Unit - I: Java Basics

History of Java

History of Java

- Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991.
- This language was initially called “Oak,” but was renamed “Java” in 1995.

JAVA

The primary motivation for Java was the need for a platform-independent (that is, architecture-neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls.

The Java Buzzwords

The Java Buzzwords

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

Robust

- The ability to create robust programs was given a high priority in the design of Java.
- Consider two of the main reasons for program failure: memory management mistakes and mishandled exceptional conditions.

Memory management

- It can be a difficult, tedious task in traditional programming environments. Java virtually eliminates these problems by managing memory allocation and de-allocation.
- De-allocation is completely automatic, because Java provides **garbage collection** for unused objects.
- **Exceptional conditions** - such as division by zero or “file not found,”. Java helps by providing object-oriented **exception handling**.

Multithreaded

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

Multithreaded

- The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems.

Architecture-Neutral

- A central issue for the Java designers was that of code longevity and portability.
- One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow—even on the same machine.

Architecture-Neutral

- Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction.
- The **Java Virtual Machine** is the solution to this situation. Their goal was (WORA)

"**write once; run anywhere, any time, forever**."

Data Types

Data Types

- Data Types are mainly classified into 4 types:
  - »Integers
  - »Floating point
  - »Characters
  - »Booleans

Data Types

Integers

- Bytes - 8 bits
- Short - 16 bits
- Int - 32 bits
- Long - 64 bits

Data Types

Floating point

- Float- 32 bits
- Double- 64 bits

Characters – 16 bits

Booleans

- True
- False

Variable

Variable

Initialization

- Static: The value of the variables does not change. Example, int x=5.
- Dynamic: Initialization at the time of declaration at runtime. Example taking values from the user.

Scope & Life Time of Variables

Scope & Life Time of Variables

Scope of block

{

}

- Variables are created when they enter the scope first

Scope & Life Time of Variables

Scope of methods

{

}

- Variables declared within the scope are not visible outside

Scope & Life Time of Variables

Scope of class

{

}

- Variables are destroyed when scope is entered and destroyed when scope is lift.

Type Conversion and Casting

Type Conversion and Casting

Automatic Conversion

- Conditions:
  - ✓Possible if variables of like nature
  - ✓The destination type is larger than the source type.
  - ✓When these two conditions are met, a *widening conversion takes place*
  - ✓*int a = byte b;*

Type Conversion and Casting

- The following rules takes place:
  - Numeric variables are compatible
  - Numeric variables are not compatible with char, Boolean.
  - Char & Boolean are not compatible
  - Auto conversion takes place while assigning integers, literals to byte, short, long.

Type Conversion and Casting

- Casting
  - Auto conversion is not useful when conversion like integers to byte is needed.
  - Casting is useful for narrowing conversions
  - A cast is simply explicit type conversion
  - Ex:

```
byte b;
int a;
b = (byte) a
```

Type Conversion and Casting

Automatic Type Promotion in Expressions

- When two operands of lower size are operated, the resultant value may far exceed the size of operand.
- Byte & short operands are automatically promoted to int.

Type Conversion and Casting

- If expression is in float, then its sub expression is also promoted to float
- If expression is in double, then its sub expression is also promoted to double.

Array

Array

- Like typed variables referred by a common name
- May have one or more dimensions
- A specific element of an array is referred by an index
- Offers convenient means of grouping related information.

Array

- Syntax

  <data type> <Variable name> [];
- Only defines variables not the size.
- Size is allocated by using a ’new’ operator.

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

In Java all arrays are dynamically allocated

Array

- Array elements are referred by index.
- Index value will start from zero
- It is possible to define and allocate size simultaneously.

```
int monthdays[] = new int[12];
```

- Arrays can be initialized
    - Comma separated
    - Enclosed in curly bracket

Array

- Array size can be defined by initialization.

  int monthdays[] = { 31, 28 …….}
- JAVA checks the boundaries at run time

Multi Dimension Array

- In Java, *multidimensional arrays are actually arrays of arrays.*
- There are a couple of subtle differences.
- To declare a multidimensional array variable, specify each additional index using another set of square brackets.

