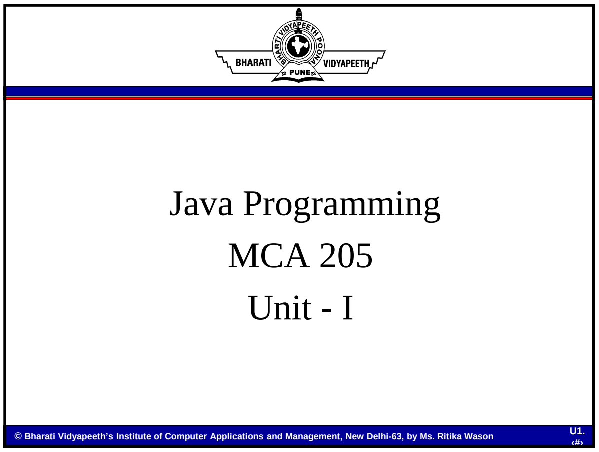
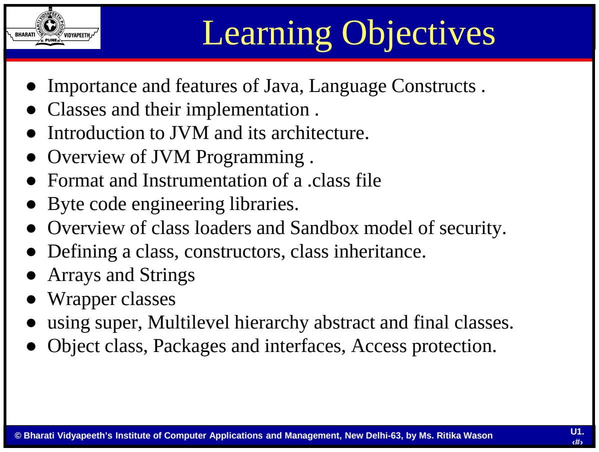


Java Programming

MCA 205

Unit - I

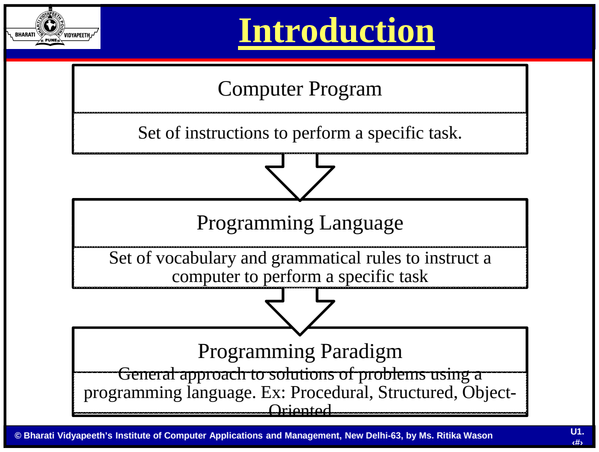
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Learning Objectives

- Importance and features of Java, Language Constructs .
- Classes and their implementation .
- Introduction to JVM and its architecture.
- Overview of JVM Programming .
- Format and Instrumentation of a .class file
- Byte code engineering libraries.
- Overview of class loaders and Sandbox model of security.
- Defining a class, constructors, class inheritance.
- Arrays and Strings
- Wrapper classes
- using super, Multilevel hierarchy abstract and final classes.
- Object class, Packages and interfaces, Access protection.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Introduction

Computer Program

Set of instructions to perform a specific task.

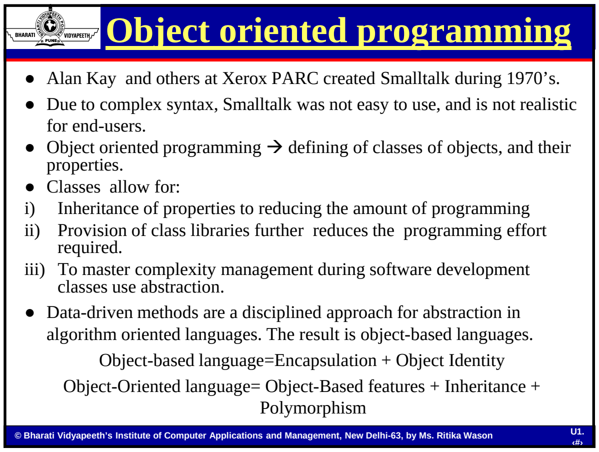
Programming Language

Set of vocabulary and grammatical rules to instruct a computer to perform a specific task

Programming Paradigm

General approach to solutions of problems using a programming language. Ex: Procedural, Structured, Object-Oriented

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Object oriented programming

- Alan Kay and others at Xerox PARC created Smalltalk during 1970's.
- Due to complex syntax, Smalltalk was not easy to use, and is not realistic for end-users.
- Object oriented programming → defining of classes of objects, and their properties.
- Classes allow for:

i) Inheritance of properties to reducing the amount of programming

ii) Provision of class libraries further reduces the programming effort required.

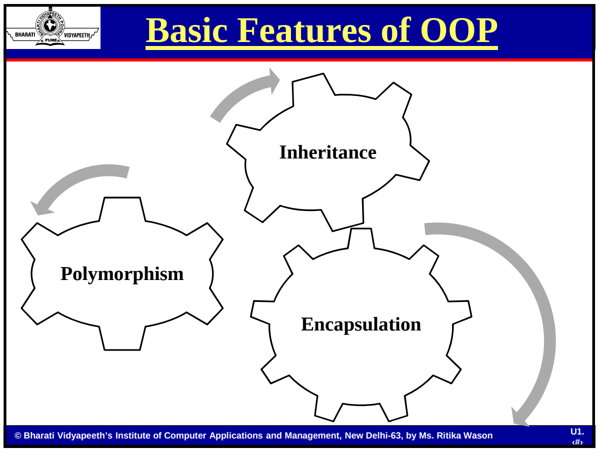
iii) To master complexity management during software development classes use abstraction.

- Data-driven methods are a disciplined approach for abstraction in algorithm oriented languages. The result is object-based languages.

Object-based language=Encapsulation + Object Identity

Object-Oriented language= Object-Based features + Inheritance + Polymorphism

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



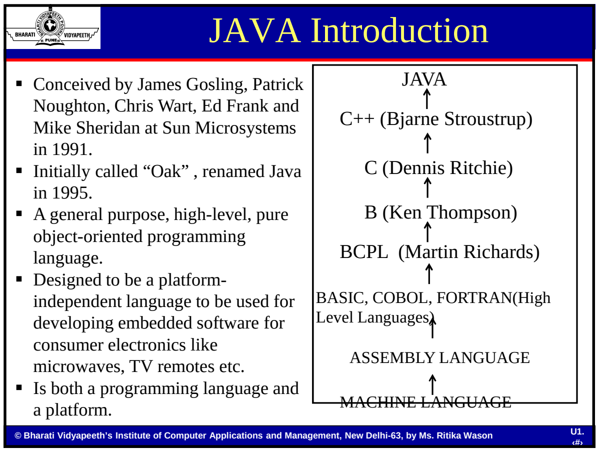
Basic Features of OOP

Inheritance

Polymorphism

Encapsulation

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›

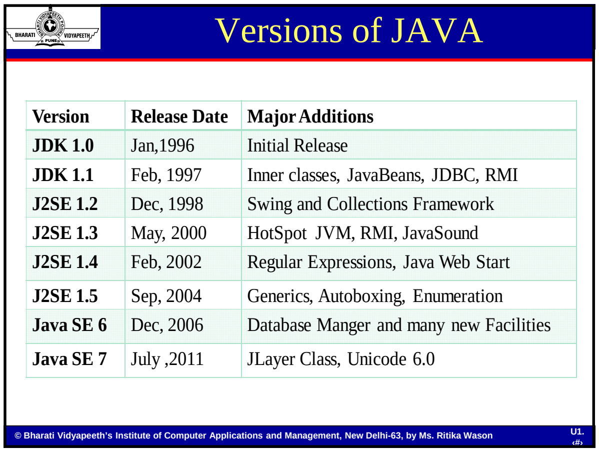


JAVA Introduction

- Conceived by James Gosling, Patrick Noughton, Chris Wart, Ed Frank and Mike Sheridan at Sun Microsystems in 1991.
- Initially called "Oak" , renamed Java in 1995.
- A general purpose, high-level, pure object-oriented programming language.
- Designed to be a platform-independent language to be used for developing embedded software for consumer electronics like microwaves, TV remotes etc.
- Is both a programming language and a platform.

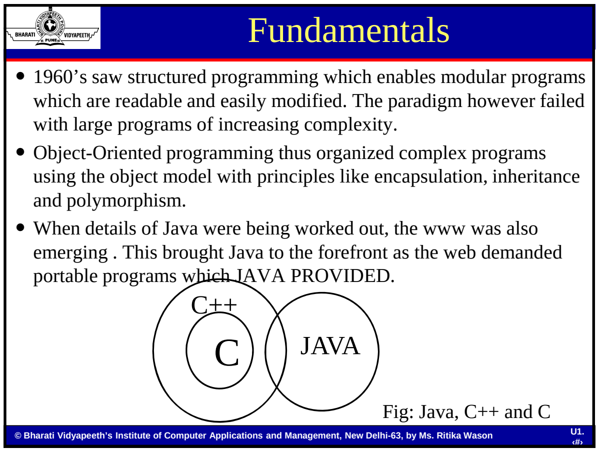
JAVA
↑
C++ (Bjarne Stroustrup)
↑
C (Dennis Ritchie)
↑
B (Ken Thompson)
↑
BCPL (Martin Richards)
↑
BASIC, COBOL, FORTRAN(High Level Languages)
↑
ASSEMBLY LANGUAGE
↑
MACHINE LANGUAGE

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Versions of JAVA

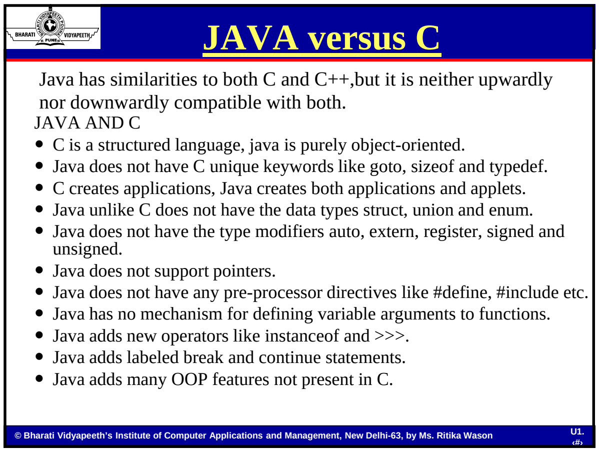
Version	Release Date	Major Additions
JDK 1.0	Jan,1996	Initial Release
JDK 1.1	Feb, 1997	Inner classes, JavaBeans, JDBC, RMI
J2SE 1.2	Dec, 1998	Swing and Collections Framework
J2SE 1.3	May, 2000	HotSpot JVM, RMI, JavaSound
J2SE 1.4	Feb, 2002	Regular Expressions, Java Web Start
J2SE 1.5	Sep, 2004	Generics, Autoboxing, Enumeration
Java SE 6	Dec, 2006	Database Manger and many new Facilities
Java SE 7	July ,2011	JLayer Class, Unicode 6.0



Fundamentals

- 1960's saw structured programming which enables modular programs which are readable and easily modified. The paradigm however failed with large programs of increasing complexity.
- Object-Oriented programming thus organized complex programs using the object model with principles like encapsulation, inheritance and polymorphism.
- When details of Java were being worked out, the www was also emerging . This brought Java to the forefront as the web demanded portable programs which JAVA PROVIDED.

Fig: Java, C++ and C

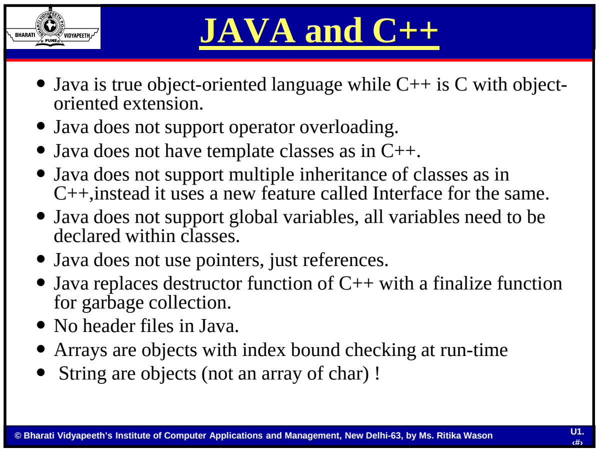


JAVA versus C

Java has similarities to both C and C++,but it is neither upwardly nor downwardly compatible with both.

JAVA AND C

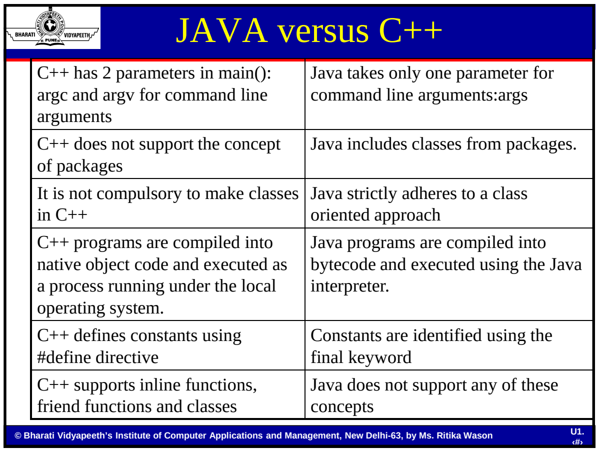
- C is a structured language, java is purely object-oriented.
- Java does not have C unique keywords like goto, sizeof and typedef.
- C creates applications, Java creates both applications and applets.
- Java unlike C does not have the data types struct, union and enum.
- Java does not have the type modifiers auto, extern, register, signed and unsigned.
- Java does not support pointers.
- Java does not have any pre-processor directives like #define, #include etc.
- Java has no mechanism for defining variable arguments to functions.
- Java adds new operators like instanceof and >>>.
- Java adds labeled break and continue statements.
- Java adds many OOP features not present in C.



JAVA and C++

- Java is true object-oriented language while C++ is C with object-oriented extension.
- Java does not support operator overloading.
- Java does not have template classes as in C++.
- Java does not support multiple inheritance of classes as in C++,instead it uses a new feature called Interface for the same.
- Java does not support global variables, all variables need to be declared within classes.
- Java does not use pointers, just references.
- Java replaces destructor function of C++ with a finalize function for garbage collection.
- No header files in Java.
- Arrays are objects with index bound checking at run-time
- String are objects (not an array of char) !

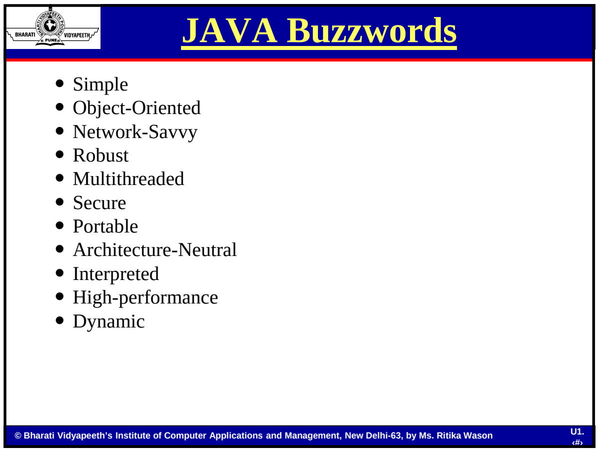
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



JAVA versus C++

C++ has 2 parameters in main(): argc and argv for command line arguments	Java takes only one parameter for command line arguments:args
C++ does not support the concept of packages	Java includes classes from packages.
It is not compulsory to make classes in C++	Java strictly adheres to a class oriented approach
C++ programs are compiled into native object code and executed as a process running under the local operating system.	Java programs are compiled into bytecode and executed using the Java interpreter.
C++ defines constants using #define directive	Constants are identified using the final keyword
C++ supports inline functions, friend functions and classes	Java does not support any of these concepts

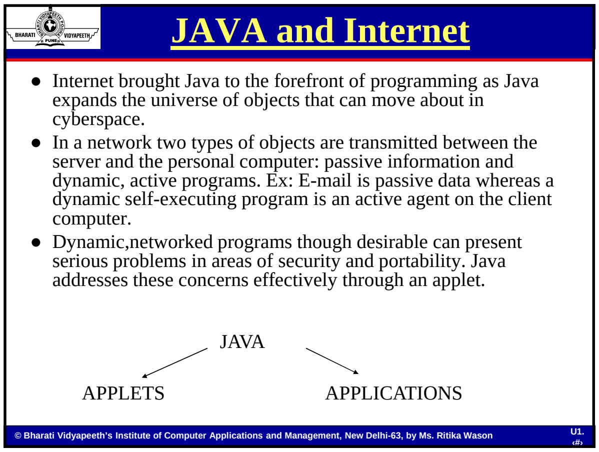
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



JAVA Buzzwords

- Simple
- Object-Oriented
- Network-Savvy
- Robust
- Multithreaded
- Secure
- Portable
- Architecture-Neutral
- Interpreted
- High-performance
- Dynamic

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



JAVA and Internet

- Internet brought Java to the forefront of programming as Java expands the universe of objects that can move about in cyberspace.
- In a network two types of objects are transmitted between the server and the personal computer: passive information and dynamic, active programs. Ex: E-mail is passive data whereas a dynamic self-executing program is an active agent on the client computer.
- Dynamic,networked programs though desirable can present serious problems in areas of security and portability. Java addresses these concerns effectively through an applet.

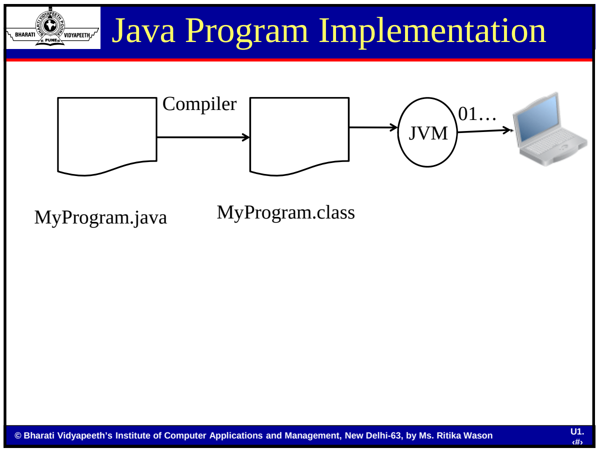
JAVA

APPLETS

APPLICATIONS

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason

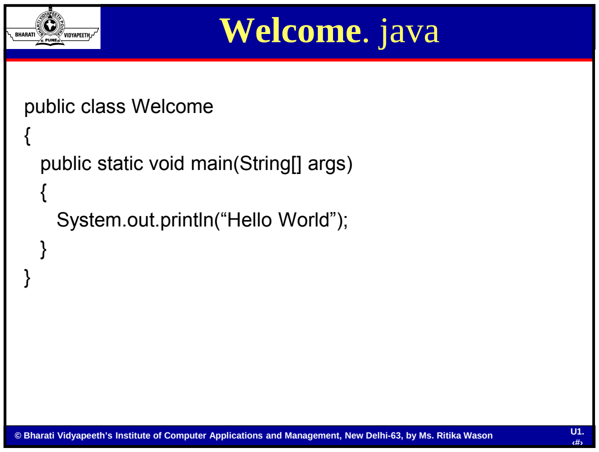
U1.‹#›



Java Program Implementation

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason

U1.‹#›

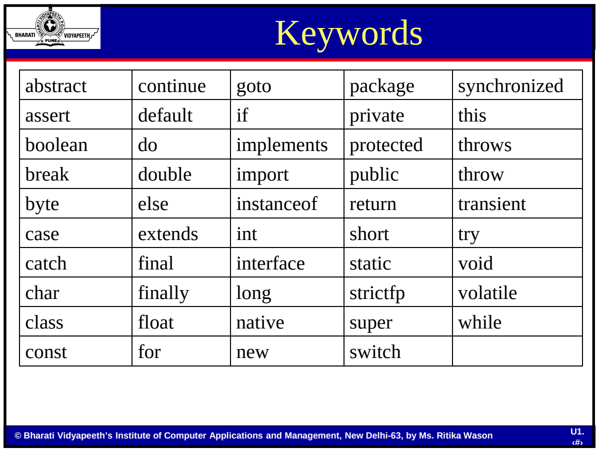


Welcome. java

```
public class Welcome
{
  public static void main(String[] args)
  {
    System.out.println(“Hello World”);
  }
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason

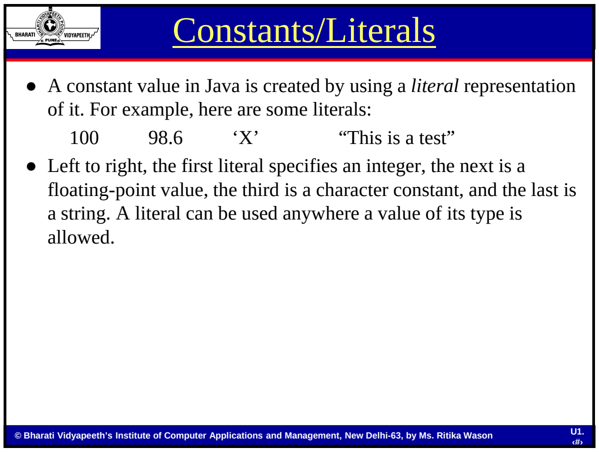
U1.‹#›



Keywords

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

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›

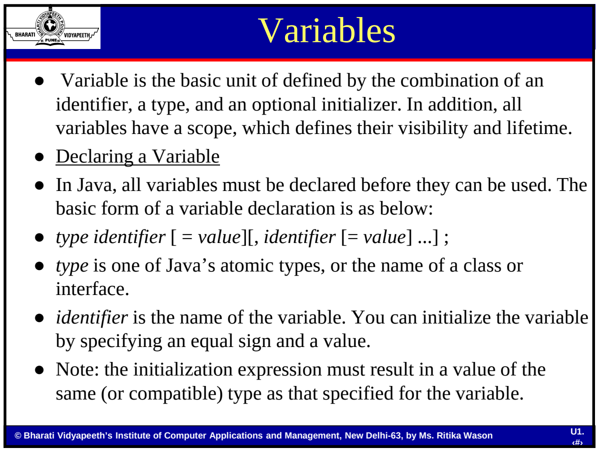


Constants/Literals

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

  100 98.6 ‘X’ “This is a test”
- Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string. A literal can be used anywhere a value of its type is allowed.

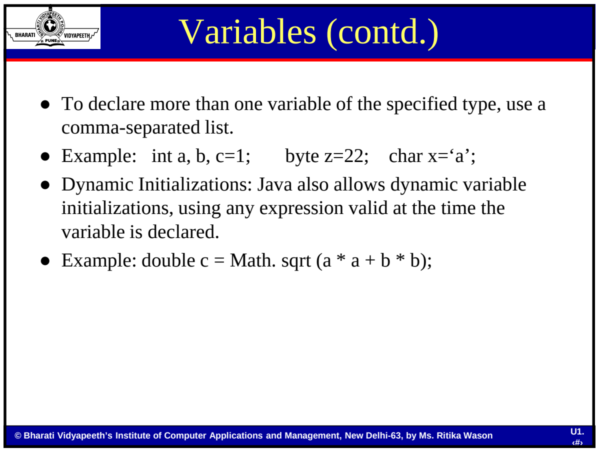
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Variables

- Variable is the basic unit of 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 lifetime.
- Declaring a Variable
- In Java, all variables must be declared before they can be used. The basic form of a variable declaration is as below:
- *type identifier* [= *value*][, *identifier* [= *value*] ...] ;
- *type* is one of Java’s atomic types, or the name of a class or interface.
- *identifier* is the name of the variable. You can initialize the variable by specifying an equal sign and a value.
- Note: the initialization expression must result in a value of the same (or compatible) type as that specified for the variable.

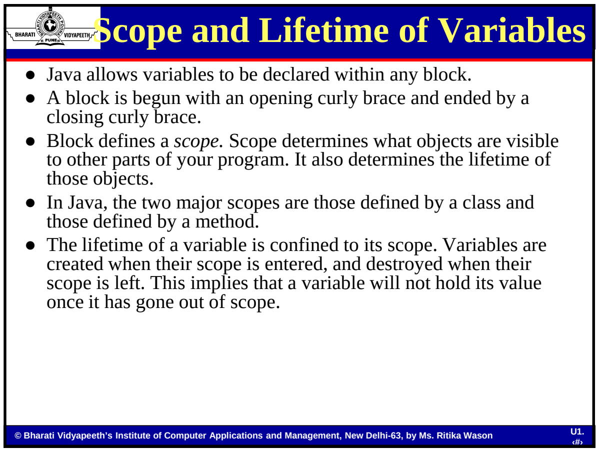
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Variables (contd.)

- To declare more than one variable of the specified type, use a comma-separated list.
- Example: int a, b, c=1; byte z=22; char x='a';
- Dynamic Initializations: Java also allows dynamic variable initializations, using any expression valid at the time the variable is declared.
- Example: double c = Math. sqrt (a * a + b * b);

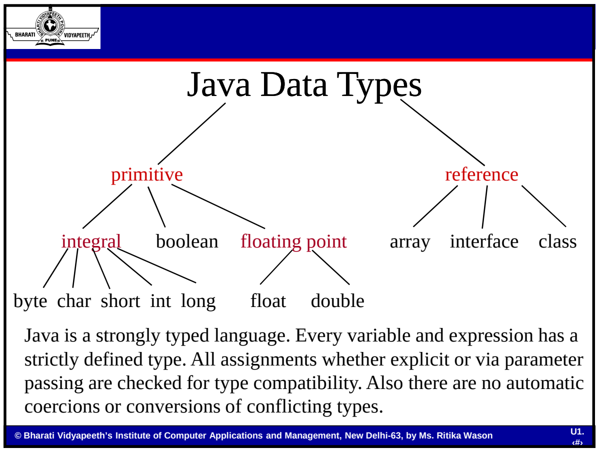
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Scope and Lifetime of Variables

- Java allows variables to be declared within any block.
- A block is begun with an opening curly brace and ended by a closing curly brace.
- Block defines a *scope.* Scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.
- In Java, the two major scopes are those defined by a class and those defined by a method.
- The lifetime of a variable is confined to its scope. Variables are created when their scope is entered, and destroyed when their scope is left. This implies that a variable will not hold its value once it has gone out of scope.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Java Data Types

primitive

reference

integral

boolean

floating point

array

interface

class

byte

char

short

int

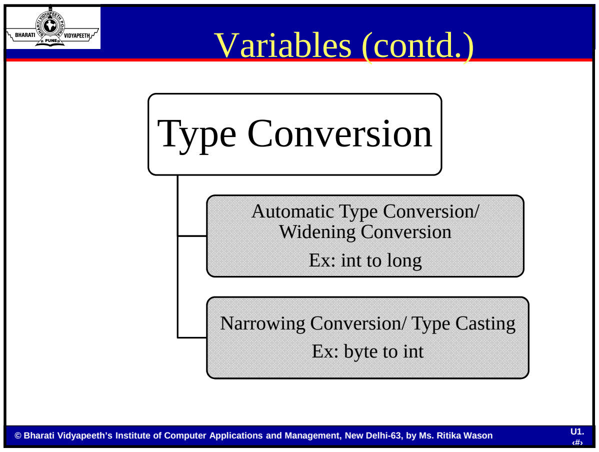
long

float

double

Java is a strongly typed language. Every variable and expression has a strictly defined type. All assignments whether explicit or via parameter passing are checked for type compatibility. Also there are no automatic coercions or conversions of conflicting types.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Variables (contd.)

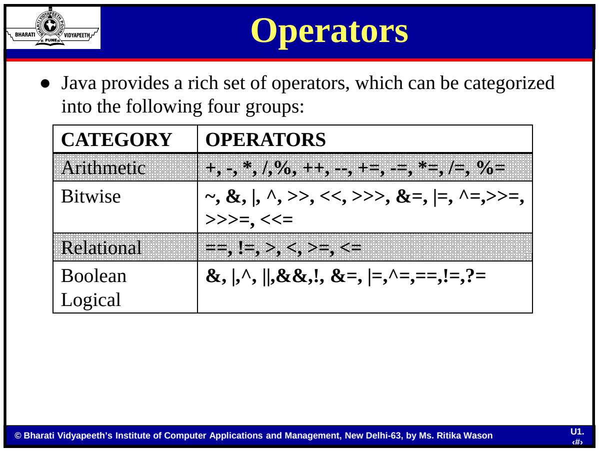
Type Conversion

Automatic Type Conversion/ Widening Conversion

Ex: int to long

Narrowing Conversion/ Type Casting

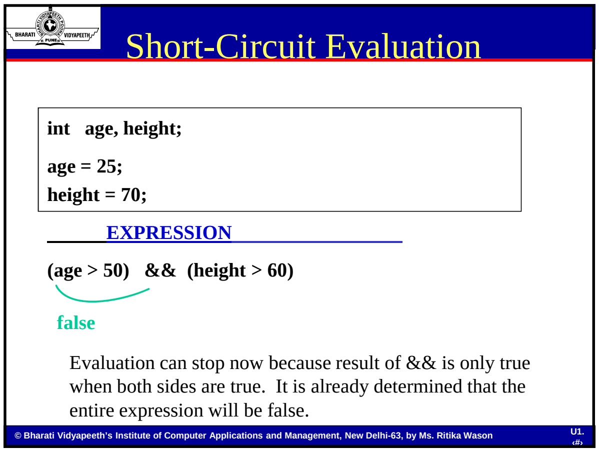
Ex: byte to int



Operators

- Java provides a rich set of operators, which can be categorized into the following four groups:

<table>
<tr><th>CATEGORY</th><th>OPERATORS</th></tr>
<tr><td>Arithmetic</td><td>+, -, *, /,%, ++, --, +=, -=, *=, /=, %=</td></tr>
<tr><td>Bitwise</td><td>~, &, |, ^, >>, <<, >>>, &=, |=, ^=,>>=, >>>=, <<=</td></tr>
<tr><td>Relational</td><td>==, !=, >, <, >=, <=</td></tr>
<tr><td>Boolean Logical</td><td>&, |,^, ||,&&,!, &=, |=,^=,==,!=,?=</td></tr>
</table>



Short-Circuit Evaluation

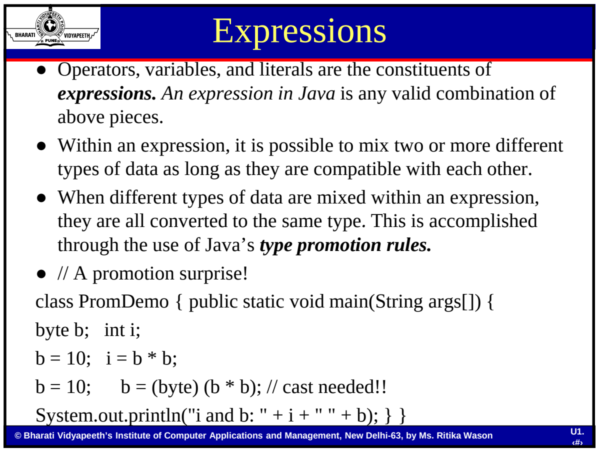
```
int   age, height;

