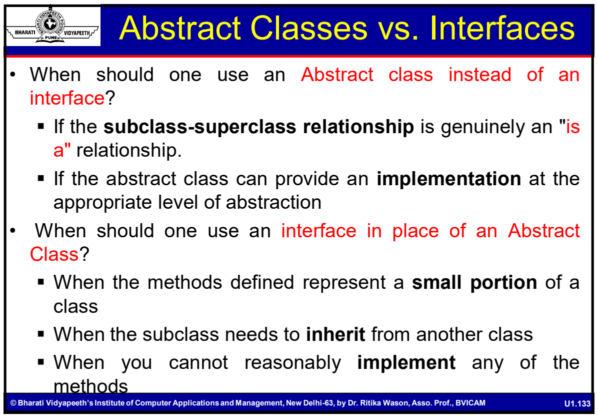
Interfaces..

- Partial Implementation
- If a class includes an interface but does not fully implements the methods defined by that interface then that class must be declared as abstract
- Used in initial stages of **Project Planning** as a **blueprint**



Multiple Inheritance?

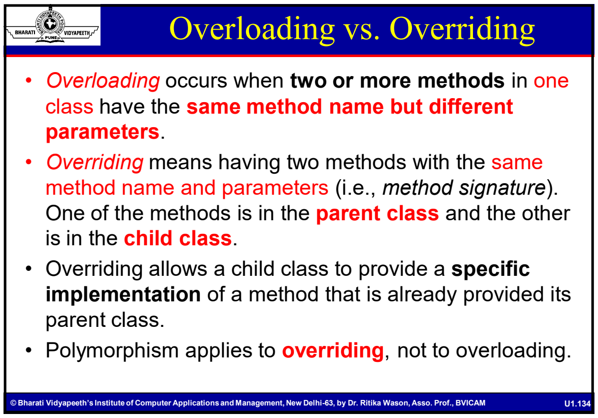
- Allowing classes to implement multiple interfaces is the same thing as multiple inheritance
- This is **NOT** true. When you implement an interface:
  - The implementing class **does not inherit instance variables**
  - The implementing class **does not inherit methods** (none are defined)
  - The Implementing class **does not inherit associations**
- Implementation of interfaces is not inheritance.
- *An interface defines a list of methods which must be implemented.*
- Interfaces afford the benefits of multiple inheritance while avoiding the complexities and inefficiencies



Abstract Classes vs. Interfaces

- When should one use an Abstract class instead of an interface?
  - If the **subclass-superclass relationship** is genuinely an "is a" relationship.
  - If the abstract class can provide an **implementation** at the appropriate level of abstraction
- When should one use an interface in place of an Abstract Class?
  - When the methods defined represent a **small portion** of a class
  - When the subclass needs to **inherit** from another class
  - When you cannot reasonably **implement** any of the methods

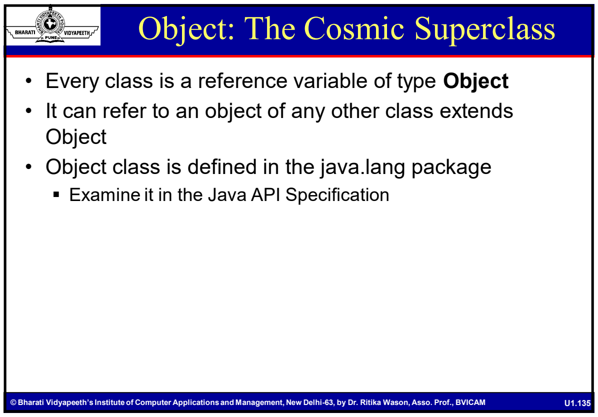
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.133



Overloading vs. Overriding

- *Overloading* occurs when **two or more methods** in one class have the **same method name but different parameters**.
- *Overriding* means having two methods with the same method name and parameters (i.e., *method signature*). One of the methods is in the **parent class** and the other is in the **child class**.
- Overriding allows a child class to provide a **specific implementation** of a method that is already provided its parent class.
- Polymorphism applies to **overriding**, not to overloading.