Multi Dimension Array

- For example, the following declares a two dimensional array variable called **twoD.**

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

- Multi dimension array examples are matrix multiplication, matrix addition, matrix subtraction, etc.

LTC: one-dimensional array.

```
// Demonstrate a one-dimensional array.
class Array {
public static void main(String args[]) {
int month_days[];
month_days = new int[12];
month_days[0] = 31;
month_days[1] = 28;
month_days[2] = 31;
```

```
month_days[3] = 30;
month_days[4] = 31;
month_days[5] = 30;
month_days[6] = 31;
month_days[7] = 31;
month_days[8] = 30;
month_days[9] = 31;
month_days[10] = 30;
month_days[11] = 31;
System.out.println(“May has”+ month_days[4] );
} }
```

```
public class MatrixMultiplicationExample{
public static void main(String args[])
{ //creating two matrices
int a[][]={{1,1,1},{2,2,2},{3,3,3}};
int b[][]={{1,1,1},{2,2,2},{3,3,3}};
//creating another matrix to store the multiplication of two matrices
int c[][]=new int[3][3];  //3 rows and 3 columns

//multiplying and printing multiplication of 2 matrices
for(int i=0;i<3;i++){
for(int j=0;j<3;j++){
c[i][j]=0;
for(int k=0;k<3;k++)
{
c[i][j]+=a[i][k]*b[k][j];
}//end of k loop
System.out.print(c[i][j]+" ");  //printing matrix element
}//end of j loop
System.out.println();//new line
}
}}
```

Operators

Operators

Operators:

- Used to develop Expression

Types of operators:

- Arithmetic
- Bitwise
- Relational
- Logical

Arithmetic Operators:

Arithmetic Operators(Binary):

+

-

*****

/

% mod returns remainder

Arithmetic Operators:

Arithmetic Operators(Unary):

+ +	Increment
--	Decrement
+=	Assignment (Efficiently)
-=	
***=**	
/=	
%=	

Arithmetic Operators-Cont…

Increment operators can be prefix

X=42;

Y=++x;

Result

X=43

Y=43

Arithmetic Operators-Cont…

Increment operators can be postfix

X=42;

Y=x++;

Result

x=43

Y=42

The Bitwise Operators

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

Relational operators

==

!=

>

<

>=

<=

outcome is a Boolean value

The Precedence of the Java Operators

Highest			
( )	**[ ]**	**.**	
++	– –	**~**	**!**
*****	**/**	**%**	
+	**–**		
>>	**>>>**	**<<**	
>	**>=**	**<**	**<=**
==	**!=**		
&			
^			
|			
&&			
||			
?:			
=	**op=**		
Lowest			

Expression

Expression

- Expressions are essential building blocks of any Java program, usually created to produce a new value, although sometimes an expression assigns a value to a variable.
- Expressions are built using values, variables, operators and method calls.

Types of expressions

- While an expression frequently produces a result, it doesn't always.
- There are three different types of expression:
  - Those that produce a value, i.e, the result of

    (1+1)
  - Those that assign a variable, for example

    v=10

Types of expressions

- Those that have no result but might have a "side effect" because an expression can include a wide range of elements such as method invocations or increment operators that modify the state (i.e., memory) of a program.

Control statements

Control Statements

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:

1. Selection
2. iteration
3. jump

Control Statements

- **Selection statements** allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable.
- **Iteration statements** enable program execution to repeat one or more statements (that is, iteration statements form loops).

Control Statements

- **Jump statements** allow your program to execute in a nonlinear fashion.
- All of Java's control statements are examined here.