age = 25;
height = 70;
```

EXPRESSION

(age > 50)   && (height > 60)

false

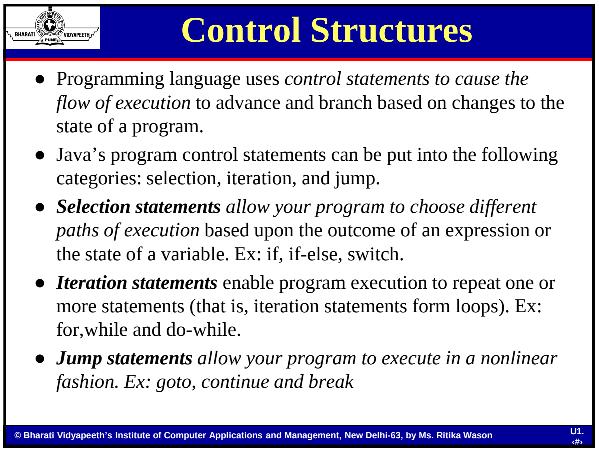
Evaluation can stop now because result of && is only true when both sides are true. It is already determined that the entire expression will be false.



Expressions

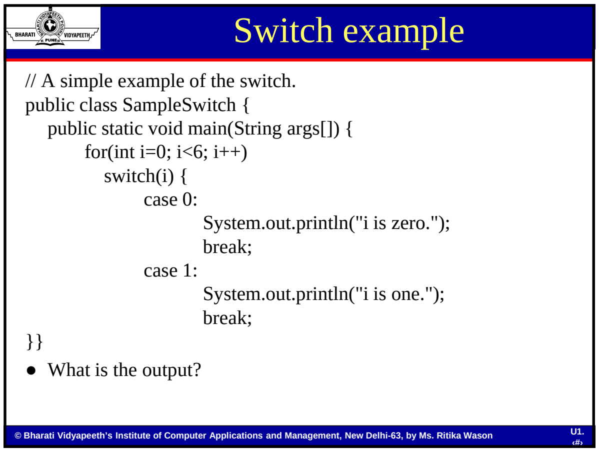
- Operators, variables, and literals are the constituents of ***expressions.*** *An expression in Java* is any valid combination of above pieces.
- Within an expression, it is possible to mix two or more different types of data as long as they are compatible with each other.
- When different types of data are mixed within an expression, they are all converted to the same type. This is accomplished through the use of Java's ***type promotion rules.***
- // A promotion surprise!

```
class PromDemo { public static void main(String args[]) {
byte b;   int i;
b = 10;   i = b * b;
b = 10;      b = (byte) (b * b); // cast needed!!
System.out.println("i and b: " + i + " " + b); } }
```



Control Structures

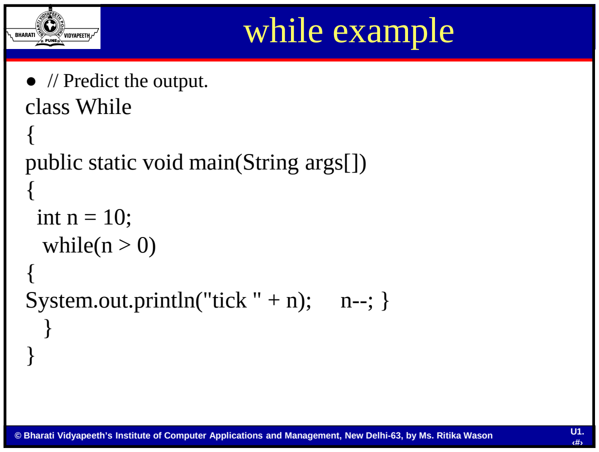
- Programming language uses *control statements to cause the flow of execution* to advance and branch based on changes to the state of a program.
- Java's program control statements can be put into the following categories: selection, iteration, and jump.
- ***Selection statements*** *allow your program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



Switch example

```
// A simple example of the switch.
public class SampleSwitch {
    public static void main(String args[]) {
          for(int i=0; i<6; i++)
             switch(i) {
                   case 0:
                            System.out.println("i is zero.");
                            break;
                   case 1:
                            System.out.println("i is one.");
                            break;
}}
```

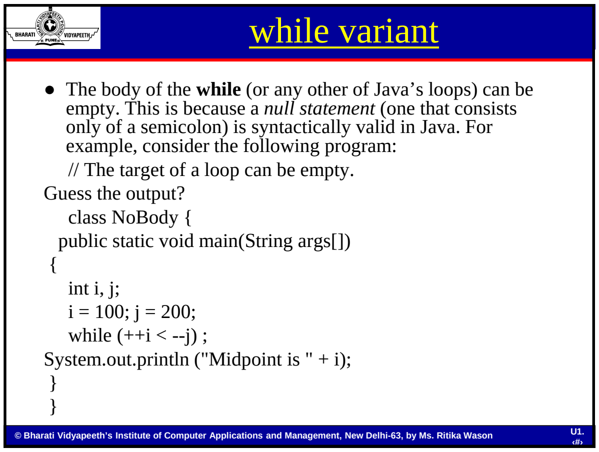
- What is the output?



while example

- // Predict the output.

```
class While
{
public static void main(String args[])
{
  int n = 10;
   while(n > 0)
{
System.out.println("tick " + n);     n--; }
   }
}
```



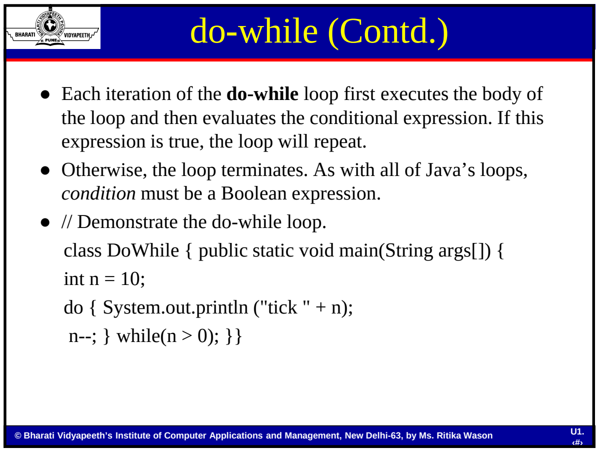
while variant

- The body of the **while** (or any other of Java's loops) can be empty. This is because a *null statement* (one that consists only of a semicolon) is syntactically valid in Java. For example, consider the following program:

  // The target of a loop can be empty.

Guess the output?

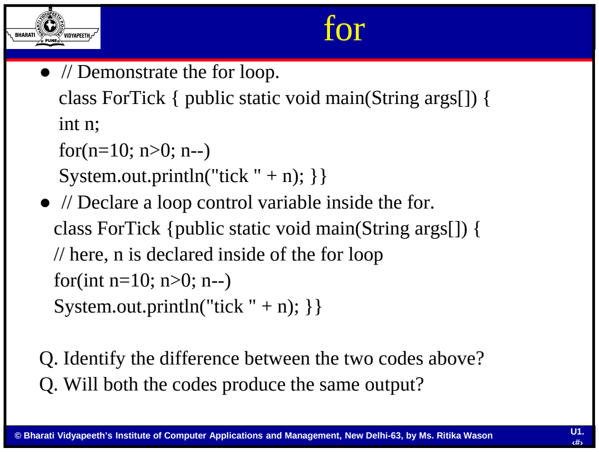
```
    class NoBody {
  public static void main(String args[])
 {
    int i, j;
    i = 100; j = 200;
    while (++i < --j) ;
System.out.println ("Midpoint is " + i);
 }
 }
```



do-while (Contd.)

- Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat.
- Otherwise, the loop terminates. As with all of Java's loops, *condition* must be a Boolean expression.
- // Demonstrate the do-while loop.

```
class DoWhile { public static void main(String args[]) {
int n = 10;
do { System.out.println ("tick " + n);
 n--; } while(n > 0); }}
```



for

- // Demonstrate the for loop.

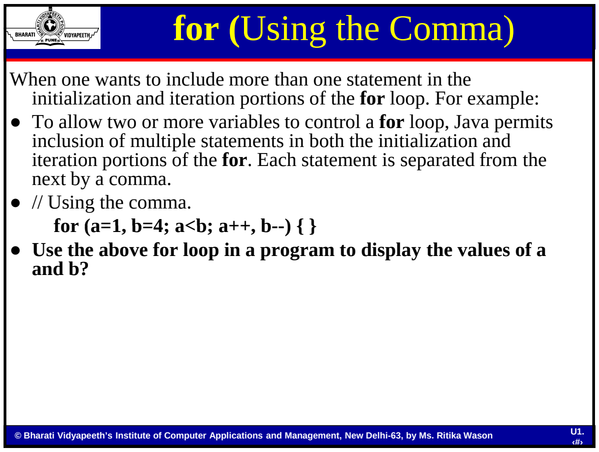
```
class ForTick { public static void main(String args[]) {
int n;
for(n=10; n>0; n--)
System.out.println("tick " + n); }}
```

- // Declare a loop control variable inside the for.

```
class ForTick {public static void main(String args[]) {
// here, n is declared inside of the for loop
for(int n=10; n>0; n--)
System.out.println("tick " + n); }}
```

Q. Identify the difference between the two codes above?

Q. Will both the codes produce the same output?

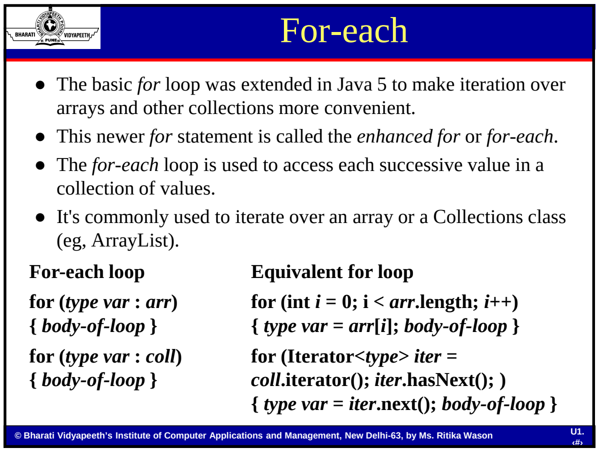


for (Using the Comma)

When one wants to include more than one statement in the initialization and iteration portions of the **for** loop. For example:

- To allow two or more variables to control a **for** loop, Java permits inclusion of multiple statements in both the initialization and iteration portions of the **for**. Each statement is separated from the next by a comma.
- // Using the comma.

  for (a=1, b=4; a<b; a++, b--) { }
- **Use the above for loop in a program to display the values of a and b?**



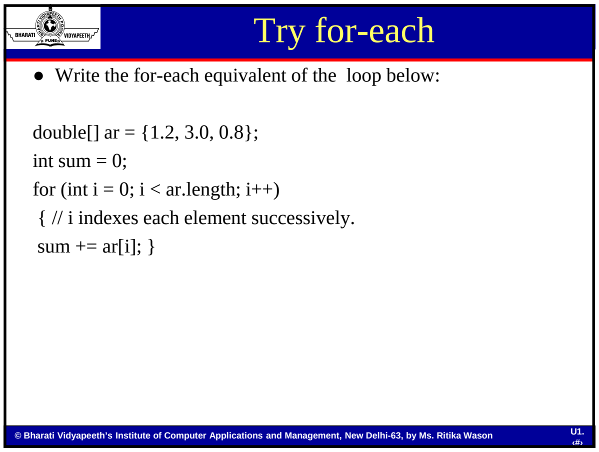
For-each

- The basic *for* loop was extended in Java 5 to make iteration over arrays and other collections more convenient.
- This newer *for* statement is called the *enhanced for* or *for-each*.
- The *for-each* loop is used to access each successive value in a collection of values.
- It's commonly used to iterate over an array or a Collections class (eg, ArrayList).

For-each loop	Equivalent for loop

for (*type var* : *arr*)
{ *body-of-loop* }
	for (int *i* = 0; i < *arr*.length; *i*++)
	{ *type var* = *arr*[*i*]; *body-of-loop* }

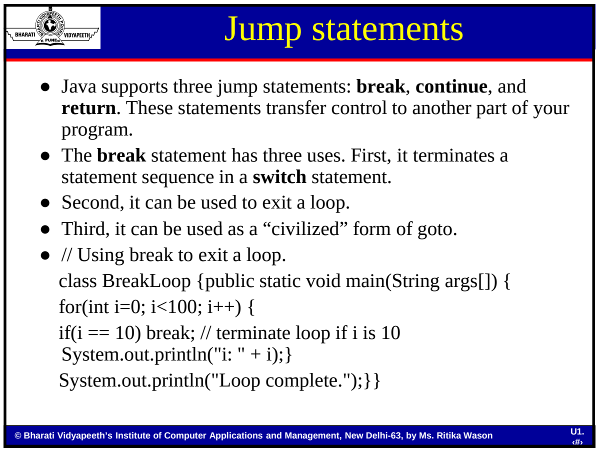
for (*type var* : *coll*)
{ *body-of-loop* }
	for (Iterator<*type*> *iter* = *coll*.iterator(); *iter*.hasNext();)
	{ *type var* = *iter*.next(); *body-of-loop* }



Try for-each

- Write the for-each equivalent of the loop below:

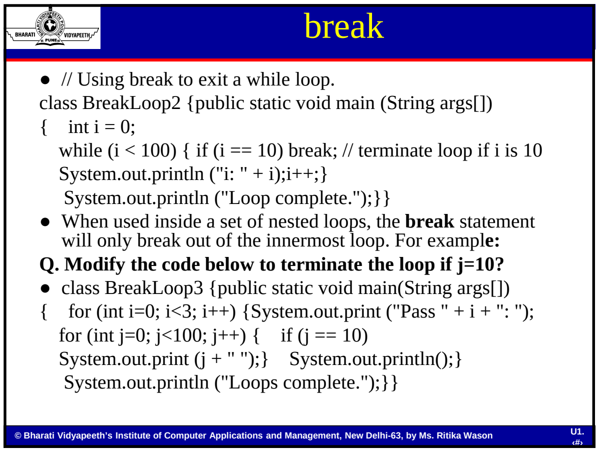
```
double[] ar = {1.2, 3.0, 0.8};
int sum = 0;
for (int i = 0; i < ar.length; i++)
 { // i indexes each element successively.
 sum += ar[i]; }
```



Jump statements

- Java supports three jump statements: **break**, **continue**, and **return**. These statements transfer control to another part of your program.
- The **break** statement has three uses. First, it terminates a statement sequence in a **switch** statement.
- Second, it can be used to exit a loop.
- Third, it can be used as a “civilized” form of goto.
- // Using break to exit a loop.

```
class BreakLoop {public static void main(String args[]) {
for(int i=0; i<100; i++) {
if(i == 10) break; // terminate loop if i is 10
 System.out.println("i: " + i);}
System.out.println("Loop complete.");}}
```



break

- // Using break to exit a while loop.

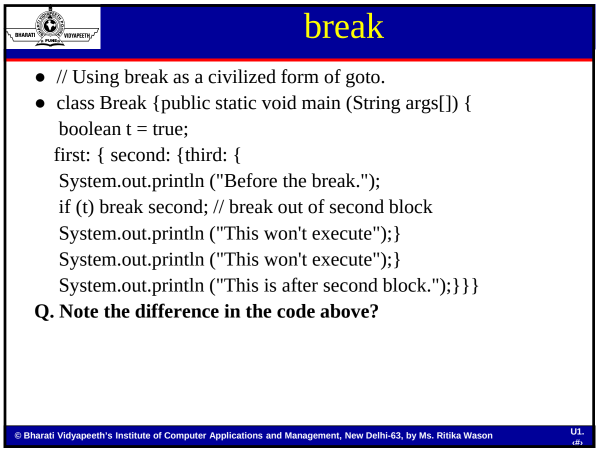
```
class BreakLoop2 {public static void main (String args[])
{    int i = 0;
   while (i < 100) { if (i == 10) break; // terminate loop if i is 10
   System.out.println ("i: " + i);i++;}
    System.out.println ("Loop complete.");}}
```

- When used inside a set of nested loops, the **break** statement will only break out of the innermost loop. For exampl**e:**

Q. Modify the code below to terminate the loop if j=10?

- class BreakLoop3 {public static void main(String args[])

```
{    for (int i=0; i<3; i++) {System.out.print ("Pass " + i + ": ");
   for (int j=0; j<100; j++) {    if (j == 10)
   System.out.print (j + " ");}    System.out.println();}
    System.out.println ("Loops complete.");}}
```



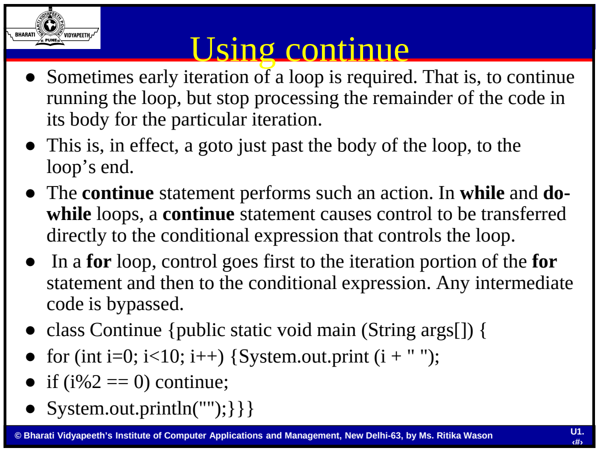
break

- // Using break as a civilized form of goto.
- class Break {public static void main (String args[]) {

```
 boolean t = true;
first: { second: {third: {
 System.out.println ("Before the break.");
 if (t) break second; // break out of second block
 System.out.println ("This won't execute");}
 System.out.println ("This won't execute");}
 System.out.println ("This is after second block.");}}}
```

Q. Note the difference in the code above?

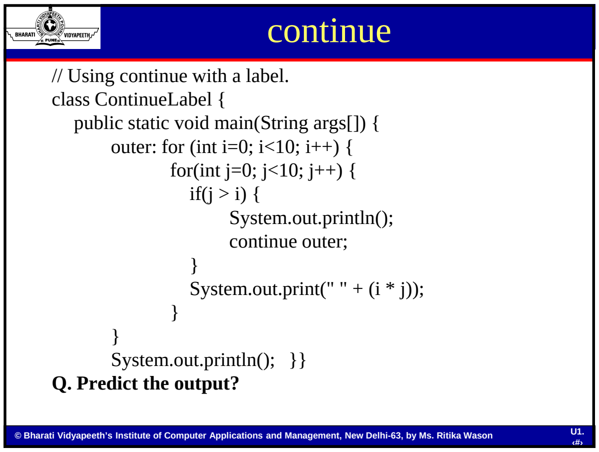
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Using continue

- Sometimes early iteration of a loop is required. That is, to continue running the loop, but stop processing the remainder of the code in its body for the particular iteration.
- This is, in effect, a goto just past the body of the loop, to the loop’s end.
- The **continue** statement performs such an action. In **while** and **do-while** loops, a **continue** statement causes control to be transferred directly to the conditional expression that controls the loop.
- In a **for** loop, control goes first to the iteration portion of the **for** statement and then to the conditional expression. Any intermediate code is bypassed.
- class Continue {public static void main (String args[]) {
- for (int i=0; i<10; i++) {System.out.print (i + " ");
- if (i%2 == 0) continue;
- System.out.println("");}}}

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



continue