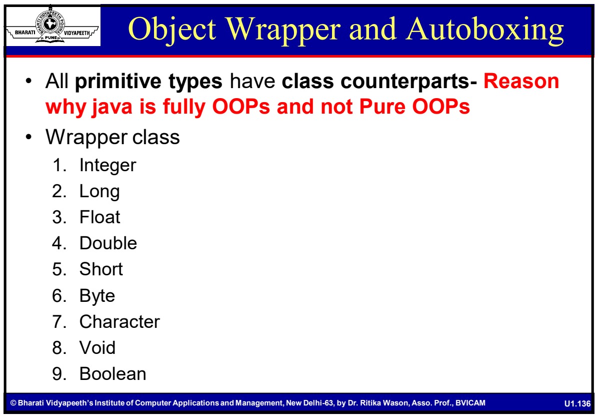
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.134



Object: The Cosmic Superclass

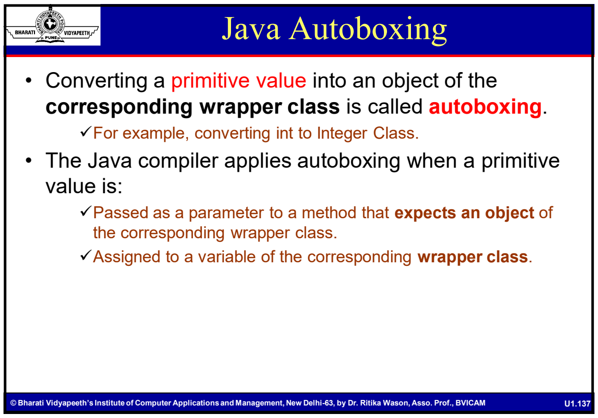
- Every class is a reference variable of type **Object**
- It can refer to an object of any other class extends Object
- Object class is defined in the java.lang package
  - Examine it in the Java API Specification

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.135



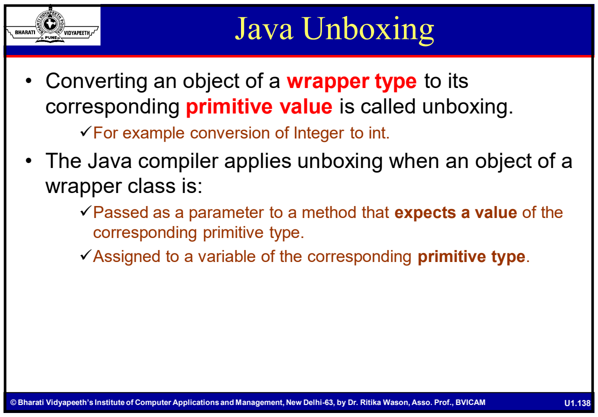
Object Wrapper and Autoboxing

- All **primitive types** have **class counterparts- Reason why java is fully OOPs and not Pure OOPs**
- Wrapper class
  1. Integer
  2. Long
  3. Float
  4. Double
  5. Short
  6. Byte
  7. Character
  8. Void
  9. Boolean



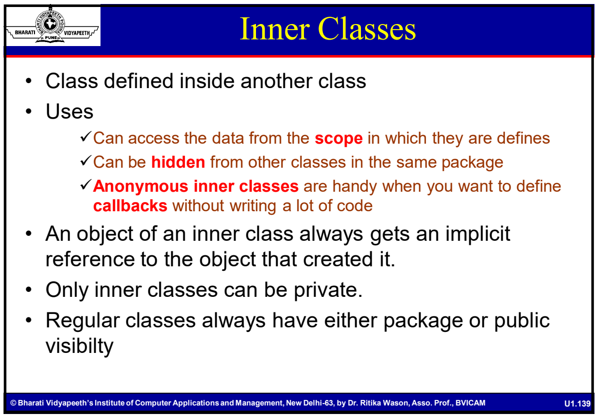
Java Autoboxing

- Converting a primitive value into an object of the **corresponding wrapper class** is called **autoboxing**.
  - ✓For example, converting int to Integer Class.
- The Java compiler applies autoboxing when a primitive value is:
  - ✓Passed as a parameter to a method that **expects an object** of the corresponding wrapper class.
  - ✓Assigned to a variable of the corresponding **wrapper class**.