Selection statements

- if Statement
- Nested if Statement
- if-else-if Ladder
- Switch Statement
- Nested switch Statement

IF Statement

```
if(condition)
    Statement(s);
else
    Statement(s);
```

If-else-if ladder

if(*condition)*
    statement;
else if(*condition)*
    statement;
else if(*condition)*
    statement;
...
else
    statement;

If-else-if ladder

- Executed from Top to Bottom
- As soon as a True condition is traced the statements associated with trace condition are executed and the rest of the ladder is by passed.
- If none of the if condition is true final else statement will be executed

Nested if

```
public class NestedIfDemo {
public static void main(String[] args) {
    int age=18;
   int weight=60;
    if(age>=18){
       if(weight>50){
           System.out.println("You are eligible to donate blood");
       } else{
           System.out.println("You are not eligible to donate
    blood");
       }
   } else{
     System.out.println("Age must be greater than 18");
   }
} }
```

Switch Statement

switch (*expression) {*

case *value1:*

// statement sequence

break;

case *value2:*

// statement sequence

break;

…

Switch Statement

case *value N:*

// statement sequence

break;

default:

// default statement sequence

}

Switch Statement

- Expression must be of type byte, short, int, char
- Value must be of type expression
- Value must be a literal
- Break statement is optional
- If none of the case values tallies, default will be processed.

Switch Statement

- A set of statements can be executed related to different cases
- Switch can only test for equality
- No two cases can have same value.

Nested switch Statement

```
switch(count) {
case 1:
   switch(target) { // nested switch
        case 0:
                System.out.println("target is zero");
                break;
```

Nested switch Statement

```
case 1: // no conflicts with outer switch
   System.out.println("target is one");
   break;
}
break;
case 2: //
...
```

Iteration statements

1) while
2) do while
3) for

While

```
while(condition)
{
    ___
    ___
    ___

}
```

- **The body of the while or any other of Java's loops can be empty. (**A null statement**)**

do while

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

for statement

```
for (initialization; condition; iteration)
{
    ___
    ___
    ___
}
```

- In case of **only one statement,** no curly brackets are needed.

Iteration statement

- Variables declared in a loop is limited to the loop only
- Loop variables can be declared within the loop itself
- More than one statement can be included in Initialization & iteration of for loop statement.

Iteration statement

- Initialization or iteration portions can be absent in a for loop. The absence is indicated by a semi colon
- An infinite loop can be created by dropping all the portions.

LTC: Demo and execution of program to print 1 to 100 odd numbers

```
public class OddNumbers {
        public static void main(String args[] )
        {
                int i;
                for(i=1;i<=100;i+=2)
                {
                        System.out.println(i);
                }
        }
}
```

This program generates the following output:

1

3

5

7

9

11

Jump Statements

- Continue statement
- Return Statement
- Break statement

Continue Statement

- Helps in continuing with next iteration ignoring the rest of the loop code
- Continue can be specified to transfer control to a label
- A statement can be provided with a label

Return

- The return statement is used to explicitly return from a method.
- Return will return a value or does not return a value based on whether a method returns a value or not.

Jump Statement

Break

- Break will break a loop and control is passed to outside the loop

LTC: Demo and execution of program Using break to exit a while loop

```
public 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 is completed.");
}
}
```

This program generates the following output:

i: 0

i: 1

i: 2

i: 3

i: 4

i: 5

i: 6

i: 7

i: 8

i: 9

Loop is completed.

End of session

Compiling and running of sample programs in JAVA

Java SE 6

- One of the release of Java is called Java SE 6, and the material in this book has been updated to reflect the version of Java.
- With the release of Java SE 6, Sun once again decided to change the name of the Java platform.
- First, notice that the “2” has been dropped.

Java SE 6

- Thus, the platform now has the name Java SE, and the official product name is **Java Platform, Standard Edition 6**.
- As with J2SE 5, the 6 in Java SE 6 is the product version number.
- The internal, developer version number is 1.6.

Netbeans IDE 8.0.2

- The software netbeans IDE 8.0.2 will be used in this course.
- NetBeans IDE 8.0.2 provides out-of-the-box code analyzers and editors for working with the latest Java 8 technologies--Java SE 8, Java SE Embedded 8, and Java ME Embedded 8.