```
// Using continue with a label.
class ContinueLabel {
    public static void main(String args[]) {
          outer: for (int i=0; i<10; i++) {
                    for(int j=0; j<10; j++) {
                       if(j > i) {
                             System.out.println();
                             continue outer;
                       }
                       System.out.print(" " + (i * j));
                    }
          }
          System.out.println();   }}
```

Q. Predict the output?

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›

return

- **return** statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.
- // Demonstrate return.

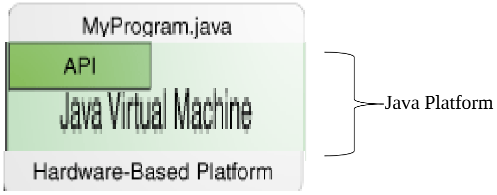
```
class Return {public static void main (String args[]) {
 boolean t = true;
 System.out.println ("Before the return.");
 if (t) return; // return to caller
 System.out.println ("This won't execute.");}}
```

Q. In above code return returns execution to whom?

JAVA Platform

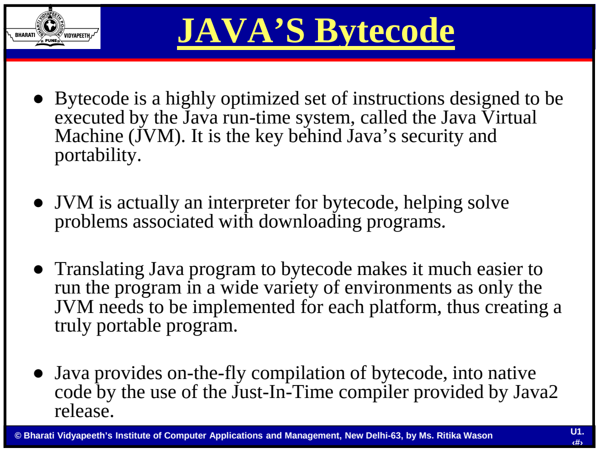
has two components:

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



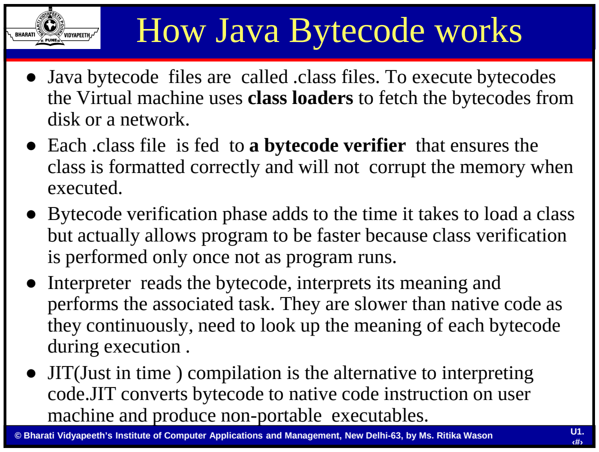
How Java works

- C++ compiler generates executable program code
- Java compiler generated architecture independent bytecode .the translation of java prg to byte code make it portable
- The bytecodes can be executed on JVM only usually independent in s/w rather h/w
  - Only JVM needs to be implemented for each platform
  - Details of JVM differ from one platform to other but all interprets same Java bytecode
- **BYTECODE**
  - Is a highly optimized set of instruction to be executed by JVM
  - Translating java program to bytecode –portable



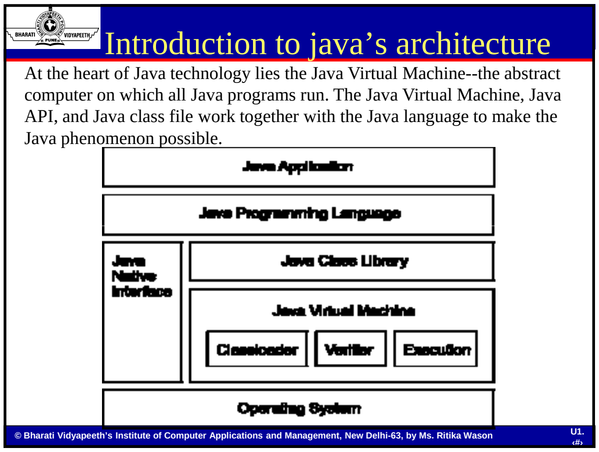
JAVA'S Bytecode

- Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, called the Java Virtual Machine (JVM). It is the key behind Java's security and portability.
- JVM is actually an interpreter for bytecode, helping solve problems associated with downloading programs.
- Translating Java program to bytecode makes it much easier to run the program in a wide variety of environments as only the JVM needs to be implemented for each platform, thus creating a truly portable program.
- Java provides on-the-fly compilation of bytecode, into native code by the use of the Just-In-Time compiler provided by Java2 release.



How Java Bytecode works

- Java bytecode files are called .class files. To execute bytecodes the Virtual machine uses **class loaders** to fetch the bytecodes from disk or a network.
- Each .class file is fed to **a bytecode verifier** that ensures the class is formatted correctly and will not corrupt the memory when executed.
- Bytecode verification phase adds to the time it takes to load a class but actually allows program to be faster because class verification is performed only once not as program runs.
- Interpreter reads the bytecode, interprets its meaning and performs the associated task. They are slower than native code as they continuously, need to look up the meaning of each bytecode during execution .
- JIT(Just in time) compilation is the alternative to interpreting code.JIT converts bytecode to native code instruction on user machine and produce non-portable executables.



Introduction to java's architecture

At the heart of Java technology lies the Java Virtual Machine--the abstract computer on which all Java programs run. The Java Virtual Machine, Java API, and Java class file work together with the Java language to make the Java phenomenon possible.

Java Application

Java Programming Language

Java Native Interface

Java Class Library

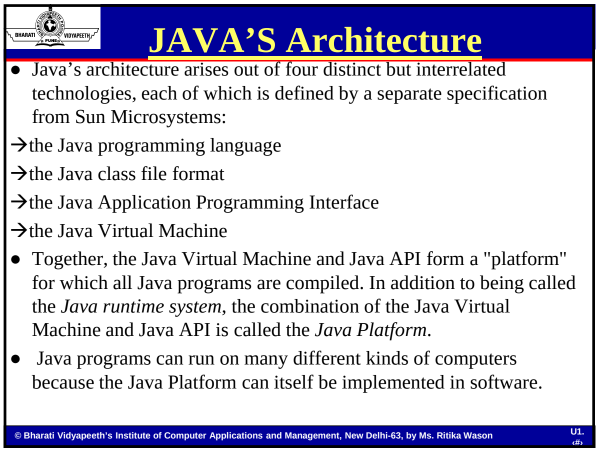
Java Virtual Machine

Classloader

Verifier

Execution

Operating System



JAVA’S Architecture

- Java’s architecture arises out of four distinct but interrelated technologies, each of which is defined by a separate specification from Sun Microsystems:

→the Java programming language

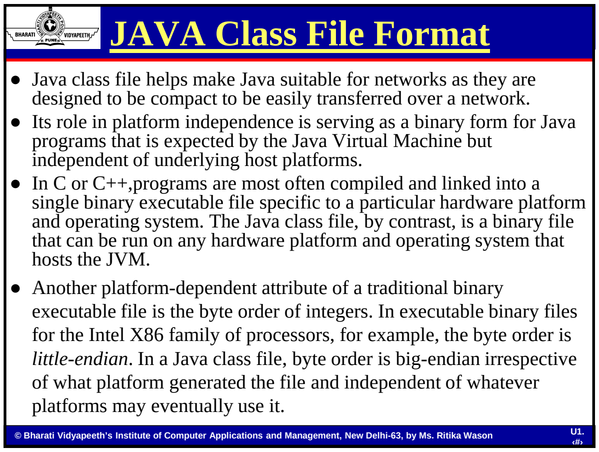
→the Java class file format

→the Java Application Programming Interface

→the Java Virtual Machine

- Together, the Java Virtual Machine and Java API form a "platform" for which all Java programs are compiled. In addition to being called the *Java runtime system*, the combination of the Java Virtual Machine and Java API is called the *Java Platform*.
- Java programs can run on many different kinds of computers because the Java Platform can itself be implemented in software.

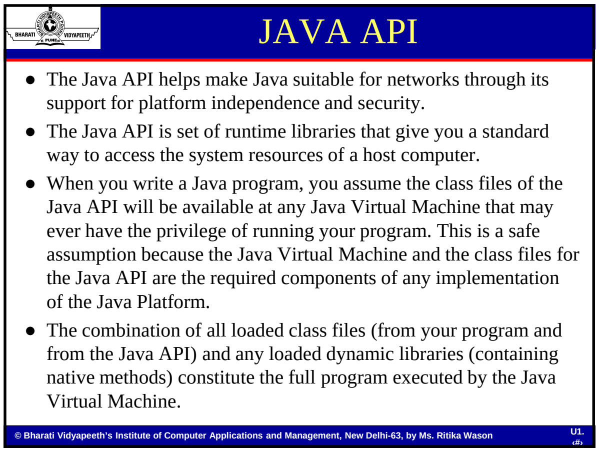
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



JAVA Class File Format

- Java class file helps make Java suitable for networks as they are designed to be compact to be easily transferred over a network.
- Its role in platform independence is serving as a binary form for Java programs that is expected by the Java Virtual Machine but independent of underlying host platforms.
- In C or C++,programs are most often compiled and linked into a single binary executable file specific to a particular hardware platform and operating system. The Java class file, by contrast, is a binary file that can be run on any hardware platform and operating system that hosts the JVM.
- Another platform-dependent attribute of a traditional binary executable file is the byte order of integers. In executable binary files for the Intel X86 family of processors, for example, the byte order is *little-endian*. In a Java class file, byte order is big-endian irrespective of what platform generated the file and independent of whatever platforms may eventually use it.

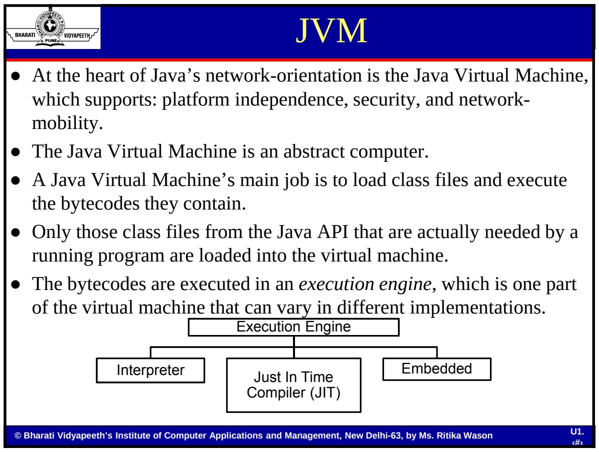
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



JAVA API

- The Java API helps make Java suitable for networks through its support for platform independence and security.
- The Java API is set of runtime libraries that give you a standard way to access the system resources of a host computer.
- When you write a Java program, you assume the class files of the Java API will be available at any Java Virtual Machine that may ever have the privilege of running your program. This is a safe assumption because the Java Virtual Machine and the class files for the Java API are the required components of any implementation of the Java Platform.
- The combination of all loaded class files (from your program and from the Java API) and any loaded dynamic libraries (containing native methods) constitute the full program executed by the Java Virtual Machine.

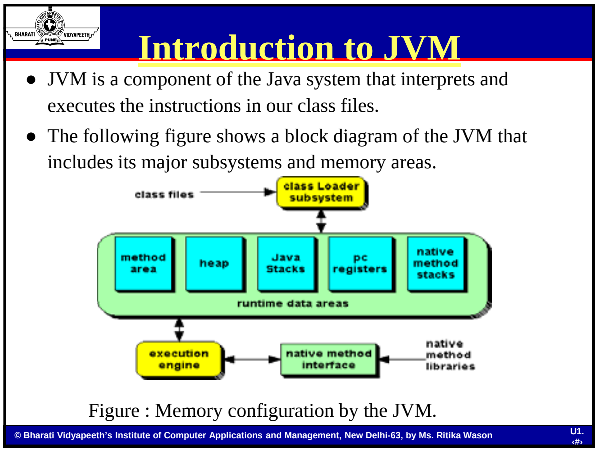
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



JVM

- At the heart of Java’s network-orientation is the Java Virtual Machine, which supports: platform independence, security, and network-mobility.
- The Java Virtual Machine is an abstract computer.
- A Java Virtual Machine’s main job is to load class files and execute the bytecodes they contain.
- Only those class files from the Java API that are actually needed by a running program are loaded into the virtual machine.
- The bytecodes are executed in an *execution engine*, which is one part of the virtual machine that can vary in different implementations.

U1.‹#›



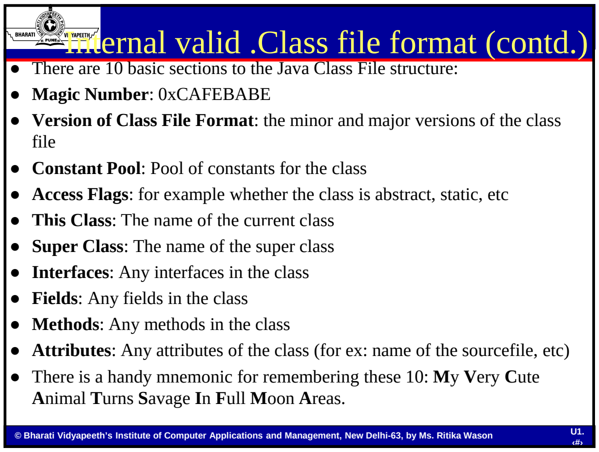
Introduction to JVM

- JVM is a component of the Java system that interprets and executes the instructions in our class files.
- The following figure shows a block diagram of the JVM that includes its major subsystems and memory areas.

class files
class Loader subsystem
method area
heap
Java Stacks
pc registers
native method stacks
runtime data areas
execution engine
native method interface
native method libraries

Figure : Memory configuration by the JVM.

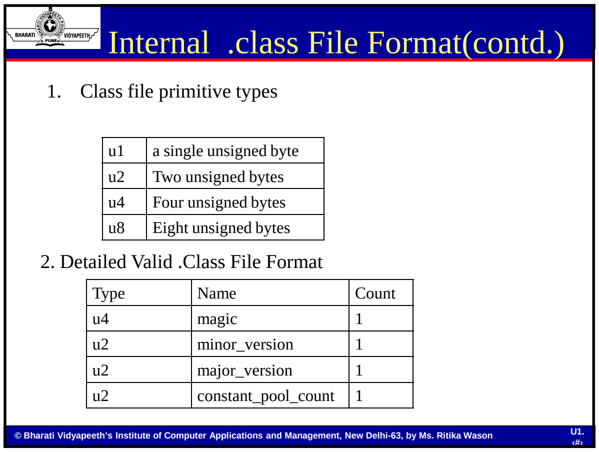
U1.‹#›



Internal valid .Class file format (contd.)

- There are 10 basic sections to the Java Class File structure:
- **Magic Number**: 0xCAFEBABE
- **Version of Class File Format**: the minor and major versions of the class file
- **Constant Pool**: Pool of constants for the class
- **Access Flags**: for example whether the class is abstract, static, etc
- **This Class**: The name of the current class
- **Super Class**: The name of the super class
- **Interfaces**: Any interfaces in the class
- **Fields**: Any fields in the class
- **Methods**: Any methods in the class
- **Attributes**: Any attributes of the class (for ex: name of the sourcefile, etc)
- There is a handy mnemonic for remembering these 10: **M**y **V**ery **C**ute **A**nimal **T**urns **S**avage **I**n **F**ull **M**oon **A**reas.

U1.‹#›



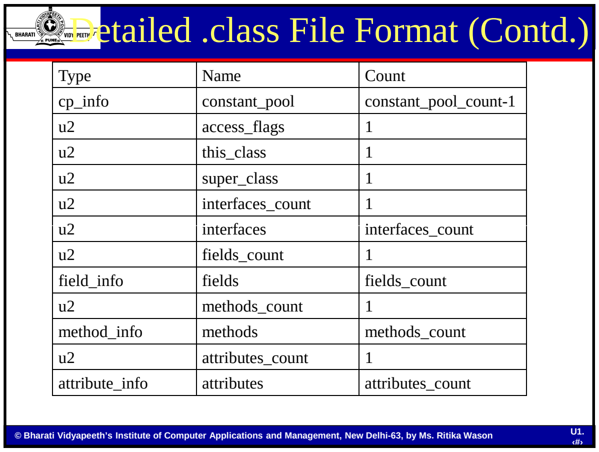
Internal .class File Format(contd.)

1. Class file primitive types

u1	a single unsigned byte
u2	Two unsigned bytes
u4	Four unsigned bytes
u8	Eight unsigned bytes

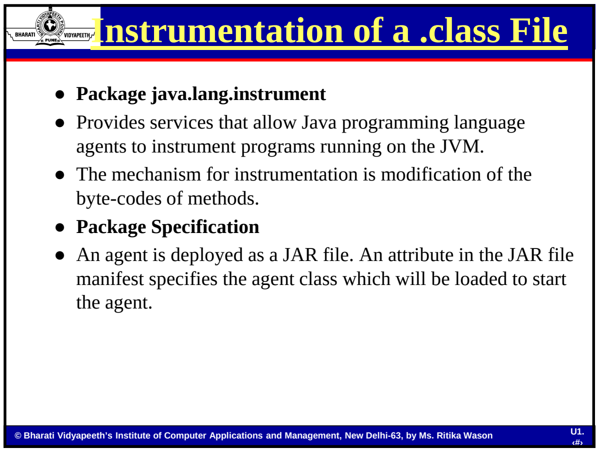
2. Detailed Valid .Class File Format

Type	Name	Count
u4	magic	1
u2	minor_version	1
u2	major_version	1
u2	constant_pool_count	1



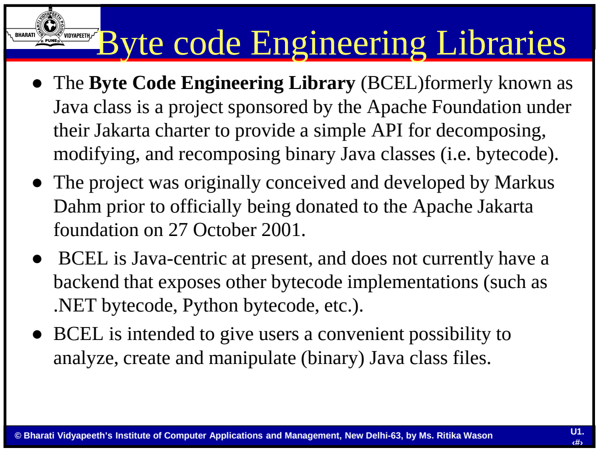
Detailed .class File Format (Contd.)

Type	Name	Count
cp_info	constant_pool	constant_pool_count-1
u2	access_flags	1
u2	this_class	1
u2	super_class	1
u2	interfaces_count	1
u2	interfaces	interfaces_count
u2	fields_count	1
field_info	fields	fields_count
u2	methods_count	1
method_info	methods	methods_count
u2	attributes_count	1
attribute_info	attributes	attributes_count



Instrumentation of a .class File

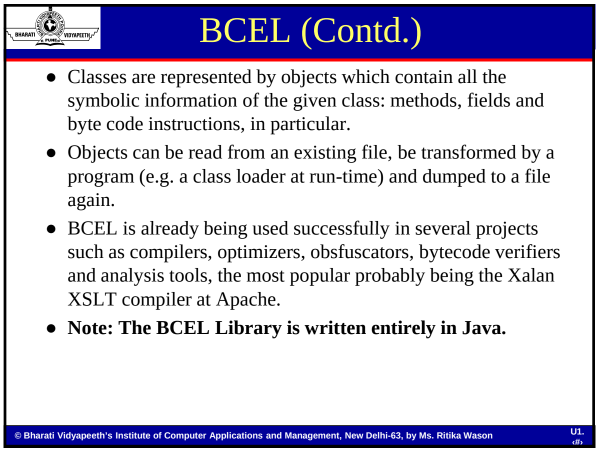
- **Package java.lang.instrument**
- Provides services that allow Java programming language agents to instrument programs running on the JVM.
- The mechanism for instrumentation is modification of the byte-codes of methods.
- **Package Specification**
- An agent is deployed as a JAR file. An attribute in the JAR file manifest specifies the agent class which will be loaded to start the agent.



Byte code Engineering Libraries

- The **Byte Code Engineering Library** (BCEL)formerly known as Java class is a project sponsored by the Apache Foundation under their Jakarta charter to provide a simple API for decomposing, modifying, and recomposing binary Java classes (i.e. bytecode).
- The project was originally conceived and developed by Markus Dahm prior to officially being donated to the Apache Jakarta foundation on 27 October 2001.
- BCEL is Java-centric at present, and does not currently have a backend that exposes other bytecode implementations (such as .NET bytecode, Python bytecode, etc.).
- BCEL is intended to give users a convenient possibility to analyze, create and manipulate (binary) Java class files.

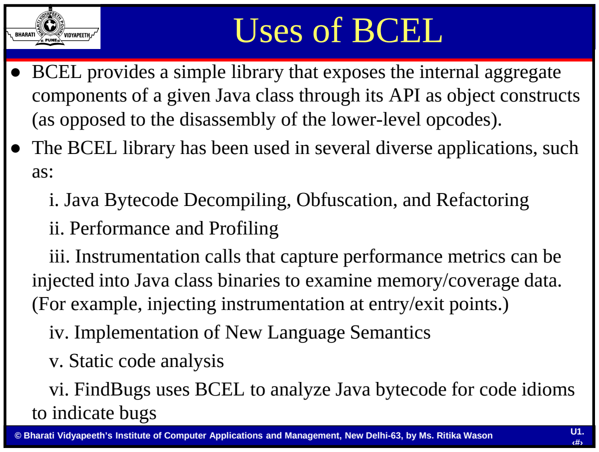
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



BCEL (Contd.)

- Classes are represented by objects which contain all the symbolic information of the given class: methods, fields and byte code instructions, in particular.
- Objects can be read from an existing file, be transformed by a program (e.g. a class loader at run-time) and dumped to a file again.
- BCEL is already being used successfully in several projects such as compilers, optimizers, obsfuscators, bytecode verifiers and analysis tools, the most popular probably being the Xalan XSLT compiler at Apache.
- **Note: The BCEL Library is written entirely in Java.**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Uses of BCEL

- BCEL provides a simple library that exposes the internal aggregate components of a given Java class through its API as object constructs (as opposed to the disassembly of the lower-level opcodes).
- The BCEL library has been used in several diverse applications, such as:

  i. Java Bytecode Decompiling, Obfuscation, and Refactoring

  ii. Performance and Profiling

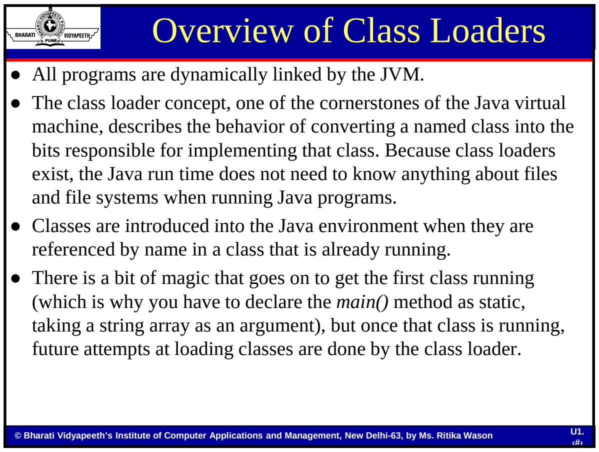
  iii. Instrumentation calls that capture performance metrics can be injected into Java class binaries to examine memory/coverage data. (For example, injecting instrumentation at entry/exit points.)

  iv. Implementation of New Language Semantics

  v. Static code analysis

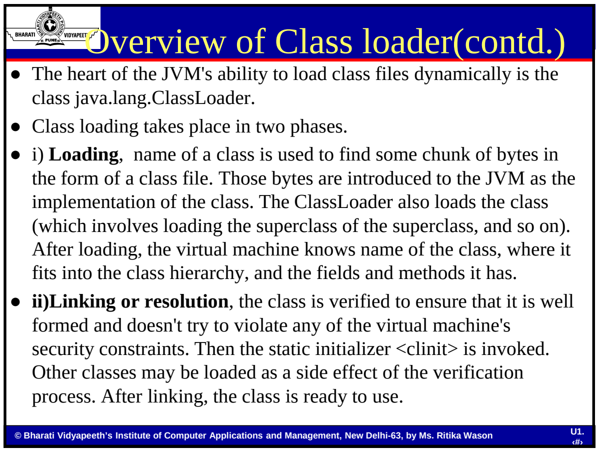
  vi. FindBugs uses BCEL to analyze Java bytecode for code idioms to indicate bugs

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



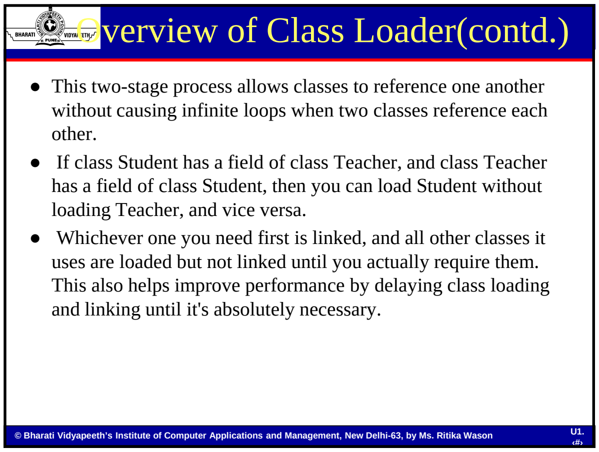
Overview of Class Loaders

- All programs are dynamically linked by the JVM.
- The class loader concept, one of the cornerstones of the Java virtual machine, describes the behavior of converting a named class into the bits responsible for implementing that class. Because class loaders exist, the Java run time does not need to know anything about files and file systems when running Java programs.
- Classes are introduced into the Java environment when they are referenced by name in a class that is already running.
- There is a bit of magic that goes on to get the first class running (which is why you have to declare the *main()* method as static, taking a string array as an argument), but once that class is running, future attempts at loading classes are done by the class loader.



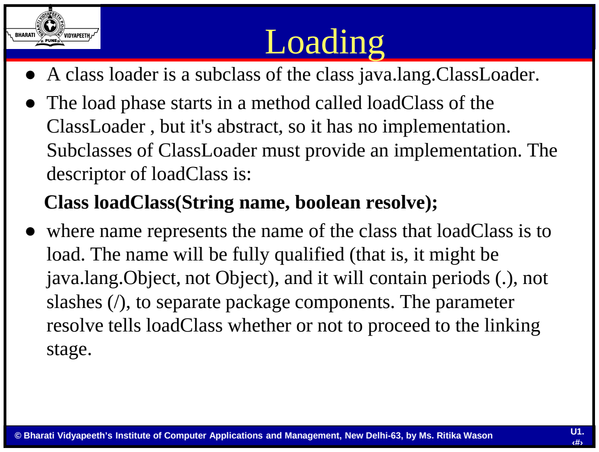
Overview of Class loader(contd.)

- The heart of the JVM's ability to load class files dynamically is the class java.lang.ClassLoader.
- Class loading takes place in two phases.
- i) **Loading**, name of a class is used to find some chunk of bytes in the form of a class file. Those bytes are introduced to the JVM as the implementation of the class. The ClassLoader also loads the class (which involves loading the superclass of the superclass, and so on). After loading, the virtual machine knows name of the class, where it fits into the class hierarchy, and the fields and methods it has.
- **ii)Linking or resolution**, the class is verified to ensure that it is well formed and doesn't try to violate any of the virtual machine's security constraints. Then the static initializer <clinit> is invoked. Other classes may be loaded as a side effect of the verification process. After linking, the class is ready to use.



Overview of Class Loader(contd.)

- This two-stage process allows classes to reference one another without causing infinite loops when two classes reference each other.
- If class Student has a field of class Teacher, and class Teacher has a field of class Student, then you can load Student without loading Teacher, and vice versa.
- Whichever one you need first is linked, and all other classes it uses are loaded but not linked until you actually require them. This also helps improve performance by delaying class loading and linking until it's absolutely necessary.

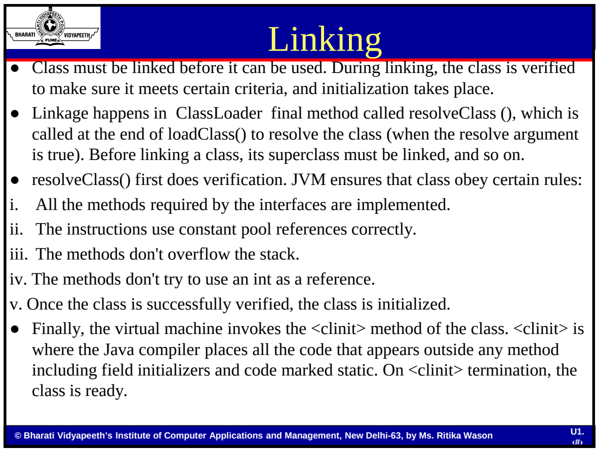


Loading

- A class loader is a subclass of the class java.lang.ClassLoader.
- The load phase starts in a method called loadClass of the ClassLoader , but it's abstract, so it has no implementation. Subclasses of ClassLoader must provide an implementation. The descriptor of loadClass is:

Class loadClass(String name, boolean resolve);

- where name represents the name of the class that loadClass is to load. The name will be fully qualified (that is, it might be java.lang.Object, not Object), and it will contain periods (.), not slashes (/), to separate package components. The parameter resolve tells loadClass whether or not to proceed to the linking stage.



Linking

- Class must be linked before it can be used. During linking, the class is verified to make sure it meets certain criteria, and initialization takes place.
- Linkage happens in ClassLoader final method called resolveClass (), which is called at the end of loadClass() to resolve the class (when the resolve argument is true). Before linking a class, its superclass must be linked, and so on.
- resolveClass() first does verification. JVM ensures that class obey certain rules:

i. All the methods required by the interfaces are implemented.

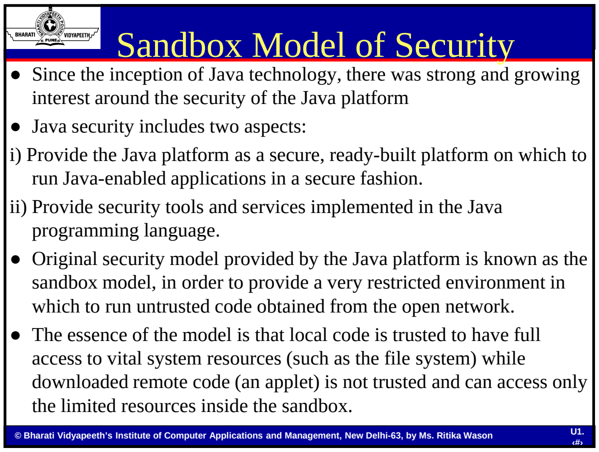
ii. The instructions use constant pool references correctly.

iii. The methods don't overflow the stack.

iv. The methods don't try to use an int as a reference.

v. Once the class is successfully verified, the class is initialized.

- Finally, the virtual machine invokes the <clinit> method of the class. <clinit> is where the Java compiler places all the code that appears outside any method including field initializers and code marked static. On <clinit> termination, the class is ready.



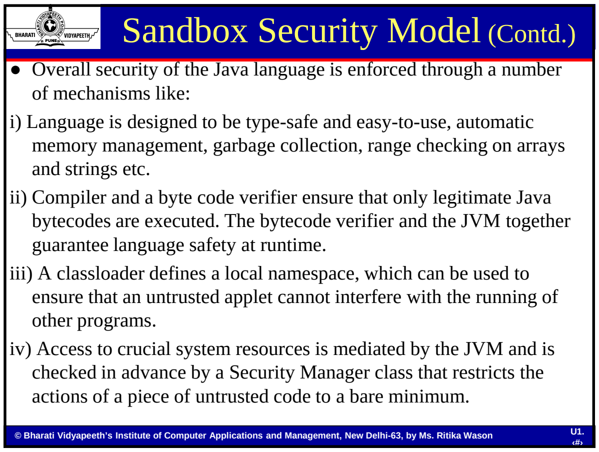
Sandbox Model of Security

- Since the inception of Java technology, there was strong and growing interest around the security of the Java platform
- Java security includes two aspects:

i) Provide the Java platform as a secure, ready-built platform on which to run Java-enabled applications in a secure fashion.

ii) Provide security tools and services implemented in the Java programming language.

- Original security model provided by the Java platform is known as the sandbox model, in order to provide a very restricted environment in which to run untrusted code obtained from the open network.
- The essence of the 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 limited resources inside the sandbox.



Sandbox Security Model (Contd.)

- Overall security of the Java language is enforced through a number of mechanisms like:

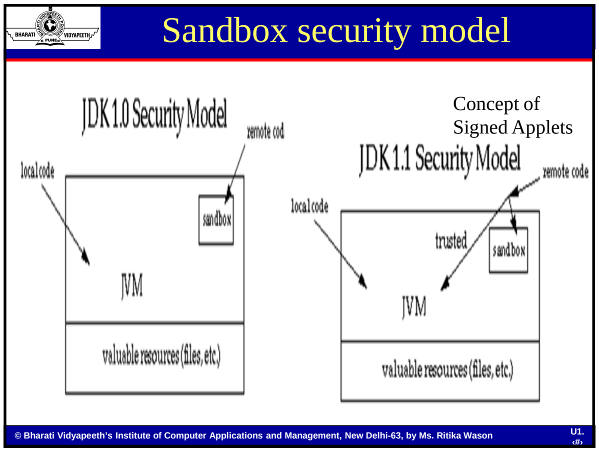
i) Language is designed to be type-safe and easy-to-use, automatic memory management, garbage collection, range checking on arrays and strings etc.

ii) Compiler and a byte code verifier ensure that only legitimate Java bytecodes are executed. The bytecode verifier and the JVM together guarantee language safety at runtime.

iii) A classloader defines a local namespace, which can be used to ensure that an untrusted applet cannot interfere with the running of other programs.

iv) Access to crucial system resources is mediated by the JVM and is checked in advance by a Security Manager class that restricts the actions of a piece of untrusted code to a bare minimum.

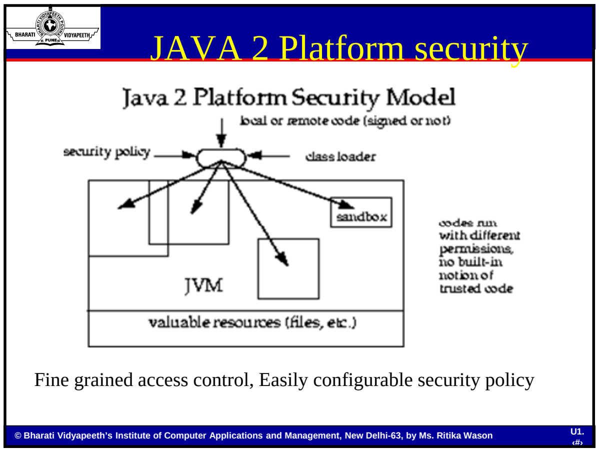
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Sandbox security model

Concept of Signed Applets

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›

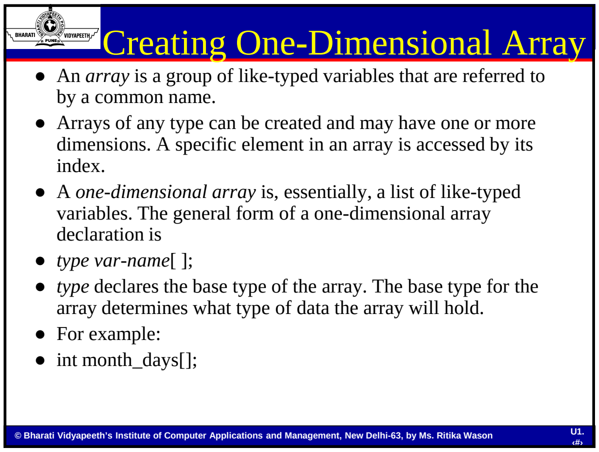


JAVA 2 Platform security

Java 2 Platform Security Model
local or remote code (signed or not)
security policy
class loader
sandbox
codes run with different permissions, no built-in notion of trusted code
JVM
valuable resources (files, etc.)

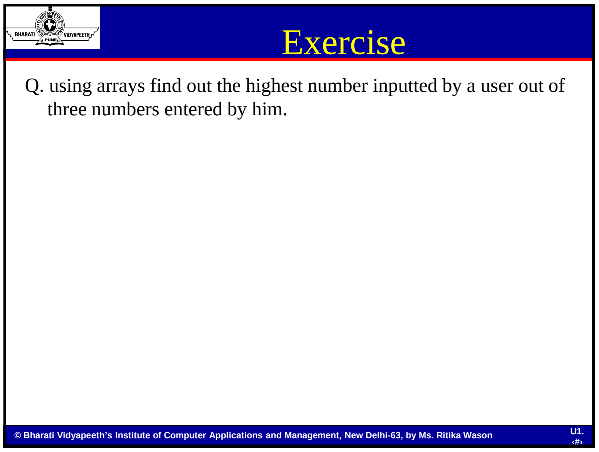
Fine grained access control, Easily configurable security policy

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



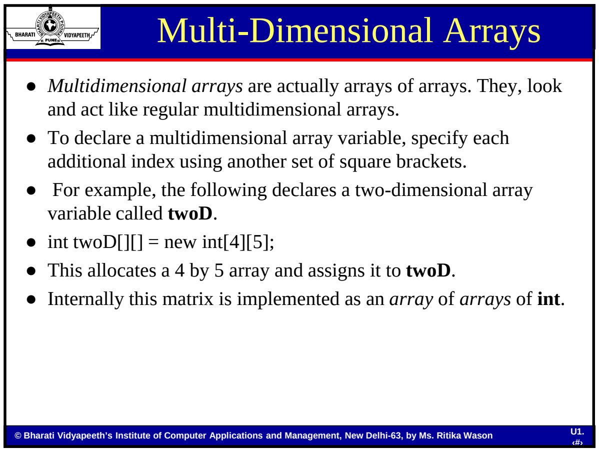
Creating One-Dimensional Array

- An *array* is a group of like-typed variables that are referred to by a common name.
- Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index.
- 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 for the array determines what type of data the array will hold.
- For example:
- int month_days[];



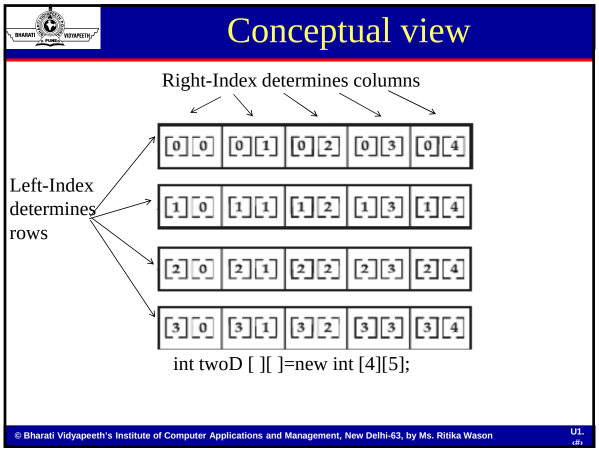
Exercise

Q. using arrays find out the highest number inputted by a user out of three numbers entered by him.



Multi-Dimensional Arrays

- *Multidimensional arrays* are actually arrays of arrays. They, look and act like regular multidimensional arrays.
- To declare a multidimensional array variable, specify each additional index using another set of square brackets.
- For example, the following declares a two-dimensional array variable called **twoD**.
- 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**.



Conceptual view

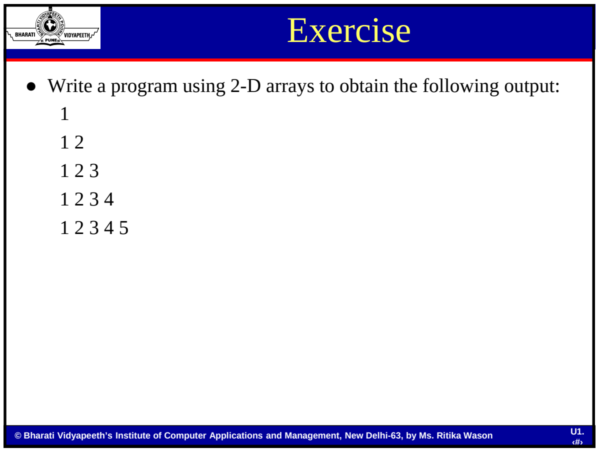
Right-Index determines columns

Left-Index determines rows

[0][0]	[0][1]	[0][2]	[0][3]	[0][4]
[1][0]	[1][1]	[1][2]	[1][3]	[1][4]
[2][0]	[2][1]	[2][2]	[2][3]	[2][4]
[3][0]	[3][1]	[3][2]	[3][3]	[3][4]

int twoD [ ][ ]=new int [4][5];

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›

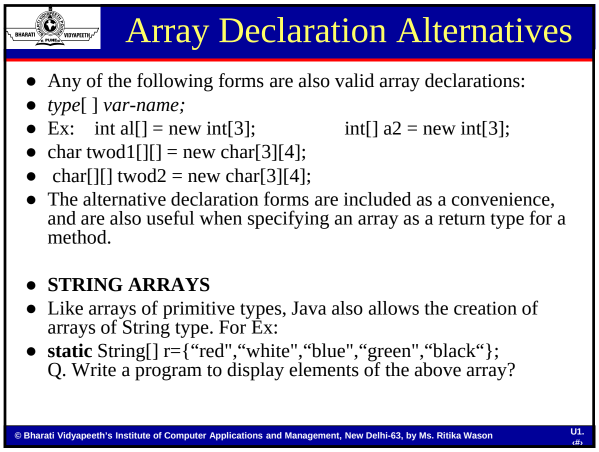


Exercise

- Write a program using 2-D arrays to obtain the following output:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Array Declaration Alternatives

- Any of the following forms are also valid array declarations:
- *type*[ ] *var-name;*
- Ex: int al[] = new int[3]; int[] a2 = new int[3];
- char twod1[][] = new char[3][4];
- char[][] twod2 = new char[3][4];
- The alternative declaration forms are included as a convenience, and are also useful when specifying an array as a return type for a method.

- **STRING ARRAYS**
- Like arrays of primitive types, Java also allows the creation of arrays of String type. For Ex:
- **static** String[] r={“red",“white",“blue",“green",“black“};
Q. Write a program to display elements of the above array?

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›

Exercise

Q. Create a class called test to have two integer type two-dimensional arrays. Write a function to perform addition of the two arrays, multiplication and then print the result?

- Q. Modify the above program to sort array elements input by the user?
- Q. Create a program to copy contents of one array into another using the following method from System class?

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

Class-Introduction

- Class is at the core of Java. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object.
- Class construct forms the basis for object-oriented programming in Java. Any concept you wish to implement in a Java program must be encapsulated within a class.
- A class is a *template* for an object, and an object is an *instance* of a class.
- **class *classname* { *type instance-variable1*;**

// ...

***type instance-variableN*;**

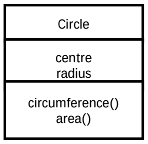
***type methodname1*(*parameter-list*) { // body of method }**

// ...

***type methodnameN*(*parameter-list*) {// body of method }}**

Classes

- A *class* is a collection of *fields* (data) and *methods* (procedure or function) that operate on that data.
- Most important point here is that a class defines a new data type that can be used to create objects of that type.

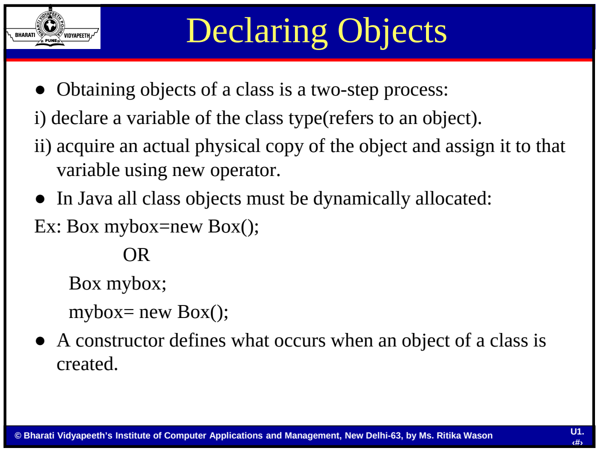


Circle

centre
radius

circumference()
area()

Figure: Conceptual Representation of Circle class



Declaring Objects

- Obtaining objects of a class is a two-step process:

i) declare a variable of the class type(refers to an object).

ii) acquire an actual physical copy of the object and assign it to that variable using new operator.

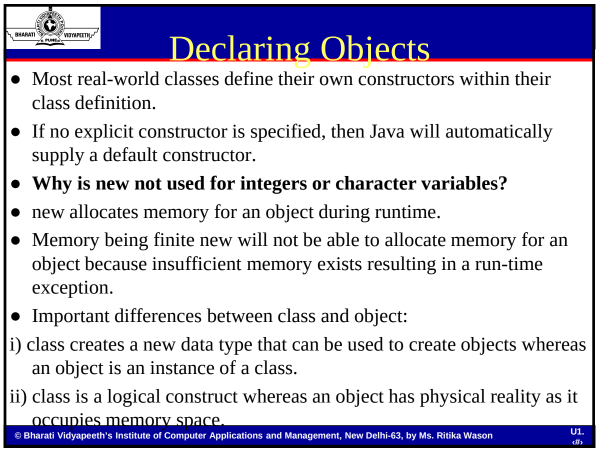
- In Java all class objects must be dynamically allocated:

Ex: Box mybox=new Box();

OR

Box mybox;
mybox= new Box();

- A constructor defines what occurs when an object of a class is created.

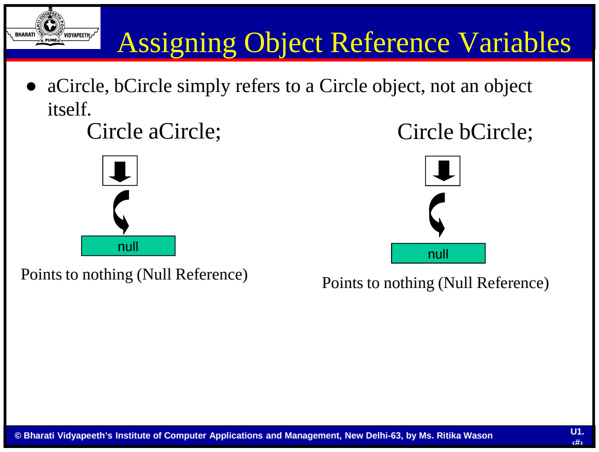


Declaring Objects

- Most real-world classes define their own constructors within their class definition.
- If no explicit constructor is specified, then Java will automatically supply a default constructor.
- **Why is new not used for integers or character variables?**
- new allocates memory for an object during runtime.
- Memory being finite new will not be able to allocate memory for an object because insufficient memory exists resulting in a run-time exception.
- Important differences between class and object:

i) class creates a new data type that can be used to create objects whereas an object is an instance of a class.

ii) class is a logical construct whereas an object has physical reality as it occupies memory space.



Assigning Object Reference Variables

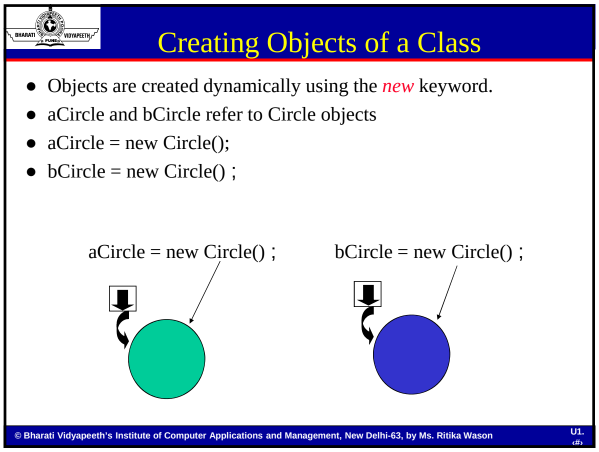
- aCircle, bCircle simply refers to a Circle object, not an object itself.

Circle aCircle;

Points to nothing (Null Reference)

Circle bCircle;

Points to nothing (Null Reference)



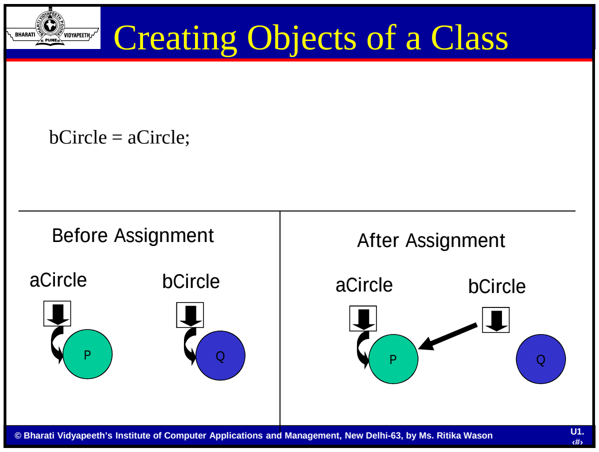
Creating Objects of a Class

- Objects are created dynamically using the *new* keyword.
- aCircle and bCircle refer to Circle objects
- aCircle = new Circle();
- bCircle = new Circle() ;

aCircle = new Circle() ;

bCircle = new Circle() ;

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Creating Objects of a Class

bCircle = aCircle;

Before Assignment

aCircle bCircle

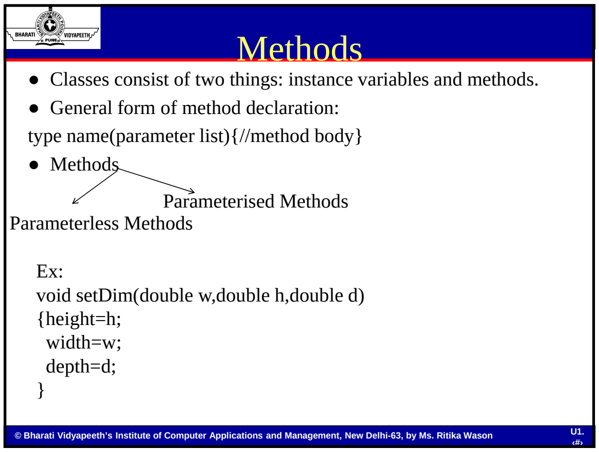
P Q

After Assignment

aCircle bCircle

P Q

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Methods

- Classes consist of two things: instance variables and methods.
- General form of method declaration:

type name(parameter list){//method body}

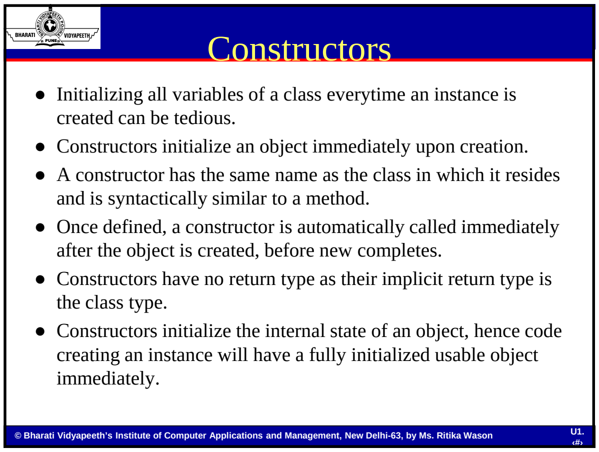
- Methods

Parameterised Methods

Parameterless Methods