Java Unboxing

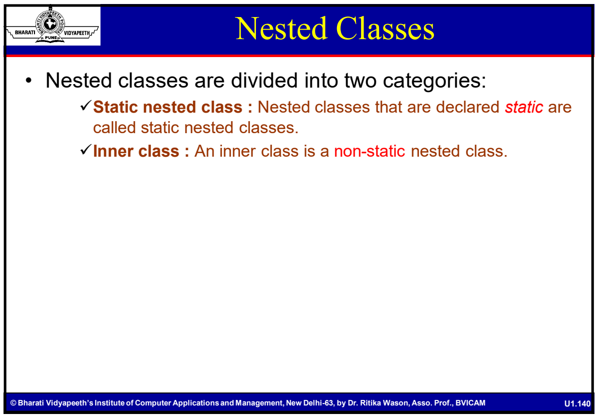
- Converting an object of a **wrapper type** to its corresponding **primitive value** is called unboxing.
  - ✓For example conversion of Integer to int.
- The Java compiler applies unboxing when an object of a wrapper class is:
  - ✓Passed as a parameter to a method that **expects a value** of the corresponding primitive type.
  - ✓Assigned to a variable of the corresponding **primitive type**.



Inner Classes

- Class defined inside another class
- Uses
  - ✓Can access the data from the **scope** in which they are defines
  - ✓Can be **hidden** from other classes in the same package
  - ✓**Anonymous inner classes** are handy when you want to define **callbacks** without writing a lot of code
- An object of an inner class always gets an implicit reference to the object that created it.
- Only inner classes can be private.
- Regular classes always have either package or public visibilty

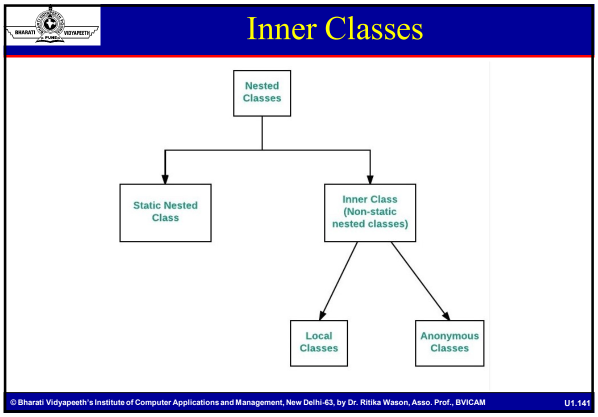
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.139



Nested Classes

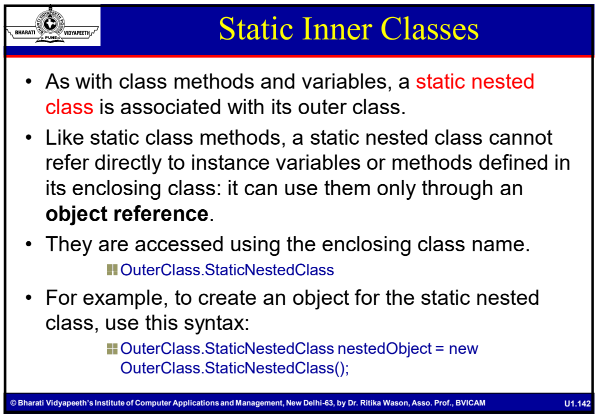
- Nested classes are divided into two categories:
  - ✓**Static nested class :** Nested classes that are declared *static* are called static nested classes.
  - ✓**Inner class :** An inner class is a non-static nested class.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.140



Inner Classes

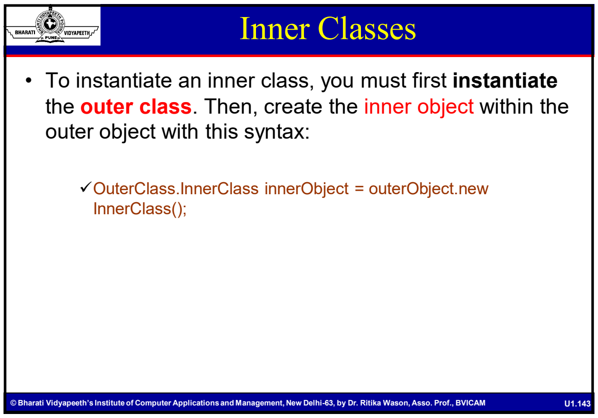
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.141



Static Inner Classes

- As with class methods and variables, a static nested class is associated with its outer class.
- Like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class: it can use them only through an **object reference**.
- They are accessed using the enclosing class name.
  - OuterClass.StaticNestedClass
- For example, to create an object for the static nested class, use this syntax:
  - OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.142