Netbeans IDE 8.0.2

- The IDE also has a range of new enhancements that further improve its support for Maven and Java EE with PrimeFaces; new tools for HTML5, in particular for AngularJS; and improvements to PHP and C/C++ support.

A First Simple Program

- let’s look at some actual Java programs.
- Let’s start by compiling and running the short sample program shown here.
- As you will see, this involves a little more work than you might imagine.

A First Simple Program

```
/*
This is a simple Java program.
Call this file "Example.java".
*/
class Example {
// Your program begins with a call to main().
public static void main(String args[]) {
System.out.println("This is a simple Java program.");
}
}
```

Entering the Program

- Entering the Program
- In Java, a source file is officially called a compilation unit.
- It is a text file that contains one or more class definitions.
- The Java compiler requires that a source file use the .java filename extension.

Entering the Program

- The name of the class defined by the program is also Example.
- All code must reside inside a class.
- By convention, the name of that class should match the name of the file that holds the program.
- The **capitalization** of the filename matches the class name. The reason for this is that Java is case-sensitive.

Compiling a JAVA program.

- The **javac** compiler creates a file called Example.
- class that contains the bytecode version of the program.
- The Java bytecode is the intermediate representation of the program that contains instructions the Java Virtual Machine will execute.
- Save the program.
- Click on 

To run the program

- Go to file and then go to src.
- Then you find Example.java.
- Right click on Example.java.
- Then you will find run file.
- Click on it. Then your program will be executed.
- You can see the output at the bottom of your program.

Display of output

- When the program is run, the following output is displayed:

  This is a simple Java program.

The next line of code in the program is shown here:

class Example {

- This line uses the keyword class to declare that a new class is being defined.
- Example is an identifier that is the name of the class.
- The entire class definition, including all of its members, will be between the opening curly brace ({) and the closing curly brace (}).

The next line of code in the program is shown here:

public static void main(String args[]) {

- This line begins the main( ) method.
- The keyword **public** is an **access specifier**, which allows the programmer to control the **visibility** of class members.
- When a class member is preceded by public, then that member may be accessed by code outside the class in which it is declared.

The next line of code in the program is shown here:

- The keyword **static** allows main( ) to be called **without having to instantiate** a particular instance of the class.
- This is necessary since main( ) is called by the Java Virtual Machine before any objects are made.
- The keyword **void** simply tells the compiler that main( ) does not return a value.

The next line of code in the program is shown here:

- In main( ), there is only one parameter- String args[ ] declares a parameter named args, which is an array of instances of the class String.
- Objects of type String store character strings.
- In this case, args receives any **command-line arguments** present when the program is executed.

The next line of code in the program is shown here:

```
System.out.println("This is a simple Java program.");
```

- Notice that it occurs inside main( ).
- This line outputs the string “This is a simple Java program.” followed by a new line on the screen.

- Output is actually accomplished by the built-in println( ) method.
- In this case, println( ) displays the string which is passed to it.
- The line begins with System.out. System is a predefined class that provides access to the system, and out is the output stream that is connected to the console.
- The first } in the program ends main( ), and the last } ends the Example class definition.

Concepts of classes and objects Declaring objects

Class

- It is a new data type.
- Objects of new data type can be created.
- Is a template.
- Object is an instance of a class.

Class

- A simplified general form of a class definition is shown here:

class *classname*

{

type instance-variable1;

type instance-variable2;

// …

type instance-variableN;

Class

type methodname1(parameter-list) {

// body of method

}

type methodname2(parameter-list) {

// body of method

}

// ...

type methodnameN(parameter-list) {

// body of method

}

}

Class

- Variables defined in a class are called **instance variables.**
- Code is contained in the **methods**.
- Methods process data referred by instance variables.
- Methods & variables defined in the class are called as **members of a class**.

Class

- The data, or variables, defined within a class are called ***instance variables***.
- a copy of the variables is created every time an object of the class is created.
- *Any class must be entirely defined in a single source file*

A Simple Class

Here is a class called Box that defines three instance variables: width, height, and depth.