```
Ex:
void setDim(double w,double h,double d)
{height=h;
 width=w;
 depth=d;
}
```

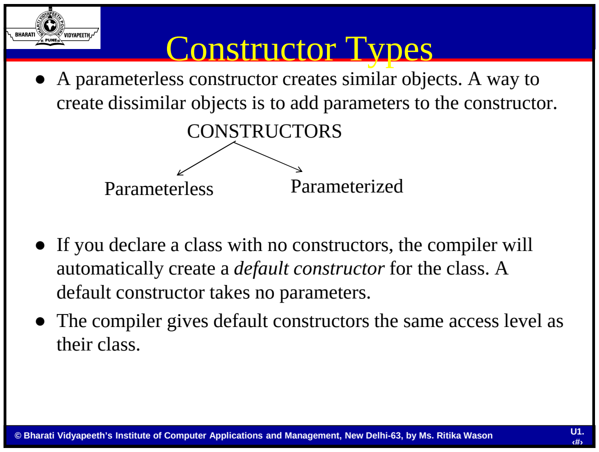
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Constructors

- Initializing all variables of a class everytime an instance is created can be tedious.
- Constructors initialize an object immediately upon creation.
- A constructor has the same name as the class in which it resides and is syntactically similar to a method.
- Once defined, a constructor is automatically called immediately after the object is created, before new completes.
- Constructors have no return type as their implicit return type is the class type.
- Constructors initialize the internal state of an object, hence code creating an instance will have a fully initialized usable object immediately.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Constructor Types

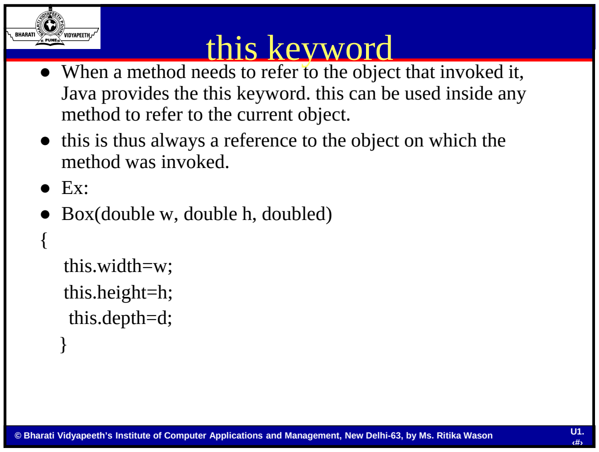
- A parameterless constructor creates similar objects. A way to create dissimilar objects is to add parameters to the constructor.

CONSTRUCTORS

Parameterless Parameterized

- If you declare a class with no constructors, the compiler will automatically create a *default constructor* for the class. A default constructor takes no parameters.
- The compiler gives default constructors the same access level as their class.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›

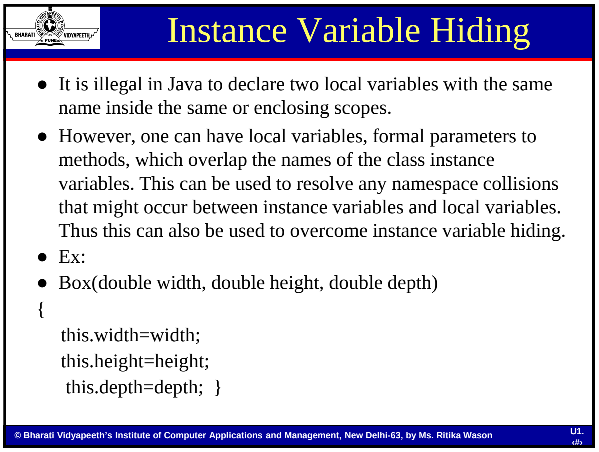


this keyword

- When a method needs to refer to the object that invoked it, Java provides the this keyword. this can be used inside any method to refer to the current object.
- this is thus always a reference to the object on which the method was invoked.
- Ex:
- Box(double w, double h, doubled)

```
{
  this.width=w;
  this.height=h;
   this.depth=d;
  }
```

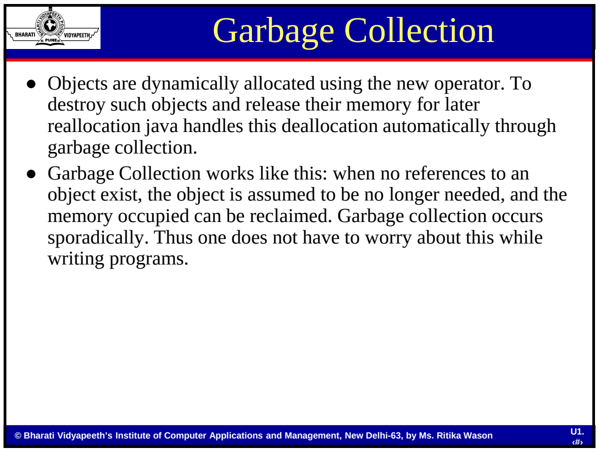
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Instance Variable Hiding

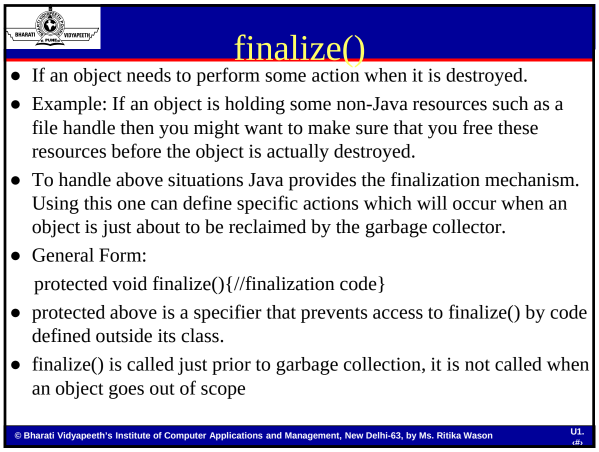
- It is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes.
- However, one can have local variables, formal parameters to methods, which overlap the names of the class instance variables. This can be used to resolve any namespace collisions that might occur between instance variables and local variables. Thus this can also be used to overcome instance variable hiding.
- Ex:
- Box(double width, double height, double depth)

```
{
     this.width=width;
     this.height=height;
      this.depth=depth;  }
```



Garbage Collection

- Objects are dynamically allocated using the new operator. To destroy such objects and release their memory for later reallocation java handles this deallocation automatically through garbage collection.
- Garbage Collection works like this: when no references to an object exist, the object is assumed to be no longer needed, and the memory occupied can be reclaimed. Garbage collection occurs sporadically. Thus one does not have to worry about this while writing programs.

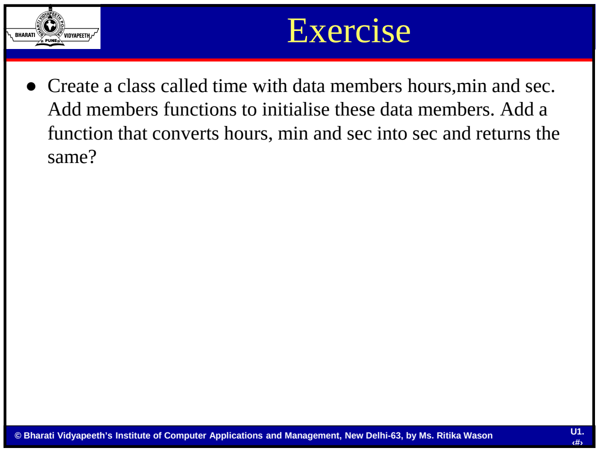


finalize()

- If an object needs to perform some action when it is destroyed.
- Example: If an object is holding some non-Java resources such as a file handle then you might want to make sure that you free these resources before the object is actually destroyed.
- To handle above situations Java provides the finalization mechanism. Using this one can define specific actions which will occur when an object is just about to be reclaimed by the garbage collector.
- General Form:

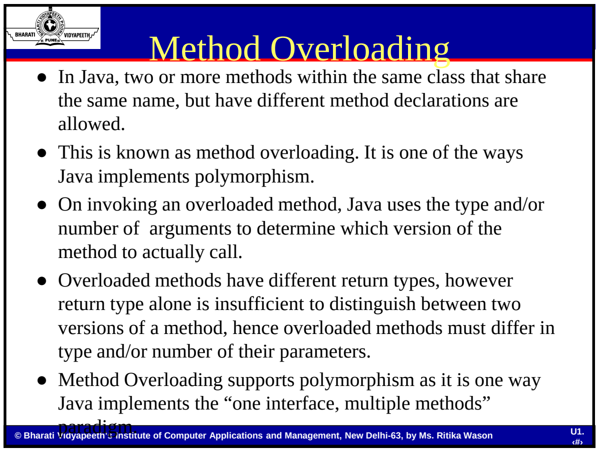
```
protected void finalize(){//finalization code}
```

- protected above is a specifier that prevents access to finalize() by code defined outside its class.
- finalize() is called just prior to garbage collection, it is not called when an object goes out of scope



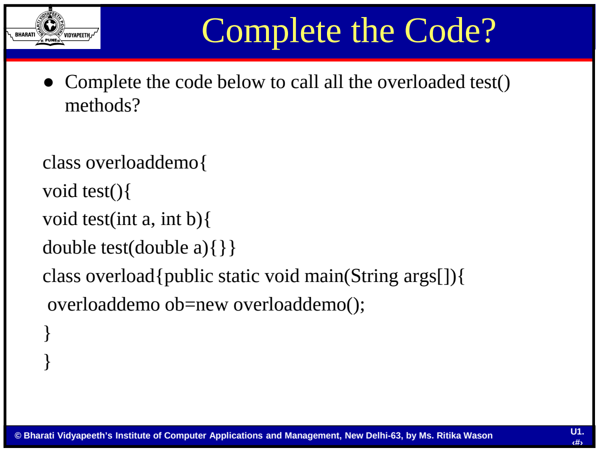
Exercise

- Create a class called time with data members hours,min and sec. Add members functions to initialise these data members. Add a function that converts hours, min and sec into sec and returns the same?



Method Overloading

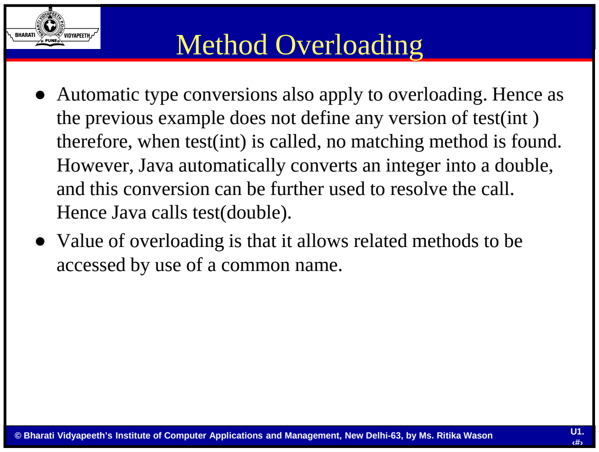
- In Java, two or more methods within the same class that share the same name, but have different method declarations are allowed.
- This is known as method overloading. It is one of the ways Java implements polymorphism.
- On invoking an overloaded method, Java uses the type and/or number of arguments to determine which version of the method to actually call.
- Overloaded methods have different return types, however return type alone is insufficient to distinguish between two versions of a method, hence overloaded methods must differ in type and/or number of their parameters.
- Method Overloading supports polymorphism as it is one way Java implements the "one interface, multiple methods" paradigm.



Complete the Code?

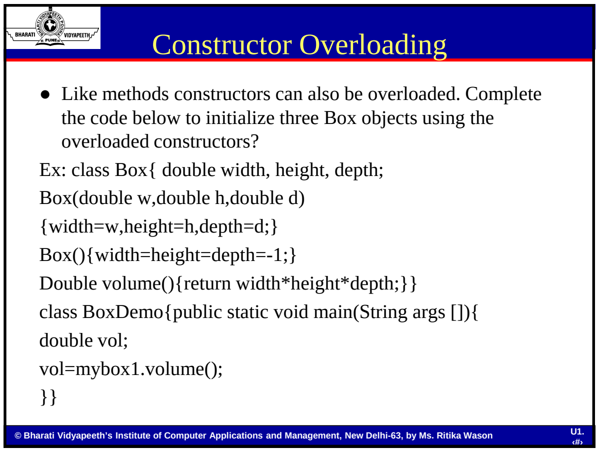
- Complete the code below to call all the overloaded test() methods?