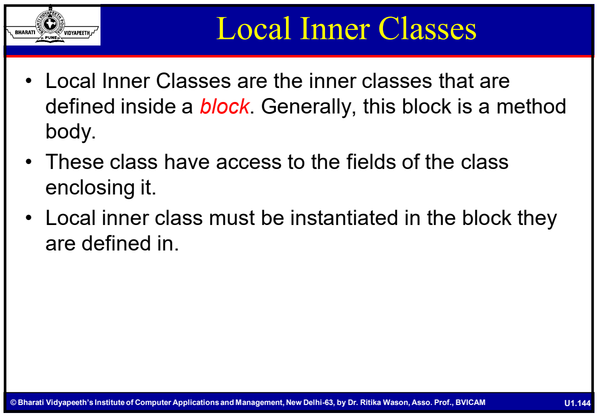


Inner Classes

- To instantiate an inner class, you must first **instantiate** the **outer class**. Then, create the inner object within the outer object with this syntax:

  ✓OuterClass.InnerClass innerObject = outerObject.new InnerClass();

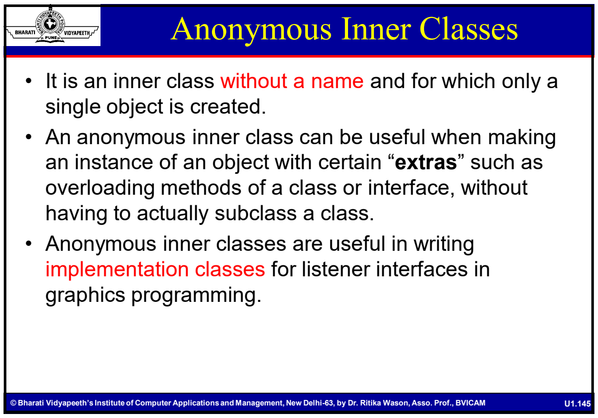
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.143



Local Inner Classes

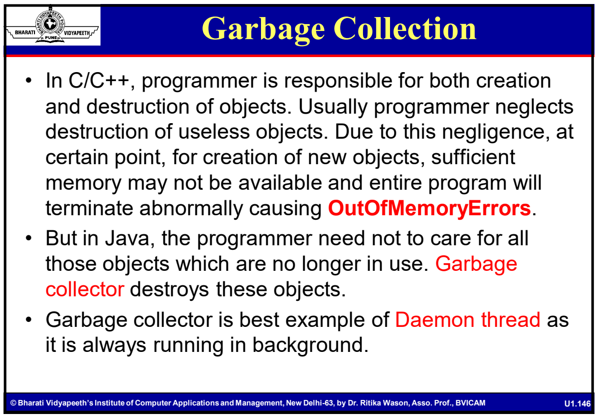
- Local Inner Classes are the inner classes that are defined inside a *block*. Generally, this block is a method body.
- These class have access to the fields of the class enclosing it.
- Local inner class must be instantiated in the block they are defined in.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.144



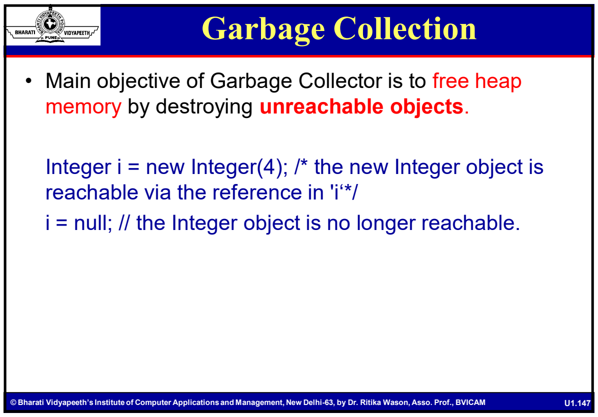
Anonymous Inner Classes

- It is an inner class without a name and for which only a single object is created.
- An anonymous inner class can be useful when making an instance of an object with certain "**extras**" such as overloading methods of a class or interface, without having to actually subclass a class.
- Anonymous inner classes are useful in writing implementation classes for listener interfaces in graphics programming.



Garbage Collection

- In C/C++, programmer is responsible for both creation and destruction of objects. Usually programmer neglects destruction of useless objects. Due to this negligence, at certain point, for creation of new objects, sufficient memory may not be available and entire program will terminate abnormally causing **OutOfMemoryErrors**.
- But in Java, the programmer need not to care for all those objects which are no longer in use. Garbage collector destroys these objects.
- Garbage collector is best example of Daemon thread as it is always running in background.