Currently, Box does not contain any methods (but some will be added soon).

/* A program that uses the Box class.

Call this file BoxDemo.java

*/

Demo and execution of program that uses the Box class

```
class Box {
double width;
double height;
double depth;
}
// This class declares an object of type Box.
class BoxDemo {
public static void main(String args[]) {
Box mybox = new Box();
double vol;
```

A Simple Class

```
// assign values to mybox's instance variables
mybox.width = 10;
mybox.height = 20;
mybox.depth = 15;
// compute volume of box
vol = mybox.width * mybox.height * mybox.depth;
System.out.println("Volume is " + vol);
}
}
```

A Simple Class

To run this program, you must execute BoxDemo.class.

When you do, you will see the following output:

```
Volume is 3000.0
```

A Simple Class

- As stated earlier, each object has its own copies of the instance variables.
- This means that if you have two Box objects, each has its own copy of depth, width, and height.
- It is important to understand that changes to the instance variables of one object have no effect on the instance variables of another.

A Simple Class

For example, the following program declares two Box objects:

```
class Box {
double width;
double height;
double depth;
}
```

A Simple Class

```
class BoxDemo2 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
```

A Simple Class

```
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
```

A Simple Class

```
// compute volume of first box
vol = mybox1.width * mybox1.height *
    mybox1.depth;
System.out.println("Volume is " + vol);
// compute volume of second box
vol = mybox2.width * mybox2.height *
    mybox2.depth;
System.out.println("Volume is " + vol);
}
}
```

A Simple Class

The output produced by this program is shown here:

Volume is 3000.0

Volume is 162.0

As you can see, mybox1's data is completely separate from the data contained in mybox2.

A Simple Class

- Creating a Box object, you will use a statement like the following:

Box mybox = new Box(); // create a Box object called mybox

A Simple Class

- As mentioned earlier, each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class.
- Thus, every Box object will contain its own copies of the instance variables width, height, and depth.

A Simple Class

- To access these variables, you will use the *dot (.) operator.*
- *The dot operator links the name of the* object with the name of an instance variable.
- For example, to assign the width variable of mybox the value 100, you would use the following statement:

```
mybox.width = 100;
```

Declaration of Object

Declaring Objects

- When you create a class, you are creating a new data type.
- You can use this type to declare objects of that type. However, obtaining objects of a class is a two-step process.
- First, you must declare a variable of the class type. This variable does not define an object.
- Syntax: <classname><Objectname>;

Declaring Objects

- Instead, it is simply a variable that can *refer to an object.*
- *Second, you must acquire an actual, physical copy of the object* and assign it to that variable. You can do this using the new operator.

Syntax: <objectname> = new <classname>();

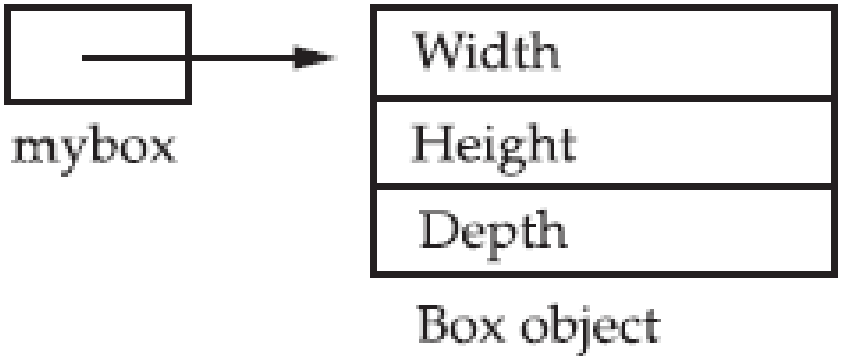
Declaring Objects

- It can be written like this to show each step more clearly:

  Box mybox; // declare reference to object

  mybox = new Box()

Declaring an object of type Box

Statement	Effect
Box mybox;	
mybox = new Box();	

A Closer Look at new

- As just explained, the new operator dynamically allocates memory for an object. It has this general form:

  class-var = new classname( );

Var- name of the object.

classname( )-Constructor.