```
class overloaddemo{
void test(){
void test(int a, int b){
double test(double a){}}
class overload{public static void main(String args[]){
 overloaddemo ob=new overloaddemo();
}
}
```



Method Overloading

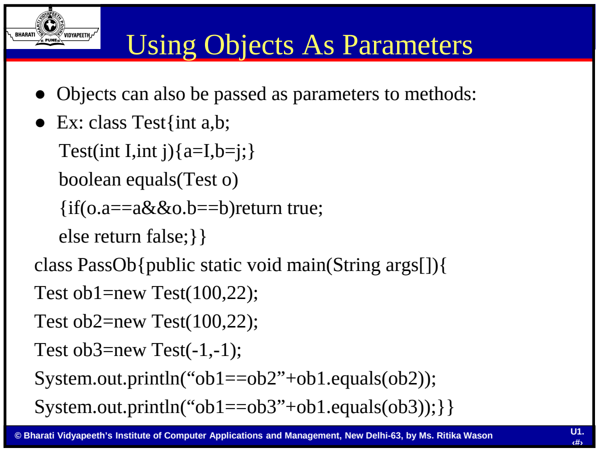
- Automatic type conversions also apply to overloading. Hence as the previous example does not define any version of test(int) therefore, when test(int) is called, no matching method is found. However, Java automatically converts an integer into a double, and this conversion can be further used to resolve the call. Hence Java calls test(double).
- Value of overloading is that it allows related methods to be accessed by use of a common name.



Constructor Overloading

- Like methods constructors can also be overloaded. Complete the code below to initialize three Box objects using the overloaded constructors?

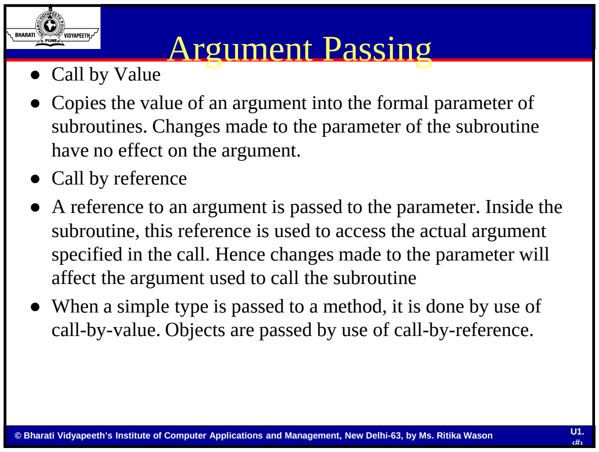
```
Ex: class Box{ double width, height, depth;
Box(double w,double h,double d)
{width=w,height=h,depth=d;}
Box(){width=height=depth=-1;}
Double volume(){return width*height*depth;}}
class BoxDemo{public static void main(String args []){
double vol;
vol=mybox1.volume();
}}
```



Using Objects As Parameters

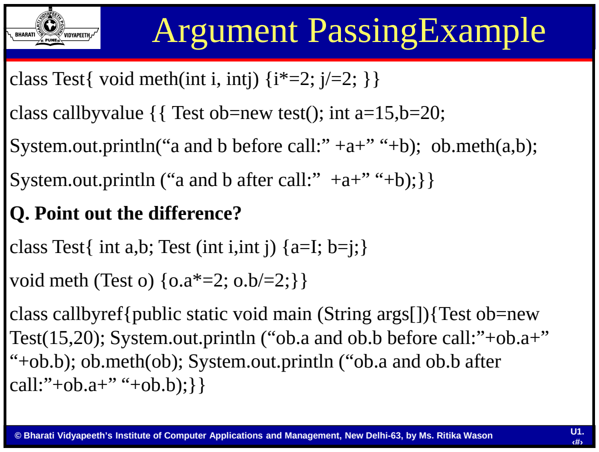
- Objects can also be passed as parameters to methods:
- Ex: class Test{int a,b;

```
 Test(int I,int j){a=I,b=j;}
 boolean equals(Test o)
 {if(o.a==a&&o.b==b)return true;
 else return false;}}
class PassOb{public static void main(String args[]){
Test ob1=new Test(100,22);
Test ob2=new Test(100,22);
Test ob3=new Test(-1,-1);
System.out.println(“ob1==ob2”+ob1.equals(ob2));
System.out.println(“ob1==ob3”+ob1.equals(ob3));}}
```



Argument Passing

- Call by Value
- Copies the value of an argument into the formal parameter of subroutines. Changes made to the parameter of the subroutine have no effect on the argument.
- Call by reference
- A reference to an argument is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. Hence changes made to the parameter will affect the argument used to call the subroutine
- When a simple type is passed to a method, it is done by use of call-by-value. Objects are passed by use of call-by-reference.



Argument PassingExample

class Test{ void meth(int i, intj) {i*=2; j/=2; }}

class callbyvalue {{ Test ob=new test(); int a=15,b=20;

System.out.println(“a and b before call:” +a+” “+b); ob.meth(a,b);

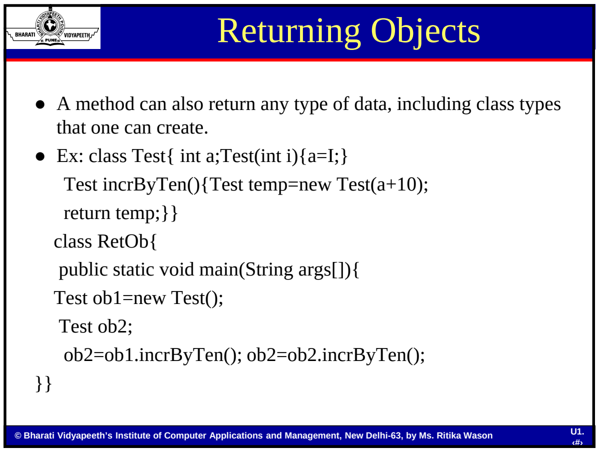
System.out.println (“a and b after call:” +a+” “+b);}}

Q. Point out the difference?

class Test{ int a,b; Test (int i,int j) {a=I; b=j;}

void meth (Test o) {o.a*=2; o.b/=2;}}

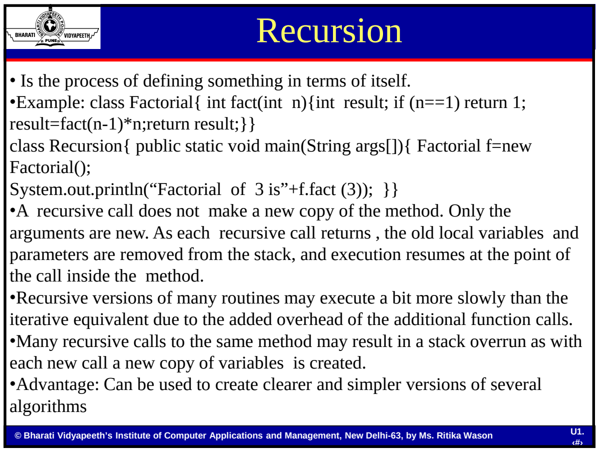
class callbyref{public static void main (String args[]){Test ob=new Test(15,20); System.out.println (“ob.a and ob.b before call:”+ob.a+” “+ob.b); ob.meth(ob); System.out.println (“ob.a and ob.b after call:”+ob.a+” “+ob.b);}}



Returning Objects

- A method can also return any type of data, including class types that one can create.
- Ex: class Test{ int a;Test(int i){a=I;}

```
  Test incrByTen(){Test temp=new Test(a+10);
  return temp;}}
class RetOb{
 public static void main(String args[]){
Test ob1=new Test();
 Test ob2;
  ob2=ob1.incrByTen(); ob2=ob2.incrByTen();
}}
```



Recursion

• Is the process of defining something in terms of itself.

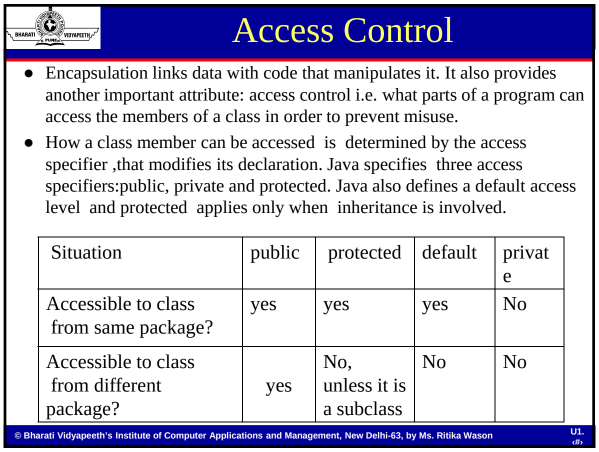
•Example: class Factorial{ int fact(int n){int result; if (n==1) return 1; result=fact(n-1)*n;return result;}}
class Recursion{ public static void main(String args[]){ Factorial f=new Factorial();
System.out.println(“Factorial of 3 is”+f.fact (3)); }}

•A recursive call does not make a new copy of the method. Only the arguments are new. As each recursive call returns , the old local variables and parameters are removed from the stack, and execution resumes at the point of the call inside the method.

•Recursive versions of many routines may execute a bit more slowly than the iterative equivalent due to the added overhead of the additional function calls.

•Many recursive calls to the same method may result in a stack overrun as with each new call a new copy of variables is created.

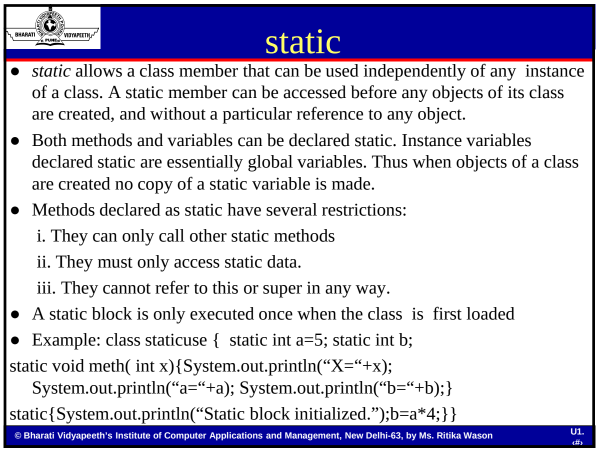
•Advantage: Can be used to create clearer and simpler versions of several algorithms



Access Control

- Encapsulation links data with code that manipulates it. It also provides another important attribute: access control i.e. what parts of a program can access the members of a class in order to prevent misuse.
- How a class member can be accessed is determined by the access specifier ,that modifies its declaration. Java specifies three access specifiers:public, private and protected. Java also defines a default access level and protected applies only when inheritance is involved.

Situation	public	protected	default	private
Accessible to class from same package?	yes	yes	yes	No
Accessible to class from different package?	yes	No, unless it is a subclass	No	No



static

- *static* allows a class member that can be used independently of any instance of a class. A static member can be accessed before any objects of its class are created, and without a particular reference to any object.
- Both methods and variables can be declared static. Instance variables declared static are essentially global variables. Thus when objects of a class are created no copy of a static variable is made.
- Methods declared as static have several restrictions:

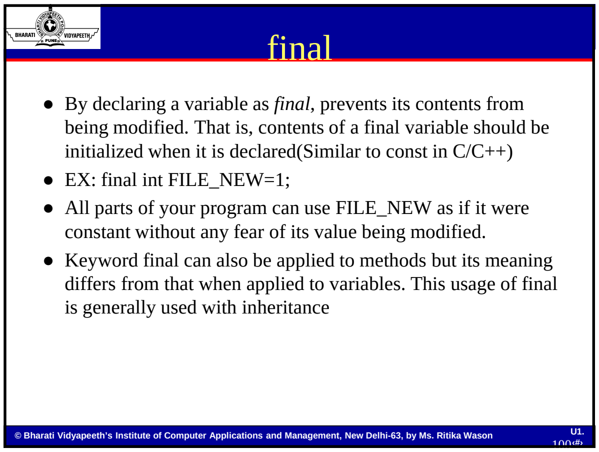
  i. They can only call other static methods

  ii. They must only access static data.

  iii. They cannot refer to this or super in any way.
- A static block is only executed once when the class is first loaded
- Example: class staticuse { static int a=5; static int b;

static void meth(int x){System.out.println(“X=“+x);
System.out.println(“a=“+a); System.out.println(“b=“+b);}

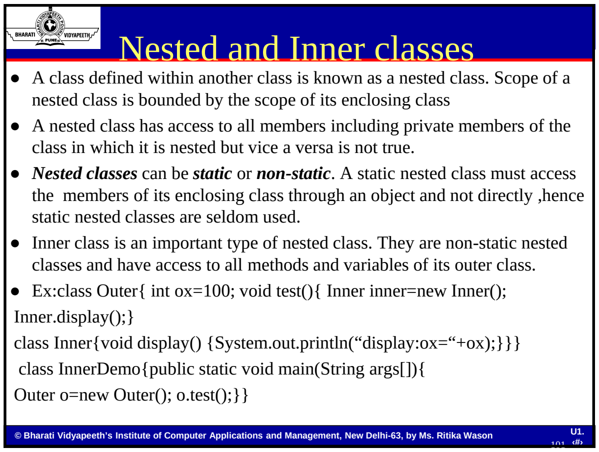
static{System.out.println(“Static block initialized.”);b=a*4;}}



final

- By declaring a variable as *final*, prevents its contents from being modified. That is, contents of a final variable should be initialized when it is declared(Similar to const in C/C++)
- EX: final int FILE_NEW=1;
- All parts of your program can use FILE_NEW as if it were constant without any fear of its value being modified.
- Keyword final can also be applied to methods but its meaning differs from that when applied to variables. This usage of final is generally used with inheritance

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1. 100‹#›

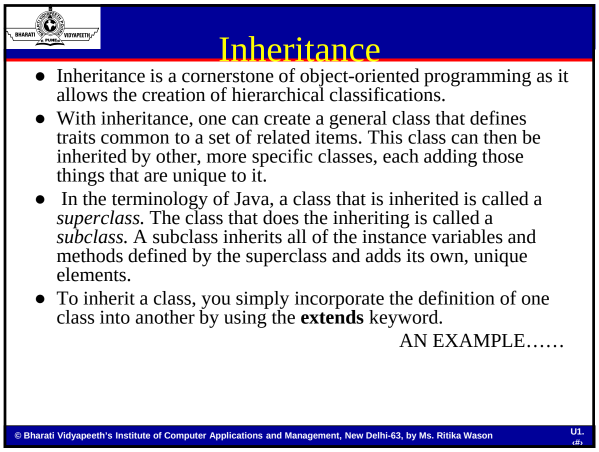


Nested and Inner classes

- A class defined within another class is known as a nested class. Scope of a nested class is bounded by the scope of its enclosing class
- A nested class has access to all members including private members of the class in which it is nested but vice a versa is not true.
- ***Nested classes*** can be ***static*** or ***non-static***. A static nested class must access the members of its enclosing class through an object and not directly ,hence static nested classes are seldom used.
- Inner class is an important type of nested class. They are non-static nested classes and have access to all methods and variables of its outer class.
- Ex:class Outer{ int ox=100; void test(){ Inner inner=new Inner();

```
Inner.display();}
class Inner{void display() {System.out.println(“display:ox=“+ox);}}}
 class InnerDemo{public static void main(String args[]){
Outer o=new Outer(); o.test();}}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1. 101 ‹#›

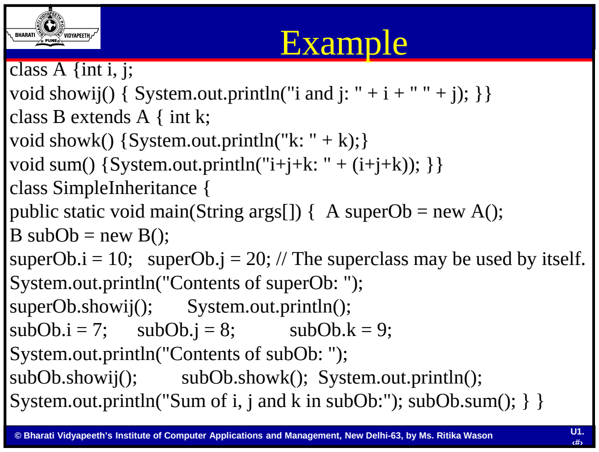


Inheritance

- Inheritance is a cornerstone of object-oriented programming as it allows the creation of hierarchical classifications.
- With inheritance, one can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it.
- In the terminology of Java, a class that is inherited is called a *superclass*. The class that does the inheriting is called a *subclass*. A subclass inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.
- To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword.

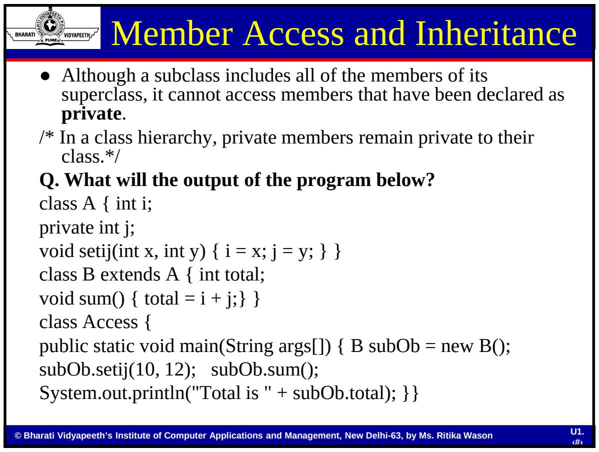
AN EXAMPLE……

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1. ‹#›



Example

```
class A {int i, j;
void showij() { System.out.println("i and j: " + i + " " + j); }}
class B extends A { int k;
void showk() {System.out.println("k: " + k);}
void sum() {System.out.println("i+j+k: " + (i+j+k)); }}
class SimpleInheritance {
public static void main(String args[]) {  A superOb = new A();
B subOb = new B();
superOb.i = 10;   superOb.j = 20; // The superclass may be used by itself.
System.out.println("Contents of superOb: ");
superOb.showij();      System.out.println();
subOb.i = 7;      subOb.j = 8;          subOb.k = 9;
System.out.println("Contents of subOb: ");
subOb.showij();        subOb.showk();  System.out.println();
System.out.println("Sum of i, j and k in subOb:"); subOb.sum(); } }
```



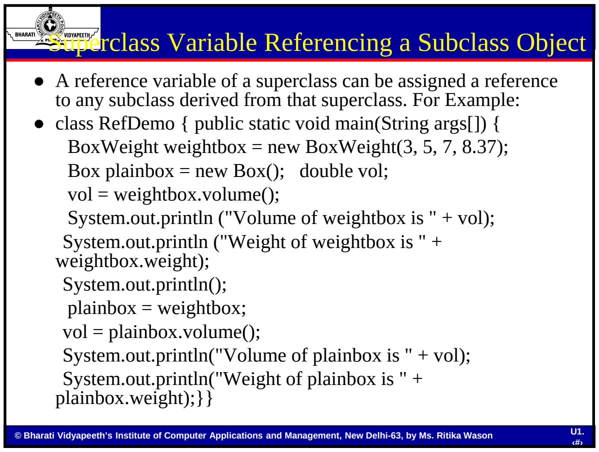
Member Access and Inheritance

- Although a subclass includes all of the members of its superclass, it cannot access members that have been declared as **private**.

/* In a class hierarchy, private members remain private to their class.*/

Q. What will the output of the program below?

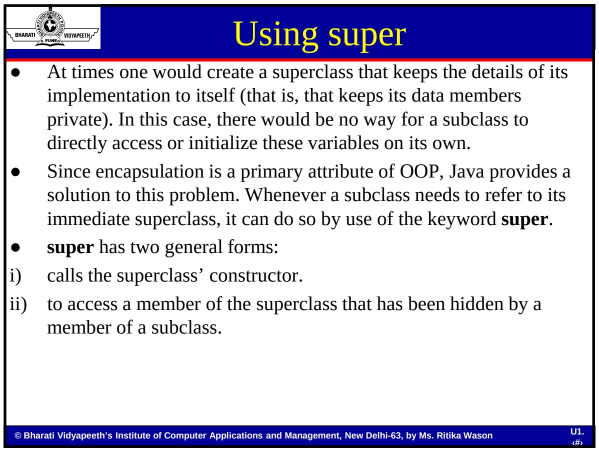
```
class A { int i;
private int j;
void setij(int x, int y) { i = x; j = y; } }
class B extends A { int total;
void sum() { total = i + j;} }
class Access {
public static void main(String args[]) { B subOb = new B();
subOb.setij(10, 12);   subOb.sum();
System.out.println("Total is " + subOb.total); }}
```



Superclass Variable Referencing a Subclass Object

- A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass. For Example:
- class RefDemo { public static void main(String args[]) {

```
  BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
  Box plainbox = new Box();   double vol;
  vol = weightbox.volume();
  System.out.println ("Volume of weightbox is " + vol);
 System.out.println ("Weight of weightbox is " +
weightbox.weight);
 System.out.println();
  plainbox = weightbox;
 vol = plainbox.volume();
 System.out.println("Volume of plainbox is " + vol);
 System.out.println("Weight of plainbox is " +
plainbox.weight);}}
```



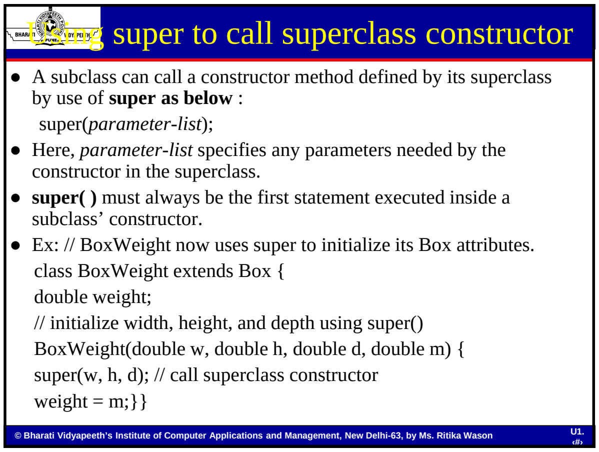
Using super

- At times one would create a superclass that keeps the details of its implementation to itself (that is, that keeps its data members private). In this case, there would be no way for a subclass to directly access or initialize these variables on its own.
- Since encapsulation is a primary attribute of OOP, Java provides a solution to this problem. Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.
- **super** has two general forms:

i) calls the superclass' constructor.

ii) to access a member of the superclass that has been hidden by a member of a subclass.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



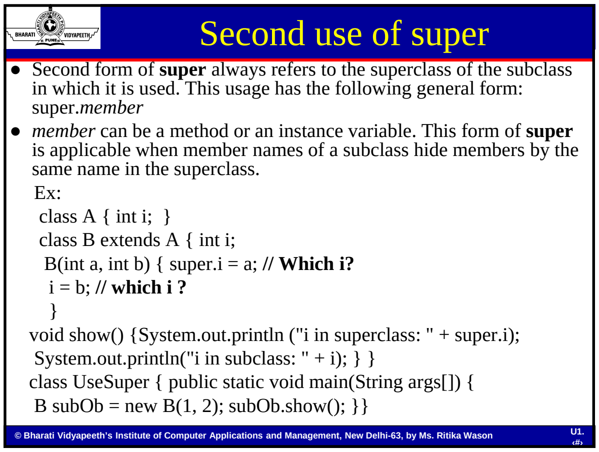
Using super to call superclass constructor

- A subclass can call a constructor method defined by its superclass by use of **super as below** :

  super(*parameter-list*);
- Here, *parameter-list* specifies any parameters needed by the constructor in the superclass.
- **super( )** must always be the first statement executed inside a subclass' constructor.
- Ex: // BoxWeight now uses super to initialize its Box attributes.

```
class BoxWeight extends Box {
double weight;
// initialize width, height, and depth using super()
BoxWeight(double w, double h, double d, double m) {
super(w, h, d); // call superclass constructor
weight = m;}}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›

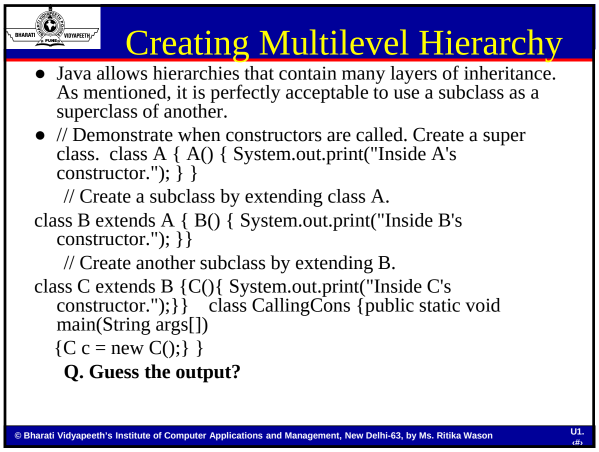


Second use of super

- Second form of **super** always refers to the superclass of the subclass in which it is used. This usage has the following general form: super.*member*
- *member* can be a method or an instance variable. This form of **super** is applicable when member names of a subclass hide members by the same name in the superclass.