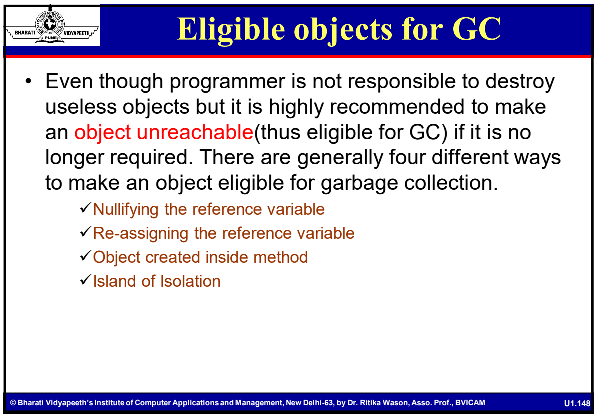


Garbage Collection

- Main objective of Garbage Collector is to free heap memory by destroying **unreachable objects**.

Integer i = new Integer(4); /* the new Integer object is reachable via the reference in 'i'*/

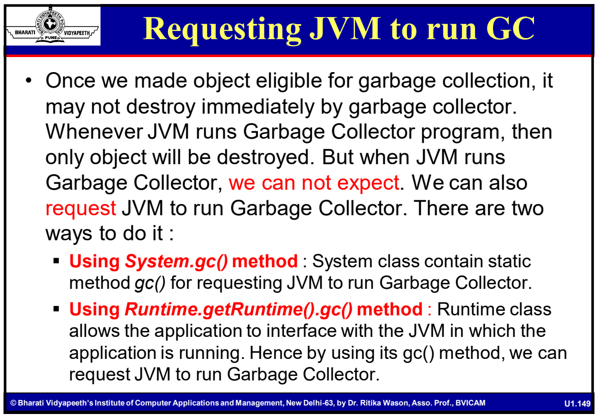
i = null; // the Integer object is no longer reachable.



Eligible objects for GC

- Even though programmer is not responsible to destroy useless objects but it is highly recommended to make an object unreachable(thus eligible for GC) if it is no longer required. There are generally four different ways to make an object eligible for garbage collection.
  - ✓Nullifying the reference variable
  - ✓Re-assigning the reference variable
  - ✓Object created inside method
  - ✓Island of Isolation

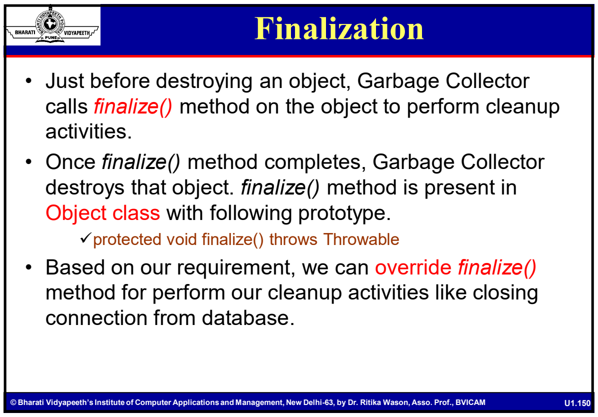
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.148



Requesting JVM to run GC

- Once we made object eligible for garbage collection, it may not destroy immediately by garbage collector. Whenever JVM runs Garbage Collector program, then only object will be destroyed. But when JVM runs Garbage Collector, we can not expect. We can also request JVM to run Garbage Collector. There are two ways to do it :
  - **Using *System.gc()* method** : System class contain static method *gc()* for requesting JVM to run Garbage Collector.
  - **Using *Runtime.getRuntime().gc()* method** : Runtime class allows the application to interface with the JVM in which the application is running. Hence by using its gc() method, we can request JVM to run Garbage Collector.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.149



Finalization

- Just before destroying an object, Garbage Collector calls *finalize()* method on the object to perform cleanup activities.
- Once *finalize()* method completes, Garbage Collector destroys that object. *finalize()* method is present in Object class with following prototype.
  - ✓protected void finalize() throws Throwable
- Based on our requirement, we can override *finalize()* method for perform our cleanup activities like closing connection from database.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Ritika Wason, Asso. Prof., BVICAM U1.150