```
 Ex:
  class A { int i;  }
  class B extends A { int i;
   B(int a, int b) { super.i = a; // Which i?
    i = b; // which i ?
    }
void show() {System.out.println ("i in superclass: " + super.i);
 System.out.println("i in subclass: " + i); } }
class UseSuper { public static void main(String args[]) {
 B subOb = new B(1, 2); subOb.show(); }}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Creating Multilevel Hierarchy

- Java allows hierarchies that contain many layers of inheritance. As mentioned, it is perfectly acceptable to use a subclass as a superclass of another.
- // Demonstrate when constructors are called. Create a super class. class A { A() { System.out.print("Inside A's constructor."); } }

  // Create a subclass by extending class A.

class B extends A { B() { System.out.print("Inside B's constructor."); }}

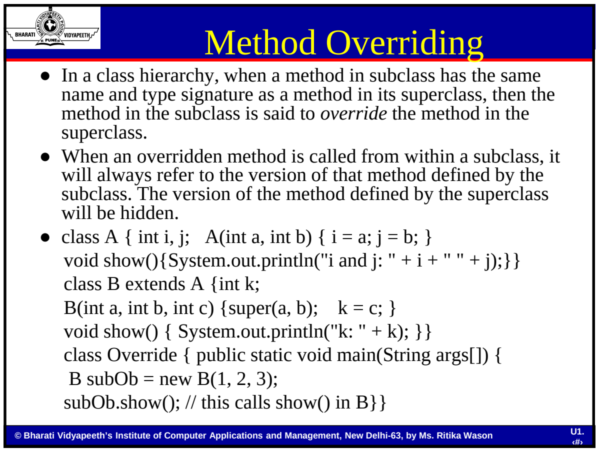
  // Create another subclass by extending B.

class C extends B {C(){ System.out.print("Inside C's constructor.");}} class CallingCons {public static void main(String args[])

{C c = new C();} }

Q. Guess the output?

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Method Overriding

- In a class hierarchy, when a method in subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass.
- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.
- class A { int i, j; A(int a, int b) { i = a; j = b; }

  void show(){System.out.println("i and j: " + i + " " + j);}}

  class B extends A {int k;

  B(int a, int b, int c) {super(a, b); k = c; }

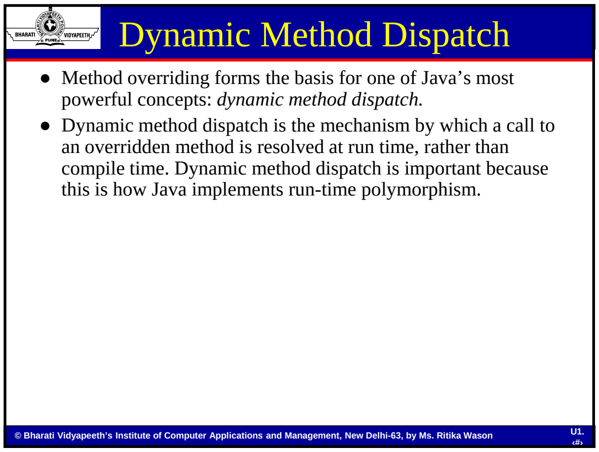
  void show() { System.out.println("k: " + k); }}

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

  B subOb = new B(1, 2, 3);

  subOb.show(); // this calls show() in B}}

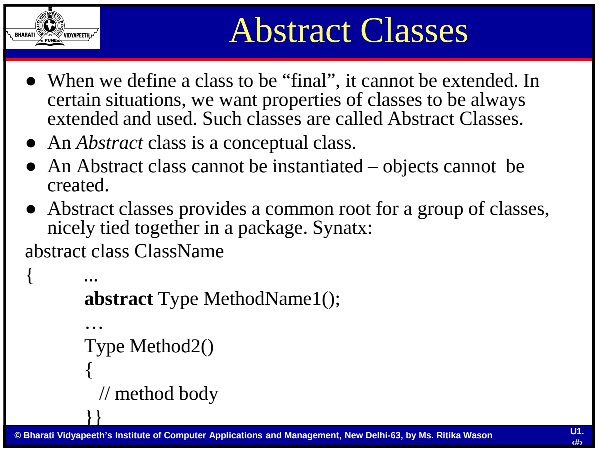
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Dynamic Method Dispatch

- Method overriding forms the basis for one of Java's most powerful concepts: *dynamic method dispatch.*
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›

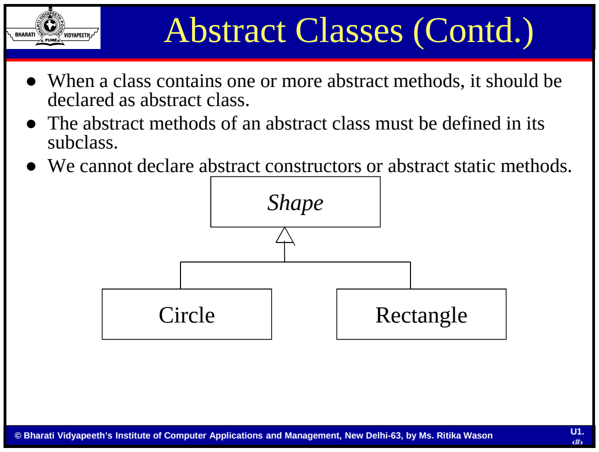


Abstract Classes

- When we define a class to be "final", it cannot be extended. In certain situations, we want properties of classes to be always extended and used. Such classes are called Abstract Classes.
- An *Abstract* class is a conceptual class.
- An Abstract class cannot be instantiated – objects cannot be created.
- Abstract classes provides a common root for a group of classes, nicely tied together in a package. Synatx:

```
abstract class ClassName
{         ...
          abstract Type MethodName1();
          …
          Type Method2()
          {
             // method body
          }}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Abstract Classes (Contd.)

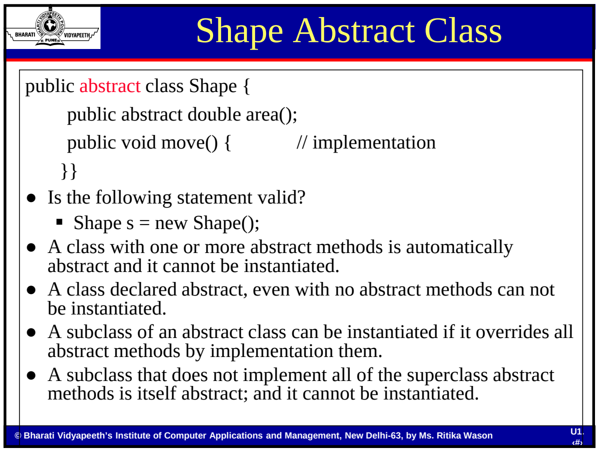
- When a class contains one or more abstract methods, it should be declared as abstract class.
- The abstract methods of an abstract class must be defined in its subclass.
- We cannot declare abstract constructors or abstract static methods.

Shape

Circle

Rectangle

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›

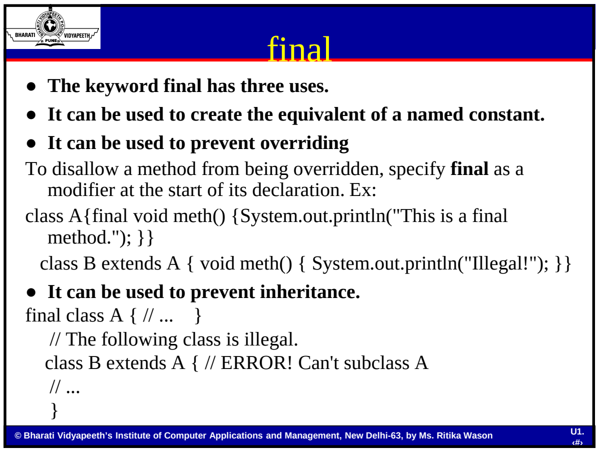


Shape Abstract Class

```
public abstract class Shape {
      public abstract double area();
      public void move() {          // implementation
     }}
```

- Is the following statement valid?
  - Shape s = new Shape();
- A class with one or more abstract methods is automatically abstract and it cannot be instantiated.
- A class declared abstract, even with no abstract methods can not be instantiated.
- A subclass of an abstract class can be instantiated if it overrides all abstract methods by implementation them.
- A subclass that does not implement all of the superclass abstract methods is itself abstract; and it cannot be instantiated.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



final

- **The keyword final has three uses.**
- **It can be used to create the equivalent of a named constant.**
- **It can be used to prevent overriding**

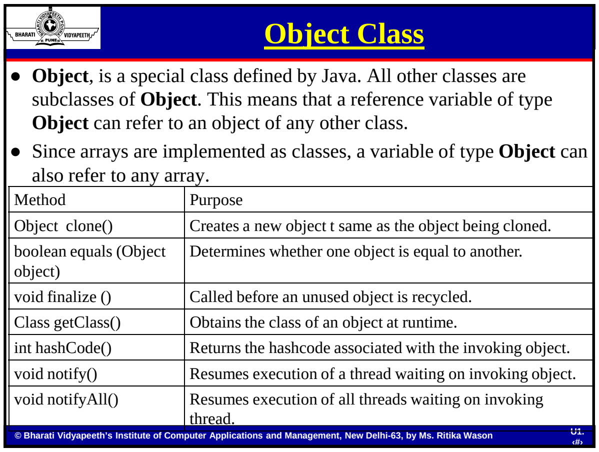
To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Ex:

```
class A{final void meth() {System.out.println("This is a final
    method."); }}
  class B extends A { void meth() { System.out.println("Illegal!"); }}
```

- **It can be used to prevent inheritance.**

```
final class A { // ...    }
    // The following class is illegal.
   class B extends A { // ERROR! Can't subclass A
    // ...
    }
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›

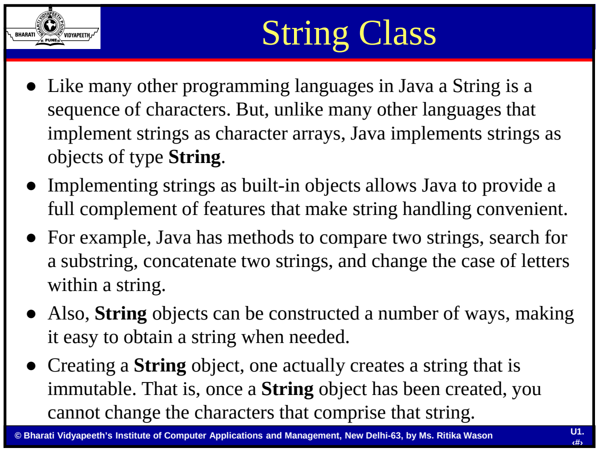


Object Class

- **Object**, is a special class defined by Java. All other classes are subclasses of **Object**. This means that a reference variable of type **Object** can refer to an object of any other class.
- Since arrays are implemented as classes, a variable of type **Object** can also refer to any array.

Method	Purpose
Object clone()	Creates a new object t same as the object being cloned.
boolean equals (Object object)	Determines whether one object is equal to another.
void finalize ()	Called before an unused object is recycled.
Class getClass()	Obtains the class of an object at runtime.
int hashCode()	Returns the hashcode associated with the invoking object.
void notify()	Resumes execution of a thread waiting on invoking object.
void notifyAll()	Resumes execution of all threads waiting on invoking thread.

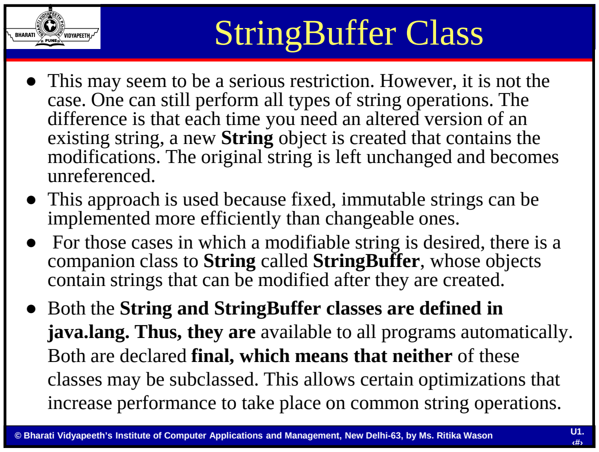
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



String Class

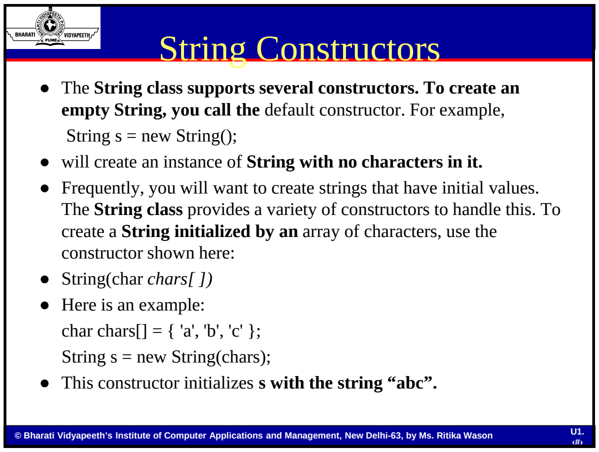
- Like many other programming languages in Java a String is a sequence of characters. But, unlike many other languages that implement strings as character arrays, Java implements strings as objects of type **String**.
- Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient.
- For example, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string.
- Also, **String** objects can be constructed a number of ways, making it easy to obtain a string when needed.
- Creating a **String** object, one actually creates a string that is immutable. That is, once a **String** object has been created, you cannot change the characters that comprise that string.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



StringBuffer Class

- This may seem to be a serious restriction. However, it is not the case. One can still perform all types of string operations. The difference is that each time you need an altered version of an existing string, a new **String** object is created that contains the modifications. The original string is left unchanged and becomes unreferenced.
- This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones.
- For those cases in which a modifiable string is desired, there is a companion class to **String** called **StringBuffer**, whose objects contain strings that can be modified after they are created.
- Both the **String and StringBuffer classes are defined in java.lang. Thus, they are** available to all programs automatically. Both are declared **final, which means that neither** of these classes may be subclassed. This allows certain optimizations that increase performance to take place on common string operations.



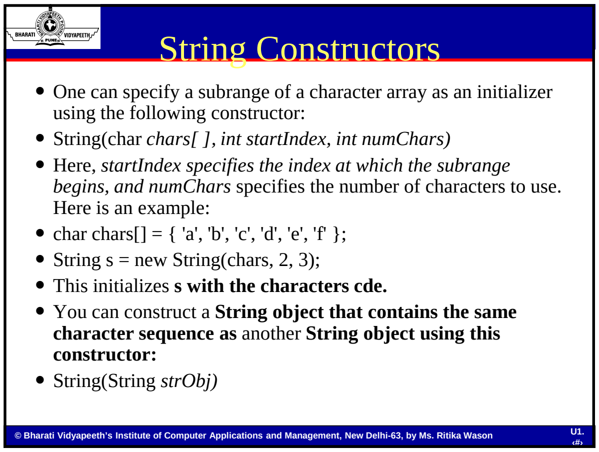
String Constructors

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

  String s = new String();
- will create an instance of **String with no characters in it.**
- Frequently, you will want to create strings that have initial values. The **String class** provides a variety of constructors to handle this. To create a **String initialized by an** array of characters, use the constructor shown here:
- String(char *chars[ ])*
- Here is an example:

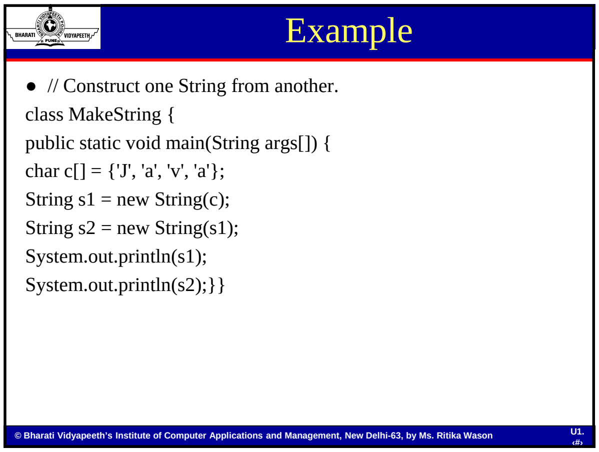
  char chars[] = { 'a', 'b', 'c' };

  String s = new String(chars);
- This constructor initializes **s with the string "abc".**



String Constructors

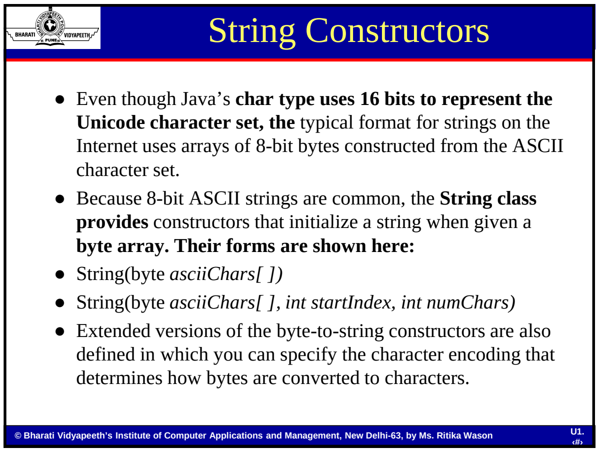
- One can specify a subrange of a character array as an initializer using the following constructor:
- String(char *chars[ ], int startIndex, int numChars)*
- Here, *startIndex specifies the index at which the subrange begins, and numChars* specifies the number of characters to use. Here is an example:
- char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
- String s = new String(chars, 2, 3);
- This initializes **s with the characters cde.**
- You can construct a **String object that contains the same character sequence as** another **String object using this constructor:**
- String(String *strObj)*



Example

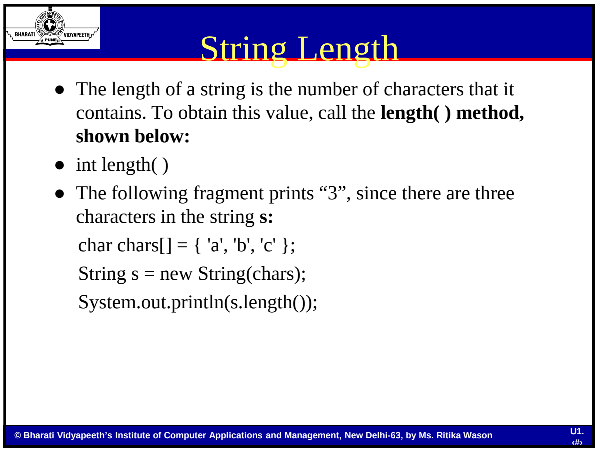
- // Construct one String from another.

```
class MakeString {
public static void main(String args[]) {
char c[] = {'J', 'a', 'v', 'a'};
String s1 = new String(c);
String s2 = new String(s1);
System.out.println(s1);
System.out.println(s2);}}
```



String Constructors

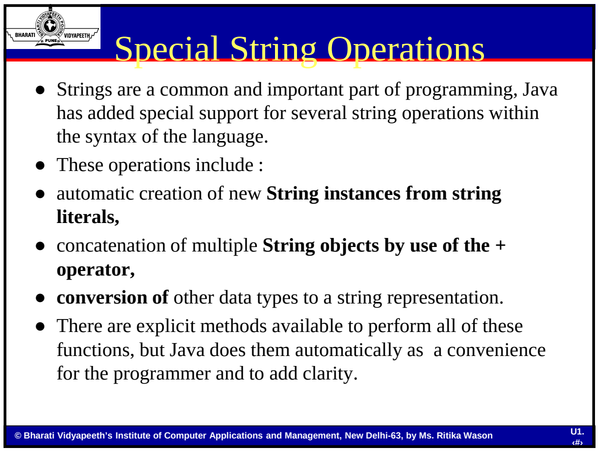
- Even though Java's **char type uses 16 bits to represent the Unicode character set, the** typical format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set.
- Because 8-bit ASCII strings are common, the **String class provides** constructors that initialize a string when given a **byte array. Their forms are shown here:**
- String(byte *asciiChars[ ])*
- String(byte *asciiChars[ ], int startIndex, int numChars)*
- Extended versions of the byte-to-string constructors are also defined in which you can specify the character encoding that determines how bytes are converted to characters.



String Length

- The length of a string is the number of characters that it contains. To obtain this value, call the **length( ) method, shown below:**
- int length( )
- The following fragment prints "3", since there are three characters in the string **s:**

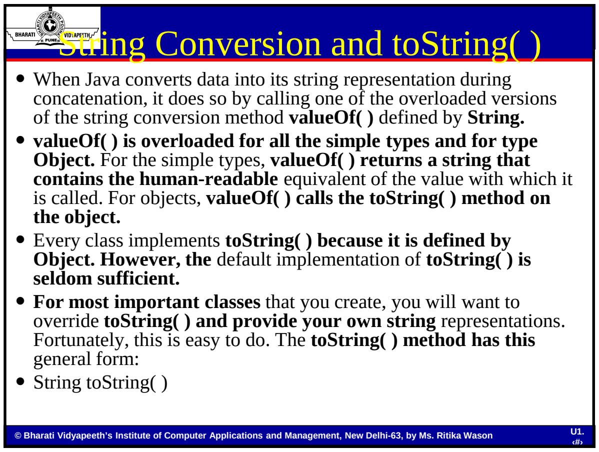
```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```



Special String Operations

- Strings are a common and important part of programming, Java has added special support for several string operations within the syntax of the language.
- These operations include :
- automatic creation of new **String instances from string literals,**
- concatenation of multiple **String objects by use of the + operator,**
- **conversion of** other data types to a string representation.
- There are explicit methods available to perform all of these functions, but Java does them automatically as a convenience for the programmer and to add clarity.

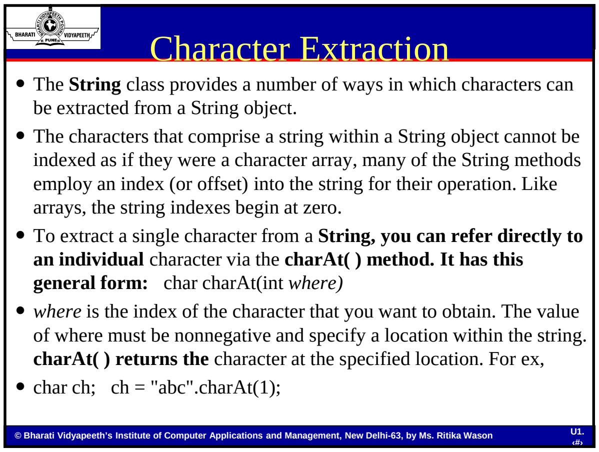
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



String Conversion and toString( )

- When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method **valueOf( )** defined by **String.**
- **valueOf( ) is overloaded for all the simple types and for type Object.** For the simple types, **valueOf( ) returns a string that contains the human-readable** equivalent of the value with which it is called. For objects, **valueOf( ) calls the toString( ) method on the object.**
- Every class implements **toString( ) because it is defined by Object. However, the** default implementation of **toString( ) is seldom sufficient.**
- **For most important classes** that you create, you will want to override **toString( ) and provide your own string** representations. Fortunately, this is easy to do. The **toString( ) method has this** general form:
- String toString( )

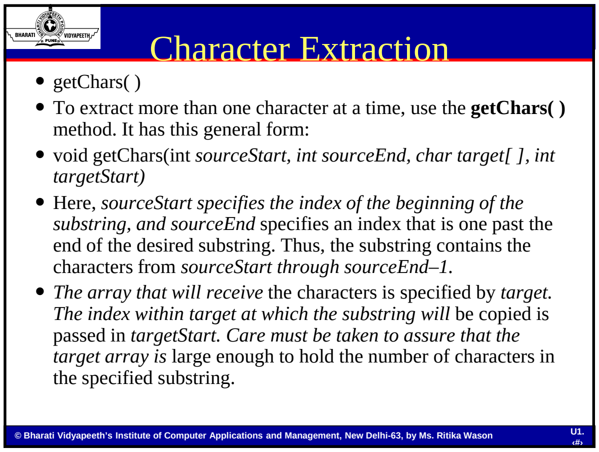
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Character Extraction

- The **String** class provides a number of ways in which characters can be extracted from a String object.
- The characters that comprise a string within a String object cannot be indexed as if they were a character array, many of the String methods employ an index (or offset) into the string for their operation. Like arrays, the string indexes begin at zero.
- To extract a single character from a **String, you can refer directly to an individual** character via the **charAt( ) method. It has this general form:**   char charAt(int *where)*
- *where* is the index of the character that you want to obtain. The value of where must be nonnegative and specify a location within the string. **charAt( ) returns the** character at the specified location. For ex,
- char ch;   ch = "abc".charAt(1);

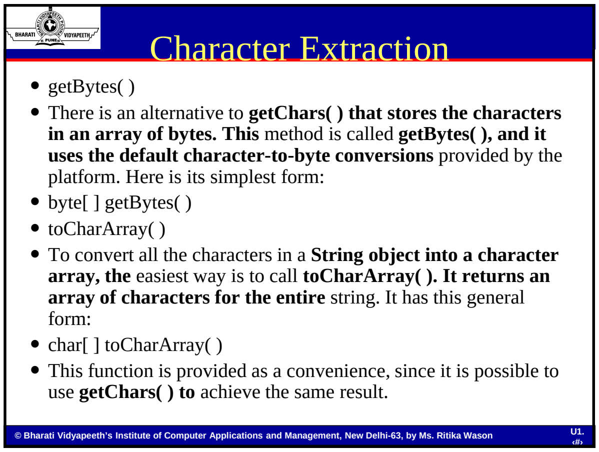
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Character Extraction

- getChars( )
- To extract more than one character at a time, use the **getChars( )** method. It has this general form:
- void getChars(int *sourceStart, int sourceEnd, char target[ ], int targetStart)*
- Here, *sourceStart specifies the index of the beginning of the substring, and sourceEnd* specifies an index that is one past the end of the desired substring. Thus, the substring contains the characters from *sourceStart through sourceEnd–1.*
- *The array that will receive* the characters is specified by *target. The index within target at which the substring will* be copied is passed in *targetStart. Care must be taken to assure that the target array is* large enough to hold the number of characters in the specified substring.

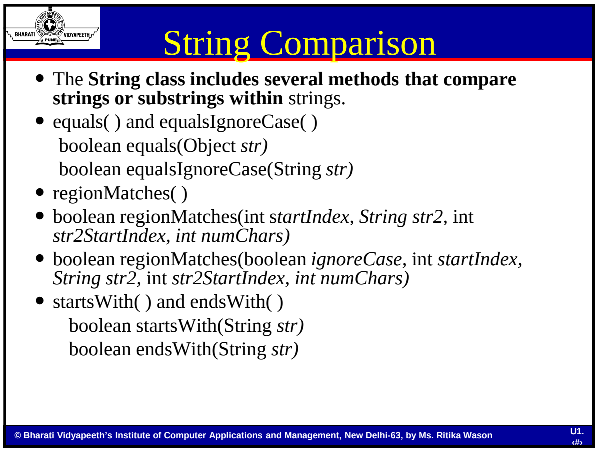
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Character Extraction

- getBytes( )
- There is an alternative to **getChars( ) that stores the characters in an array of bytes. This** method is called **getBytes( ), and it uses the default character-to-byte conversions** provided by the platform. Here is its simplest form:
- byte[ ] getBytes( )
- toCharArray( )
- To convert all the characters in a **String object into a character array, the** easiest way is to call **toCharArray( ). It returns an array of characters for the entire** string. It has this general form:
- char[ ] toCharArray( )
- This function is provided as a convenience, since it is possible to use **getChars( ) to** achieve the same result.

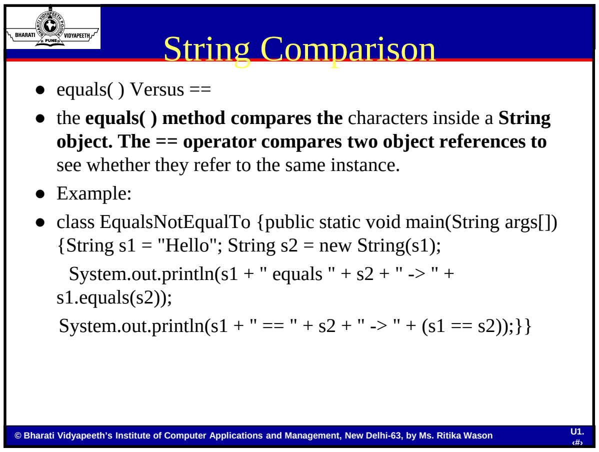
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



String Comparison

- The **String class includes several methods that compare strings or substrings within** strings.
- equals( ) and equalsIgnoreCase( )
  boolean equals(Object *str)*
  boolean equalsIgnoreCase(String *str)*
- regionMatches( )
- boolean regionMatches(int s*tartIndex, String str2*, int *str2StartIndex, int numChars)*
- boolean regionMatches(boolean *ignoreCase*, int *startIndex, String str2*, int *str2StartIndex, int numChars)*
- startsWith( ) and endsWith( )
  boolean startsWith(String *str)*
  boolean endsWith(String *str)*

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



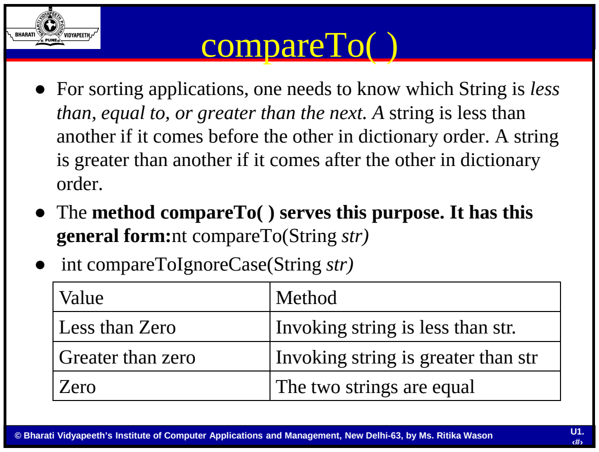
String Comparison

- equals( ) Versus ==
- the **equals( ) method compares the** characters inside a **String object. The == operator compares two object references to** see whether they refer to the same instance.
- Example:
- class EqualsNotEqualTo {public static void main(String args[]) {String s1 = "Hello"; String s2 = new String(s1);

  System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));

  System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));}}

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›

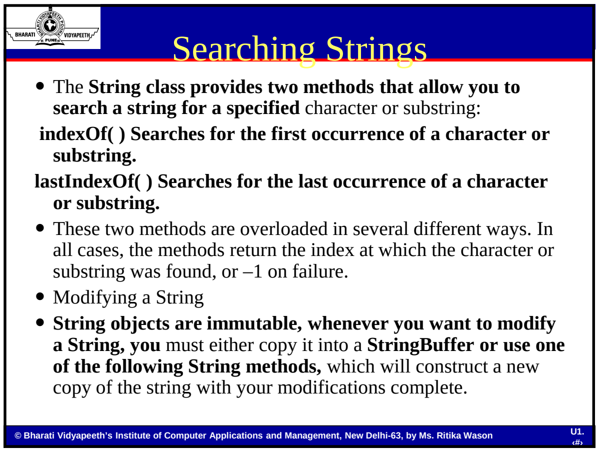


compareTo( )

- For sorting applications, one needs to know which String is *less than, equal to, or greater than the next. A* string is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order.
- The **method compareTo( ) serves this purpose. It has this general form:**nt compareTo(String *str)*
- int compareToIgnoreCase(String *str)*

Value	Method
Less than Zero	Invoking string is less than str.
Greater than zero	Invoking string is greater than str
Zero	The two strings are equal

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Searching Strings

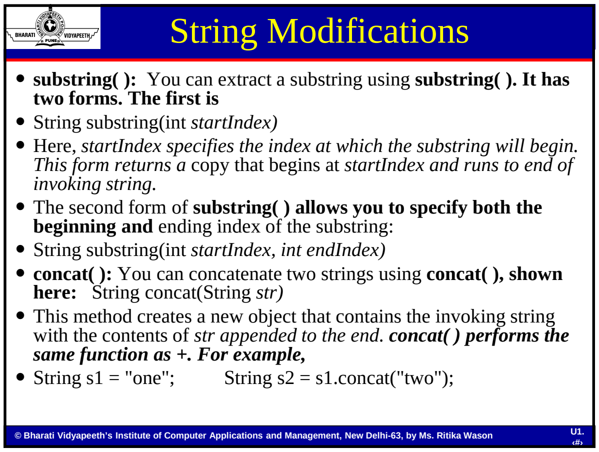
- The **String class provides two methods that allow you to search a string for a specified** character or substring:

indexOf( ) Searches for the first occurrence of a character or substring.

lastIndexOf( ) Searches for the last occurrence of a character or substring.

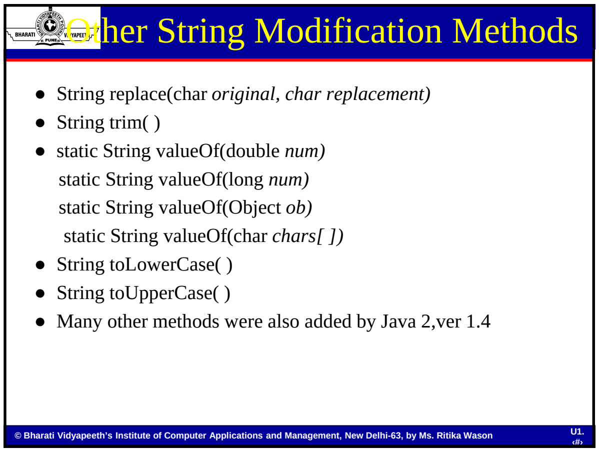
- These two methods are overloaded in several different ways. In all cases, the methods return the index at which the character or substring was found, or –1 on failure.
- Modifying a String
- **String objects are immutable, whenever you want to modify a String, you** must either copy it into a **StringBuffer or use one of the following String methods,** which will construct a new copy of the string with your modifications complete.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



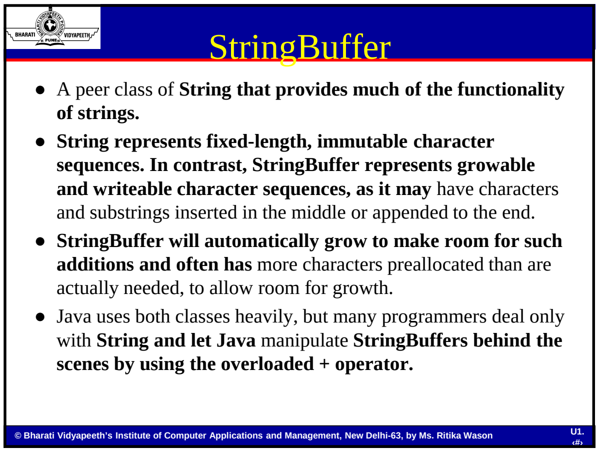
String Modifications

- **substring( ):** You can extract a substring using **substring( ). It has two forms. The first is**
- String substring(int *startIndex)*
- Here, *startIndex specifies the index at which the substring will begin. This form returns a* copy that begins at *startIndex and runs to end of invoking string.*
- The second form of **substring( ) allows you to specify both the beginning and** ending index of the substring:
- String substring(int *startIndex, int endIndex)*
- **concat( ):** You can concatenate two strings using **concat( ), shown here:** String concat(String *str)*
- This method creates a new object that contains the invoking string with the contents of *str appended to the end.* ***concat( ) performs the same function as +. For example,***
- String s1 = "one"; String s2 = s1.concat("two");



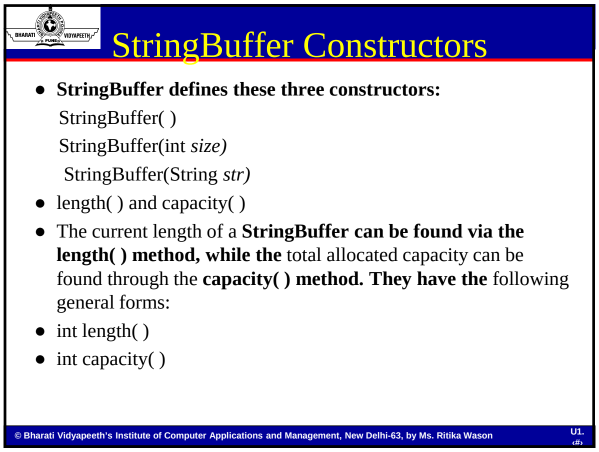
Other String Modification Methods

- String replace(char *original, char replacement)*
- String trim( )
- static String valueOf(double *num)*
  static String valueOf(long *num)*
  static String valueOf(Object *ob)*
  static String valueOf(char *chars[ ])*
- String toLowerCase( )
- String toUpperCase( )
- Many other methods were also added by Java 2,ver 1.4



StringBuffer

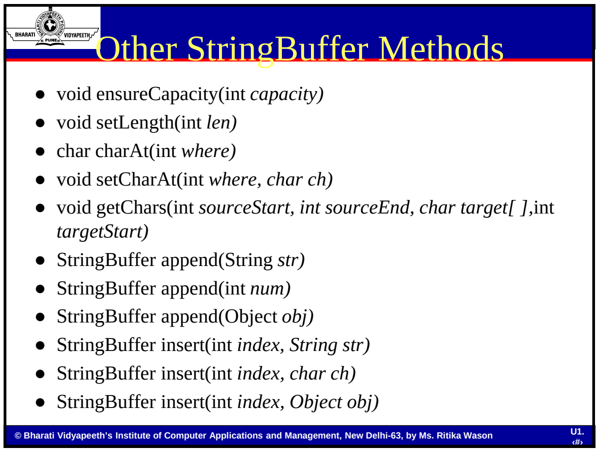
- A peer class of **String that provides much of the functionality of strings.**
- **String represents fixed-length, immutable character sequences. In contrast, StringBuffer represents growable and writeable character sequences, as it may** have characters and substrings inserted in the middle or appended to the end.
- **StringBuffer will automatically grow to make room for such additions and often has** more characters preallocated than are actually needed, to allow room for growth.
- Java uses both classes heavily, but many programmers deal only with **String and let Java** manipulate **StringBuffers behind the scenes by using the overloaded + operator.**



StringBuffer Constructors

- **StringBuffer defines these three constructors:**
  StringBuffer( )
  StringBuffer(int *size)*
  StringBuffer(String *str)*
- length( ) and capacity( )
- The current length of a **StringBuffer can be found via the length( ) method, while the** total allocated capacity can be found through the **capacity( ) method. They have the** following general forms:
- int length( )
- int capacity( )

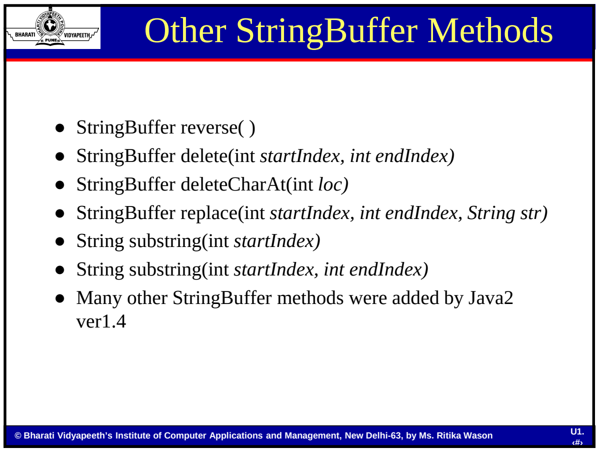
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Other StringBuffer Methods

- void ensureCapacity(int *capacity)*
- void setLength(int *len)*
- char charAt(int *where)*
- void setCharAt(int *where, char ch)*
- void getChars(int *sourceStart, int sourceEnd, char target[ ],*int *targetStart)*
- StringBuffer append(String *str)*
- StringBuffer append(int *num)*
- StringBuffer append(Object *obj)*
- StringBuffer insert(int *index, String str)*
- StringBuffer insert(int *index, char ch)*
- StringBuffer insert(int *index, Object obj)*

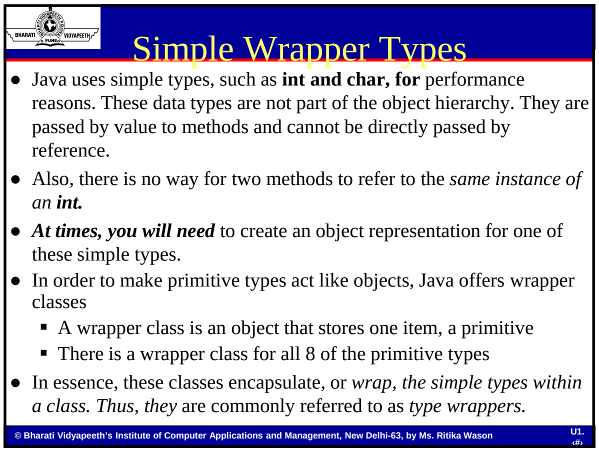
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Other StringBuffer Methods

- StringBuffer reverse( )
- StringBuffer delete(int *startIndex, int endIndex)*
- StringBuffer deleteCharAt(int *loc)*
- StringBuffer replace(int *startIndex, int endIndex, String str)*
- String substring(int *startIndex)*
- String substring(int *startIndex, int endIndex)*
- Many other StringBuffer methods were added by Java2 ver1.4

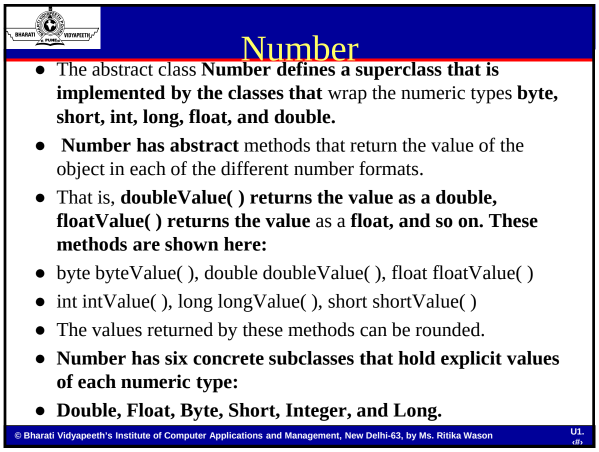
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Simple Wrapper Types

- Java uses simple types, such as **int and char, for** performance reasons. These data types are not part of the object hierarchy. They are passed by value to methods and cannot be directly passed by reference.
- Also, there is no way for two methods to refer to the *same instance of an* ***int.***
- ***At times, you will need*** to create an object representation for one of these simple types.
- In order to make primitive types act like objects, Java offers wrapper classes
  - A wrapper class is an object that stores one item, a primitive
  - There is a wrapper class for all 8 of the primitive types
- In essence, these classes encapsulate, or *wrap, the simple types within a class. Thus, they* are commonly referred to as *type wrappers.*

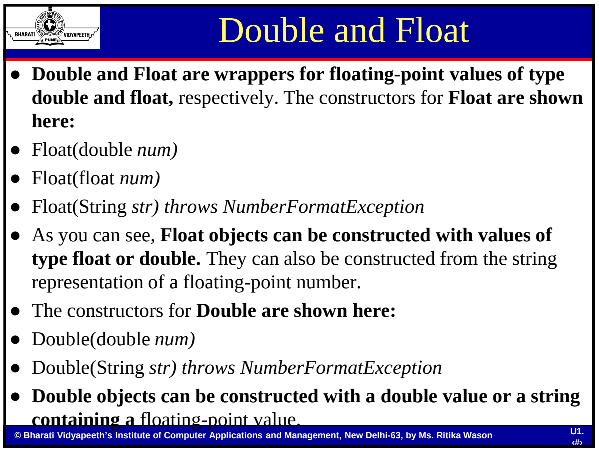
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Number

- The abstract class **Number defines a superclass that is implemented by the classes that** wrap the numeric types **byte, short, int, long, float, and double.**
- **Number has abstract** methods that return the value of the object in each of the different number formats.
- That is, **doubleValue( ) returns the value as a double, floatValue( ) returns the value** as a **float, and so on. These methods are shown here:**
- byte byteValue( ), double doubleValue( ), float floatValue( )
- int intValue( ), long longValue( ), short shortValue( )
- The values returned by these methods can be rounded.
- **Number has six concrete subclasses that hold explicit values of each numeric type:**
- **Double, Float, Byte, Short, Integer, and Long.**

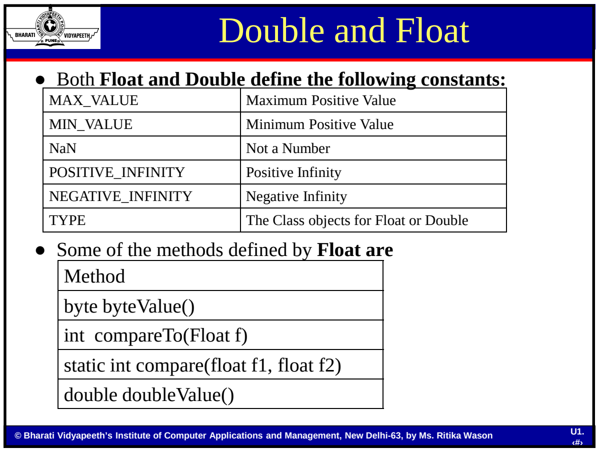
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Double and Float

- **Double and Float are wrappers for floating-point values of type double and float,** respectively. The constructors for **Float are shown here:**
- Float(double *num)*
- Float(float *num)*
- Float(String *str) throws NumberFormatException*
- As you can see, **Float objects can be constructed with values of type float or double.** They can also be constructed from the string representation of a floating-point number.
- The constructors for **Double are shown here:**
- Double(double *num)*
- Double(String *str) throws NumberFormatException*
- **Double objects can be constructed with a double value or a string containing a** floating-point value.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



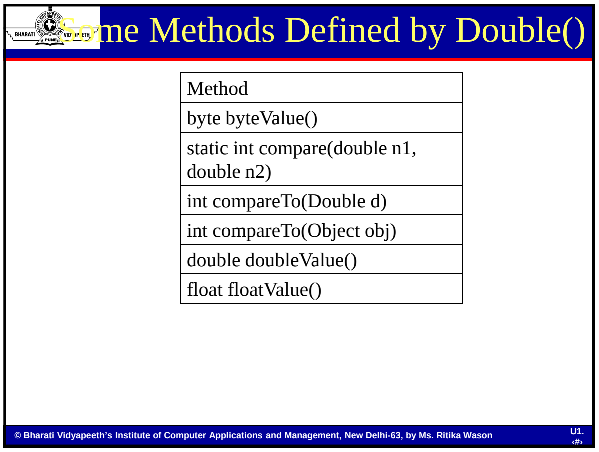
Double and Float

- Both **Float and Double define the following constants:**

MAX_VALUE	Maximum Positive Value
MIN_VALUE	Minimum Positive Value
NaN	Not a Number
POSITIVE_INFINITY	Positive Infinity
NEGATIVE_INFINITY	Negative Infinity
TYPE	The Class objects for Float or Double

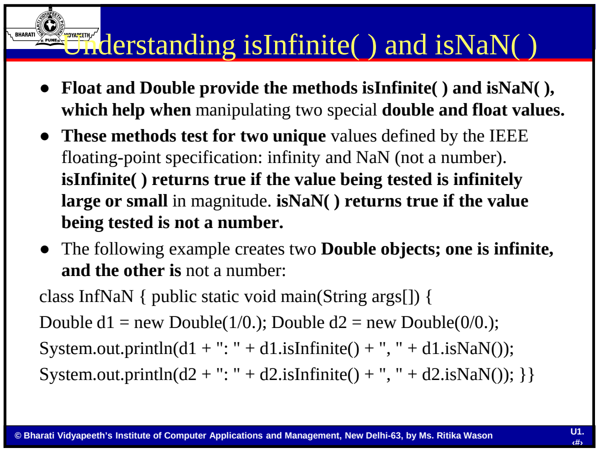
- Some of the methods defined by **Float are**

Method
byte byteValue()
int compareTo(Float f)
static int compare(float f1, float f2)
double doubleValue()



Some Methods Defined by Double()

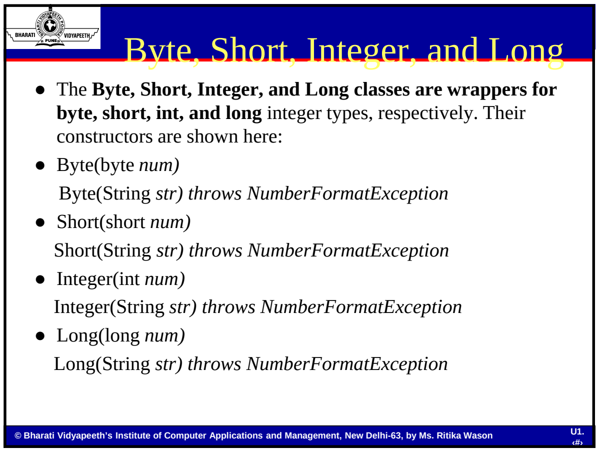
Method
byte byteValue()
static int compare(double n1, double n2)
int compareTo(Double d)
int compareTo(Object obj)
double doubleValue()
float floatValue()



Understanding isInfinite( ) and isNaN( )

- **Float and Double provide the methods isInfinite( ) and isNaN( ), which help when** manipulating two special **double and float values.**
- **These methods test for two unique** values defined by the IEEE floating-point specification: infinity and NaN (not a number). **isInfinite( ) returns true if the value being tested is infinitely large or small** in magnitude. **isNaN( ) returns true if the value being tested is not a number.**
- The following example creates two **Double objects; one is infinite, and the other is** not a number:

```
class InfNaN { public static void main(String args[]) {
Double d1 = new Double(1/0.); Double d2 = new Double(0/0.);
System.out.println(d1 + ": " + d1.isInfinite() + ", " + d1.isNaN());
System.out.println(d2 + ": " + d2.isInfinite() + ", " + d2.isNaN()); }}
```



Byte, Short, Integer, and Long

- The **Byte, Short, Integer, and Long classes are wrappers for byte, short, int, and long** integer types, respectively. Their constructors are shown here:
- Byte(byte *num)*

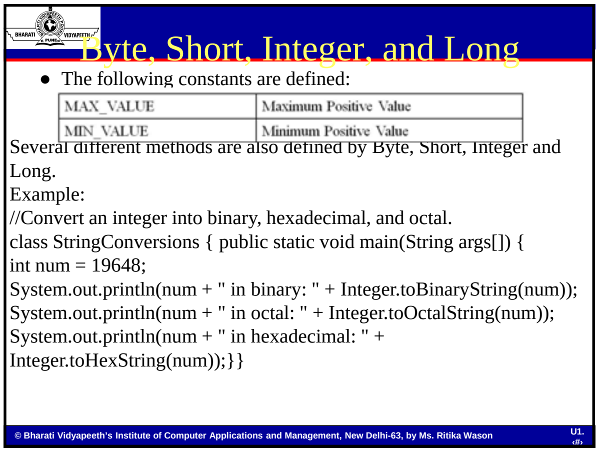
  Byte(String *str) throws NumberFormatException*
- Short(short *num)*

  Short(String *str) throws NumberFormatException*
- Integer(int *num)*

  Integer(String *str) throws NumberFormatException*
- Long(long *num)*

  Long(String *str) throws NumberFormatException*

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Byte, Short, Integer, and Long

- The following constants are defined:

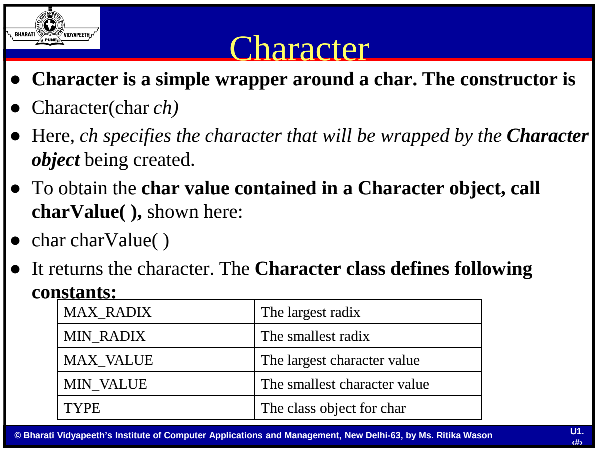
MAX_VALUE	Maximum Positive Value
MIN_VALUE	Minimum Positive Value

Several different methods are also defined by Byte, Short, Integer and Long.

Example:

```
//Convert an integer into binary, hexadecimal, and octal.
class StringConversions { public static void main(String args[]) {
int num = 19648;
System.out.println(num + " in binary: " + Integer.toBinaryString(num));
System.out.println(num + " in octal: " + Integer.toOctalString(num));
System.out.println(num + " in hexadecimal: " +
Integer.toHexString(num));}}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›

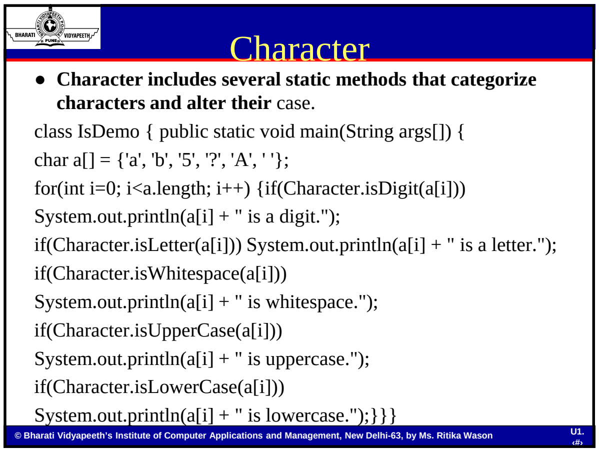


Character

- **Character is a simple wrapper around a char. The constructor is**
- Character(char *ch)*
- Here, *ch specifies the character that will be wrapped by the* ***Character object*** being created.
- To obtain the **char value contained in a Character object, call charValue( ),** shown here:
- char charValue( )
- It returns the character. The **Character class defines following constants:**

MAX_RADIX	The largest radix
MIN_RADIX	The smallest radix
MAX_VALUE	The largest character value
MIN_VALUE	The smallest character value
TYPE	The class object for char

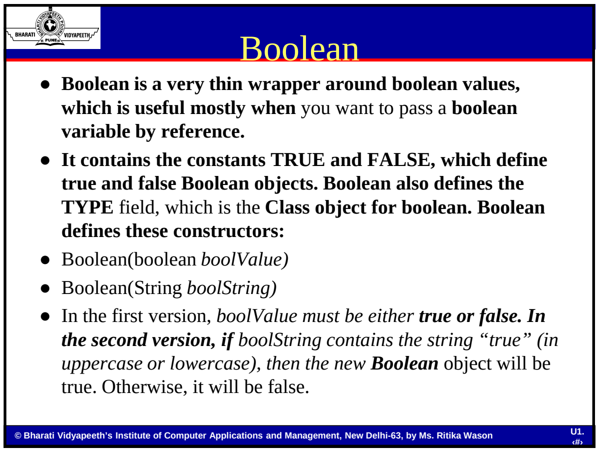
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Character

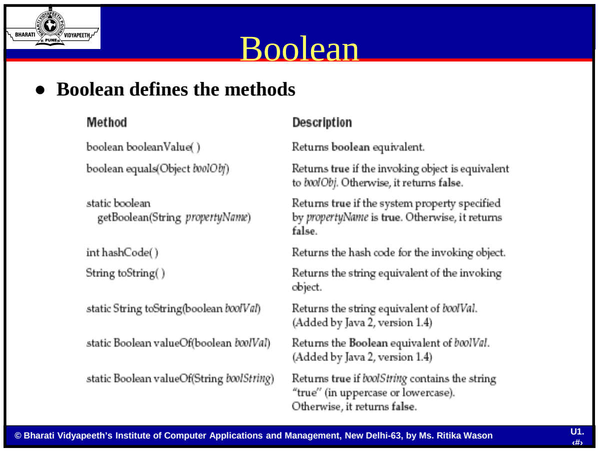
- **Character includes several static methods that categorize characters and alter their** case.

```
class IsDemo { public static void main(String args[]) {
char a[] = {'a', 'b', '5', '?', 'A', ' '};
for(int i=0; i<a.length; i++) {if(Character.isDigit(a[i]))
System.out.println(a[i] + " is a digit.");
if(Character.isLetter(a[i])) System.out.println(a[i] + " is a letter.");
if(Character.isWhitespace(a[i]))
System.out.println(a[i] + " is whitespace.");
if(Character.isUpperCase(a[i]))
System.out.println(a[i] + " is uppercase.");
if(Character.isLowerCase(a[i]))
System.out.println(a[i] + " is lowercase.");}}}
```



Boolean

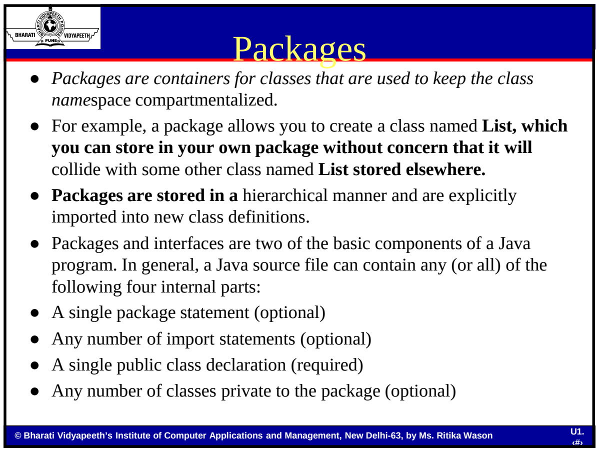
- **Boolean is a very thin wrapper around boolean values, which is useful mostly when** you want to pass a **boolean variable by reference.**
- **It contains the constants TRUE and FALSE, which define true and false Boolean objects. Boolean also defines the TYPE** field, which is the **Class object for boolean. Boolean defines these constructors:**
- Boolean(boolean *boolValue)*
- Boolean(String *boolString)*
- In the first version, *boolValue must be either* ***true or false. In the second version, if*** *boolString contains the string "true" (in uppercase or lowercase), then the new* ***Boolean*** object will be true. Otherwise, it will be false.



Boolean

- **Boolean defines the methods**

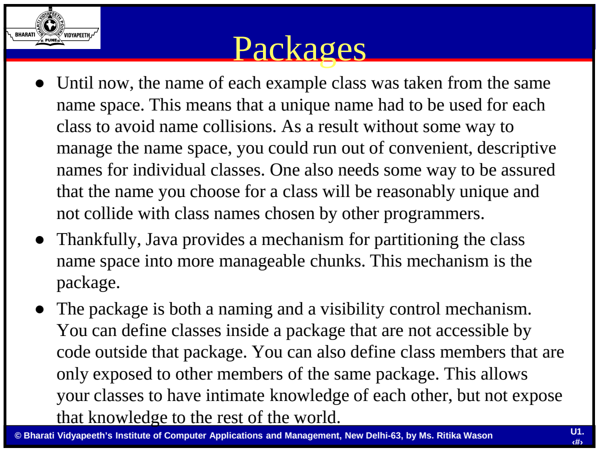
Method	Description
boolean booleanValue( )	Returns **boolean** equivalent.
boolean equals(Object *boolObj*)	Returns **true** if the invoking object is equivalent to *boolObj*. Otherwise, it returns **false**.
static boolean getBoolean(String *propertyName*)	Returns **true** if the system property specified by *propertyName* is **true**. Otherwise, it returns **false**.
int hashCode( )	Returns the hash code for the invoking object.
String toString( )	Returns the string equivalent of the invoking object.
static String toString(boolean *boolVal*)	Returns the string equivalent of *boolVal*. (Added by Java 2, version 1.4)
static Boolean valueOf(boolean *boolVal*)	Returns the **Boolean** equivalent of *boolVal*. (Added by Java 2, version 1.4)
static Boolean valueOf(String *boolString*)	Returns **true** if *boolString* contains the string "true" (in uppercase or lowercase). Otherwise, it returns **false**.



Packages

- *Packages are containers for classes that are used to keep the class name*space compartmentalized.
- For example, a package allows you to create a class named **List, which you can store in your own package without concern that it will** collide with some other class named **List stored elsewhere.**
- **Packages are stored in a** hierarchical manner and are explicitly imported into new class definitions.
- Packages and interfaces are two of the basic components of a Java program. In general, a Java source file can contain any (or all) of the following four internal parts:
- A single package statement (optional)
- Any number of import statements (optional)
- A single public class declaration (required)
- Any number of classes private to the package (optional)

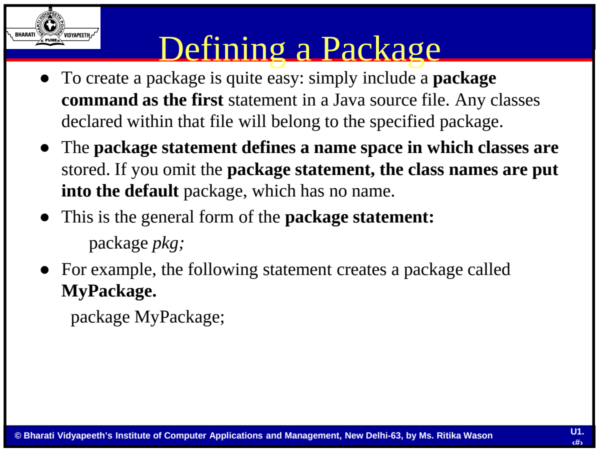
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Packages

- Until now, the name of each example class was taken from the same name space. This means that a unique name had to be used for each class to avoid name collisions. As a result without some way to manage the name space, you could run out of convenient, descriptive names for individual classes. One also needs some way to be assured that the name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers.
- Thankfully, Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package.
- The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are only exposed to other members of the same package. This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



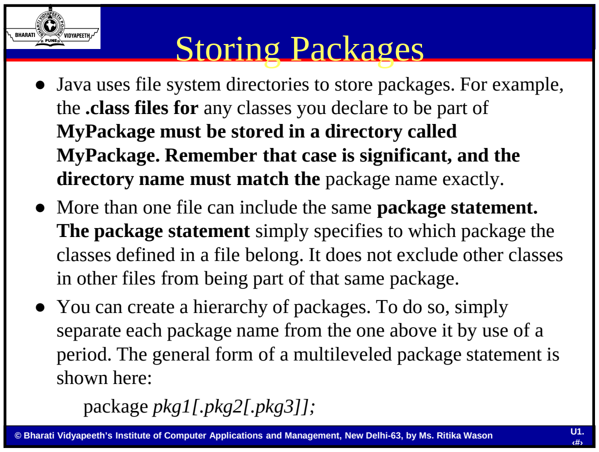
Defining a Package

- To create a package is quite easy: simply include a **package command as the first** statement in a Java source file. Any classes declared within that file will belong to the specified package.
- The **package statement defines a name space in which classes are** stored. If you omit the **package statement, the class names are put into the default** package, which has no name.
- This is the general form of the **package statement:**

  package *pkg;*
- For example, the following statement creates a package called **MyPackage.**

  package MyPackage;

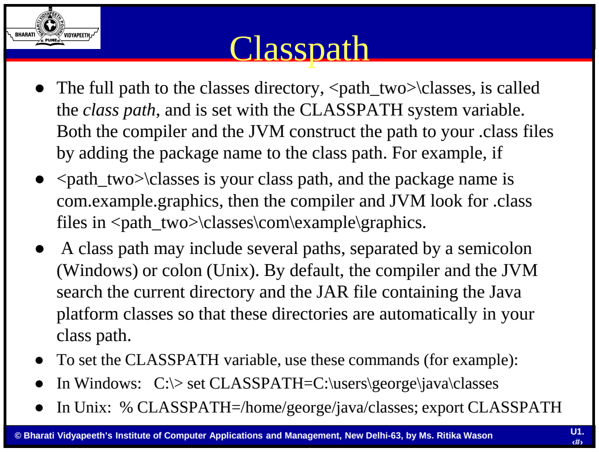
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason U1.‹#›



Storing Packages

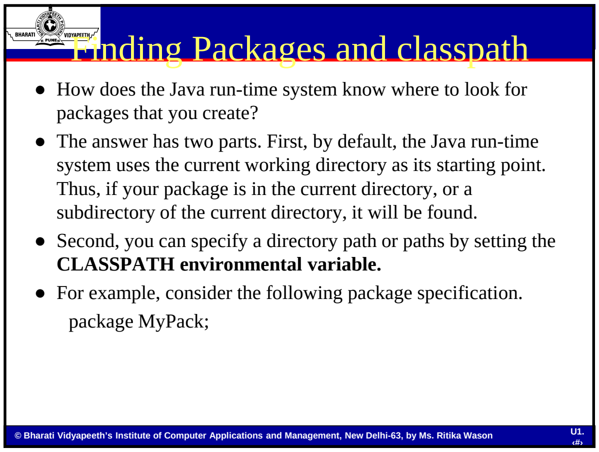
- Java uses file system directories to store packages. For example, the **.class files for** any classes you declare to be part of **MyPackage must be stored in a directory called MyPackage. Remember that case is significant, and the directory name must match the** package name exactly.
- More than one file can include the same **package statement. The package statement** simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package.
- You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

  package *pkg1[.pkg2[.pkg3]];*



Classpath

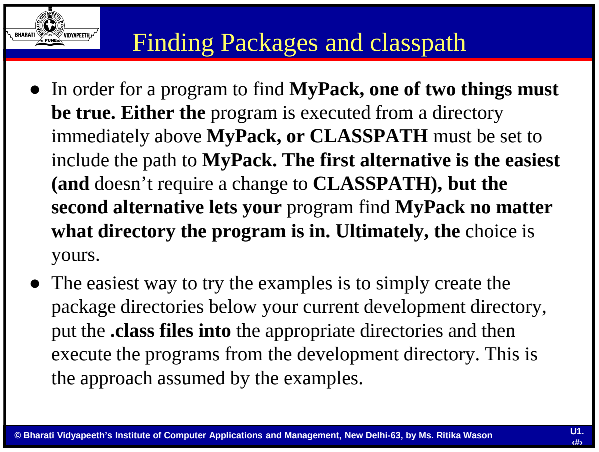
- The full path to the classes directory, <path_two>\classes, is called the *class path*, and is set with the CLASSPATH system variable. Both the compiler and the JVM construct the path to your .class files by adding the package name to the class path. For example, if
- <path_two>\classes is your class path, and the package name is com.example.graphics, then the compiler and JVM look for .class files in <path_two>\classes\com\example\graphics.
- A class path may include several paths, separated by a semicolon (Windows) or colon (Unix). By default, the compiler and the JVM search the current directory and the JAR file containing the Java platform classes so that these directories are automatically in your class path.
- To set the CLASSPATH variable, use these commands (for example):
- In Windows: C:\> set CLASSPATH=C:\users\george\java\classes
- In Unix: % CLASSPATH=/home/george/java/classes; export CLASSPATH



Finding Packages and classpath

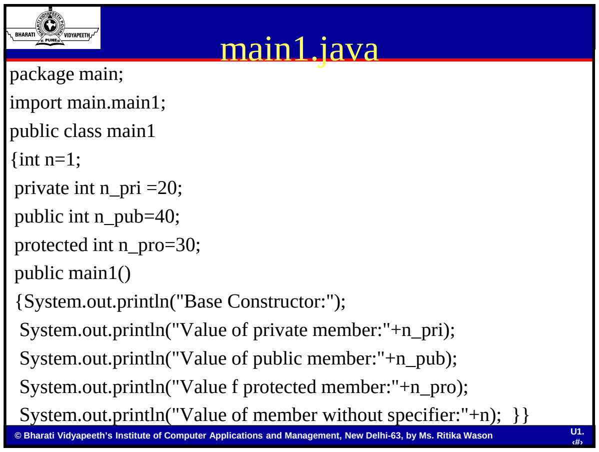
- How does the Java run-time system know where to look for packages that you create?
- The answer has two parts. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in the current directory, or a subdirectory of the current directory, it will be found.
- Second, you can specify a directory path or paths by setting the **CLASSPATH environmental variable.**
- For example, consider the following package specification.

  package MyPack;



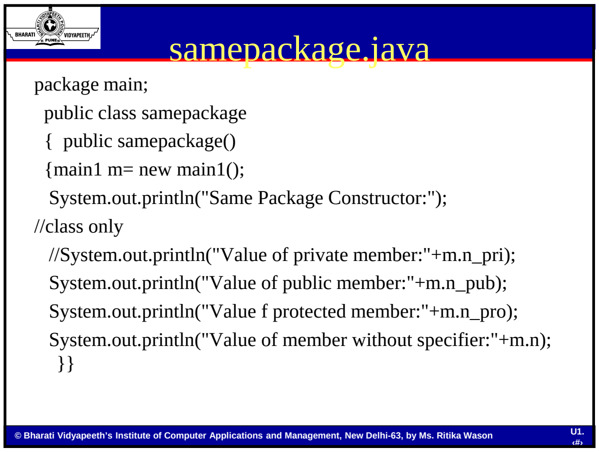
Finding Packages and classpath

- In order for a program to find **MyPack, one of two things must be true. Either the** program is executed from a directory immediately above **MyPack, or CLASSPATH** must be set to include the path to **MyPack. The first alternative is the easiest (and** doesn’t require a change to **CLASSPATH), but the second alternative lets your** program find **MyPack no matter what directory the program is in. Ultimately, the** choice is yours.
- The easiest way to try the examples is to simply create the package directories below your current development directory, put the **.class files into** the appropriate directories and then execute the programs from the development directory. This is the approach assumed by the examples.



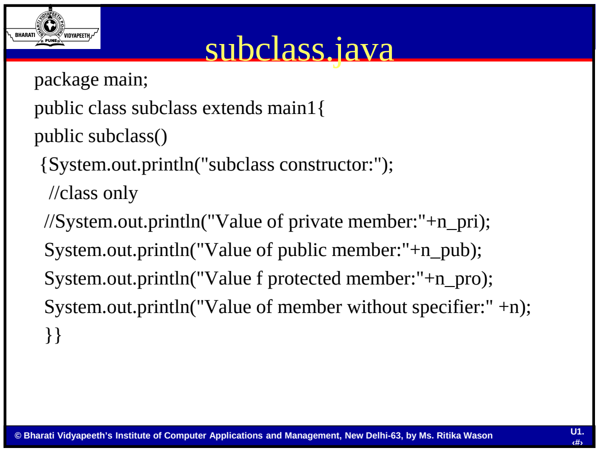
main1.java

```
package main;
import main.main1;
public class main1
{int n=1;
 private int n_pri =20;
 public int n_pub=40;
 protected int n_pro=30;
 public main1()
 {System.out.println("Base Constructor:");
  System.out.println("Value of private member:"+n_pri);
  System.out.println("Value of public member:"+n_pub);
  System.out.println("Value f protected member:"+n_pro);
  System.out.println("Value of member without specifier:"+n);  }}
```



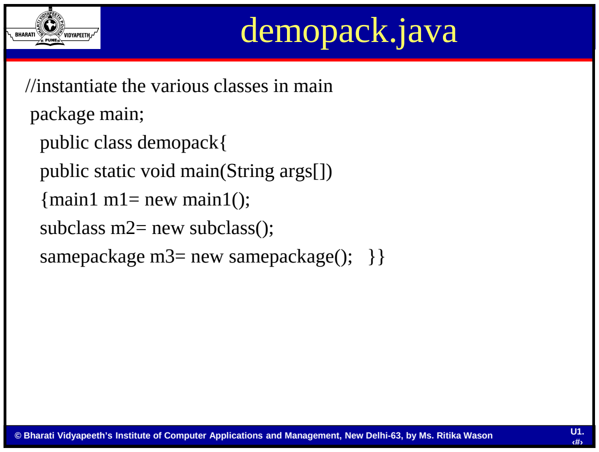
samepackage.java

```
package main;
  public class samepackage
  {  public samepackage()
  {main1 m= new main1();
   System.out.println("Same Package Constructor:");
//class only
   //System.out.println("Value of private member:"+m.n_pri);
   System.out.println("Value of public member:"+m.n_pub);
   System.out.println("Value f protected member:"+m.n_pro);
   System.out.println("Value of member without specifier:"+m.n);
    }}
```



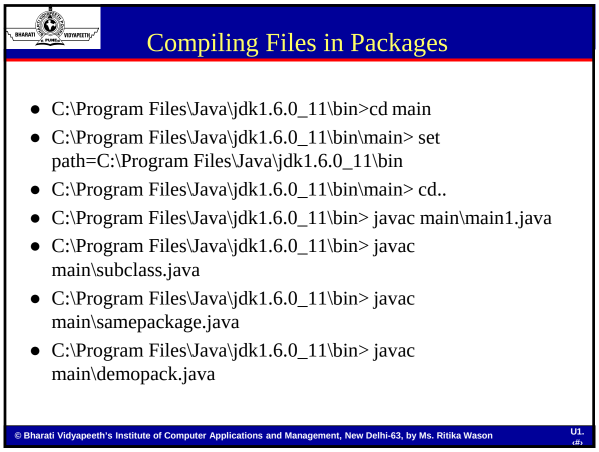
subclass.java

```
package main;
public class subclass extends main1{
public subclass()
 {System.out.println("subclass constructor:");
   //class only
  //System.out.println("Value of private member:"+n_pri);
  System.out.println("Value of public member:"+n_pub);
  System.out.println("Value f protected member:"+n_pro);
  System.out.println("Value of member without specifier:" +n);
  }}
```



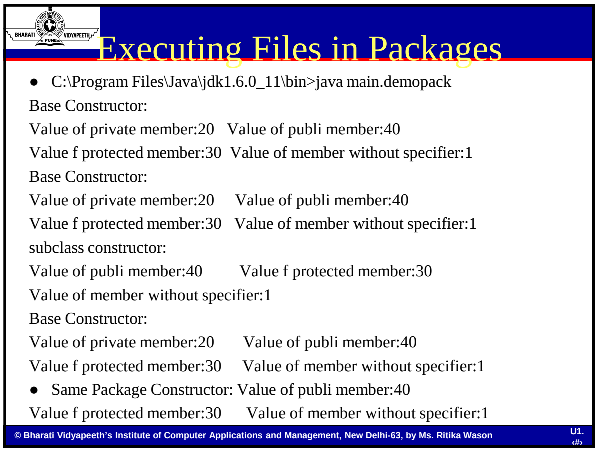
demopack.java

```
//instantiate the various classes in main
 package main;
   public class demopack{
   public static void main(String args[])
   {main1 m1= new main1();
   subclass m2= new subclass();
   samepackage m3= new samepackage();   }}
```



Compiling Files in Packages

- C:\Program Files\Java\jdk1.6.0_11\bin>cd main
- C:\Program Files\Java\jdk1.6.0_11\bin\main> set path=C:\Program Files\Java\jdk1.6.0_11\bin
- C:\Program Files\Java\jdk1.6.0_11\bin\main> cd..
- C:\Program Files\Java\jdk1.6.0_11\bin> javac main\main1.java
- C:\Program Files\Java\jdk1.6.0_11\bin> javac main\subclass.java
- C:\Program Files\Java\jdk1.6.0_11\bin> javac main\samepackage.java
- C:\Program Files\Java\jdk1.6.0_11\bin> javac main\demopack.java



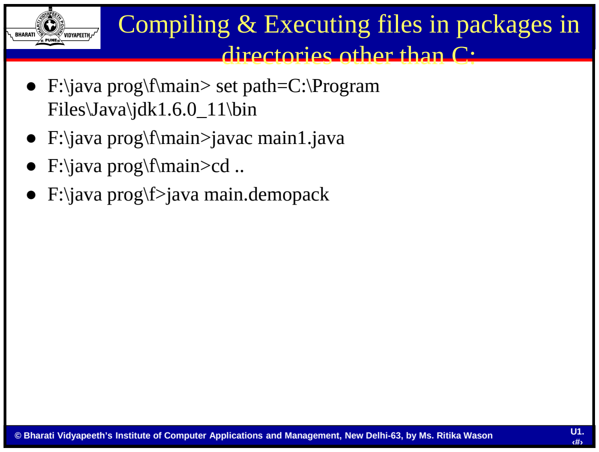
Executing Files in Packages

- C:\Program Files\Java\jdk1.6.0_11\bin>java main.demopack

```
Base Constructor:
Value of private member:20   Value of publi member:40
Value f protected member:30  Value of member without specifier:1
Base Constructor:
Value of private member:20     Value of publi member:40
Value f protected member:30   Value of member without specifier:1
subclass constructor:
Value of publi member:40        Value f protected member:30
Value of member without specifier:1
Base Constructor:
Value of private member:20      Value of publi member:40
Value f protected member:30     Value of member without specifier:1
```

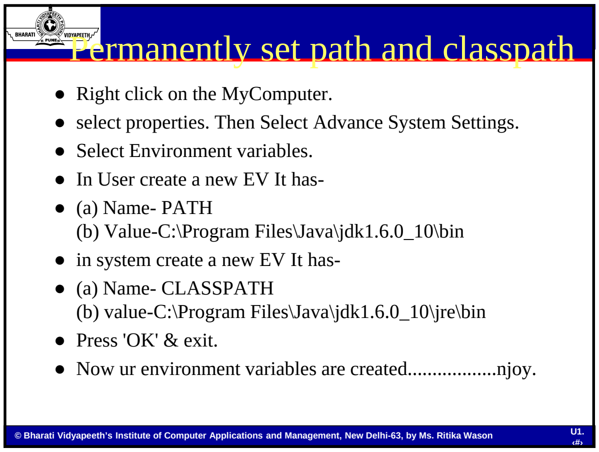
- Same Package Constructor: Value of publi member:40

```
Value f protected member:30     Value of member without specifier:1
```



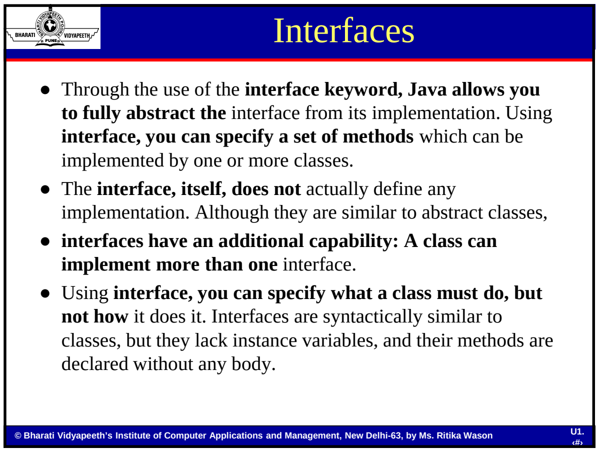
Compiling & Executing files in packages in directories other than C:

- F:\java prog\f\main> set path=C:\Program Files\Java\jdk1.6.0_11\bin
- F:\java prog\f\main>javac main1.java
- F:\java prog\f\main>cd ..
- F:\java prog\f>java main.demopack



Permanently set path and classpath

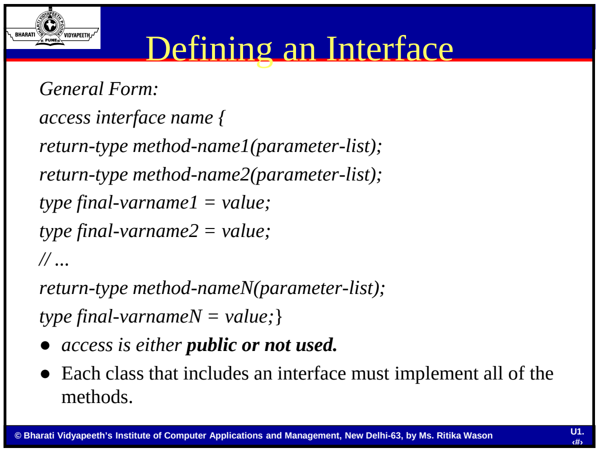
- Right click on the MyComputer.
- select properties. Then Select Advance System Settings.
- Select Environment variables.
- In User create a new EV It has-
- (a) Name- PATH
  (b) Value-C:\Program Files\Java\jdk1.6.0_10\bin
- in system create a new EV It has-
- (a) Name- CLASSPATH
  (b) value-C:\Program Files\Java\jdk1.6.0_10\jre\bin
- Press 'OK' & exit.
- Now ur environment variables are created..................njoy.



Interfaces

- Through the use of the **interface keyword, Java allows you to fully abstract the** interface from its implementation. Using **interface, you can specify a set of methods** which can be implemented by one or more classes.
- The **interface, itself, does not** actually define any implementation. Although they are similar to abstract classes,
- **interfaces have an additional capability: A class can implement more than one** interface.
- Using **interface, you can specify what a class must do, but not how** it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.

U1.‹#›

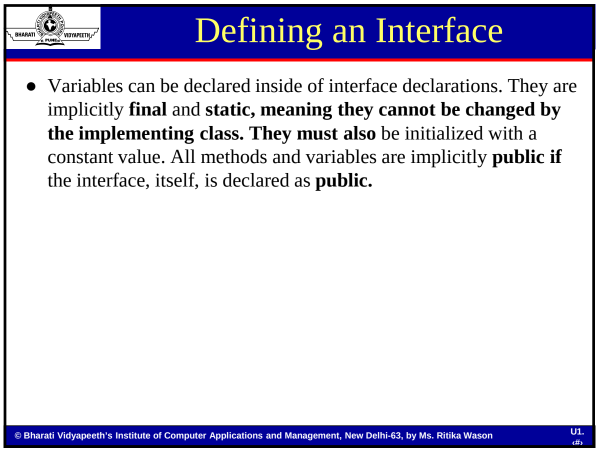


Defining an Interface

```
General Form:
access interface name {
return-type method-name1(parameter-list);
return-type method-name2(parameter-list);
type final-varname1 = value;
type final-varname2 = value;
// ...
return-type method-nameN(parameter-list);
type final-varnameN = value;}
```

- *access is either **public or not used.***
- Each class that includes an interface must implement all of the methods.

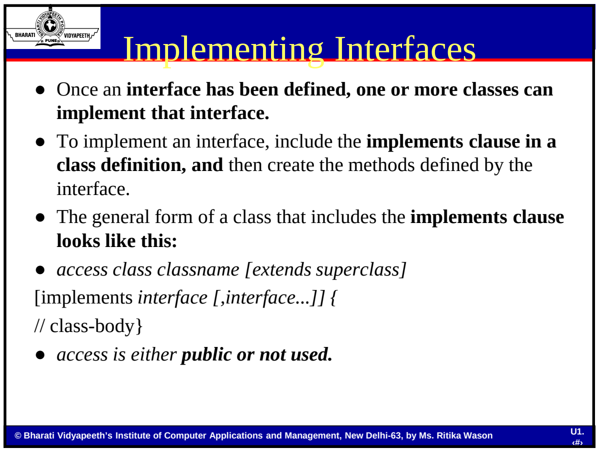
U1.‹#›



Defining an Interface

- Variables can be declared inside of interface declarations. They are implicitly **final** and **static, meaning they cannot be changed by the implementing class. They must also** be initialized with a constant value. All methods and variables are implicitly **public if** the interface, itself, is declared as **public.**

U1.‹#›



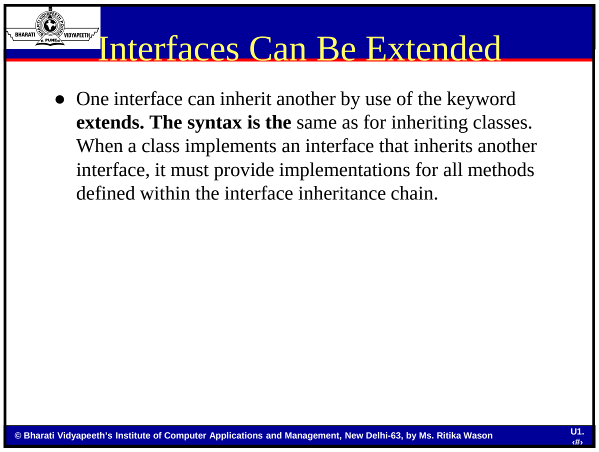
Implementing Interfaces

- Once an **interface has been defined, one or more classes can implement that interface.**
- To implement an interface, include the **implements clause in a class definition, and** then create the methods defined by the interface.
- The general form of a class that includes the **implements clause looks like this:**
- *access class classname [extends superclass]*

[implements *interface [,interface...]] {*

// class-body}

- *access is either* ***public or not used.***



Interfaces Can Be Extended

- One interface can inherit another by use of the keyword **extends. The syntax is the** same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.