Example:

Box mybox = new Box();

Declaring Objects

- The **new** operator **dynamically** allocates (that is, allocates at run time) memory for an object and returns a reference to it.
- This reference is, more or less,

  the address in memory of the object allocated by new. This reference is then stored in the variable.
- Thus, in Java,

  all class objects must be dynamically allocated.

Declaring Objects

- Let’s look at the details of this procedure.
- In the preceding sample programs, a line similar to the following is used to declare an object of type **Box:**
- Box mybox = new Box();
- This statement **combines the two steps** just described

Declaring an object of type Box

- Here, *class-var is a variable of the class type being created. The classname* is the name of the class that is being instantiated.
- The class name followed by parentheses specifies the *constructor* for the class.
- A constructor defines what occurs when an object of a class is created.

Declaring an object of type Box

- Most real-world classes explicitly define their own constructors within their class definition.
- However, if no explicit constructor is specified, then Java will automatically supply a

  default constructor

Declaring an object of type Box

- It is important to understand that new allocates memory for an object during run time.
- The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program.
- However, since **memory is finite**, it is possible that new will not be able to allocate memory for an object because insufficient memory exists.

Declaring an object of type Box

- If this happens, a **run-time exception** will occur.
- For the sample programs in this book, you won't need to worry about running out of memory, but you will need to consider this possibility in real-world programs that you write.

Distinction between a class and an object

- A class creates a new data type that can be used to create objects.
- That is, a class creates a logical framework that defines the relationship between its members.

Distinction between a class and an object

- When you declare an **object** of a class, you are creating an **instance** of that class.
- An object has physical reality. (That is, an object occupies space in memory.)

Constructors

Constructors

- It can be tedious to initialize all of the variables in a class each time an instance is created.
- Even when you add convenience functions like setDim( ), it would be simpler and more concise to have all of the setup done at the time the object is first created.

Constructors

- Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created.
- This automatic initialization is performed through the use of a **constructor.**

Constructors

- A *constructor initializes an object immediately upon creation.*
- *It has the same name as the* class in which it resides and is syntactically similar to a method.
- Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes.

Constructors

- Constructors look a little strange because they have no return type, not even void.
- This is because the implicit return type of a class. Constructor is the class type itself.
- It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

Constructors

- As you know, when you allocate an object, you use the following general form:

  class-var = new classname( );
- Now you can understand why the parentheses are needed after the class name.
- What is actually happening is that the constructor for the class is being called.

Constructors

- You can rework the Box example so that the dimensions of a box are automatically initialized when an object is constructed.
- To do so, replace setDim( ) with a constructor.
- Let’s begin by defining a simple constructor that simply sets the dimensions of each box to the same values.

Constructors

Three types

Constructors

- Default constructor:

Syntax:

<classname>()

{

//body of constructor

}

Parameterized Constructor

Syntax:

```
<class-name>(parameters-list)
{
//body of the constructor
}
/* Here, Box uses a parameterized constructor to
initialize the dimensions of a box.
*/
class Box {
double width;
double height;
double depth;
```

Parameterized Constructor

```
// This is the constructor for Box.
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
```

Parameterized Constructors

```
// compute and return volume
double volume() {
return width * height * depth;
}
}
```

Parameterized Constructors

```
class BoxDemo7 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 9);
double vol;
```

Parameterized Constructors

```
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

Parameterized Constructors

The output from this program is shown here:

```
Volume is 3000.0
Volume is 162.0
```

Parameterized Constructors

- As you can see, each object is initialized as specified in the parameters to its constructor.
- For example, in the following line,

  Box mybox1 = new Box(10, 20, 15);

Parameterized Constructors

- the values 10, 20, and 15 are passed to the Box( ) constructor when new creates the object.
- Thus, mybox1's copy of width, height, and depth will contain the values 10, 20, and 15 respectively.

Method

Method

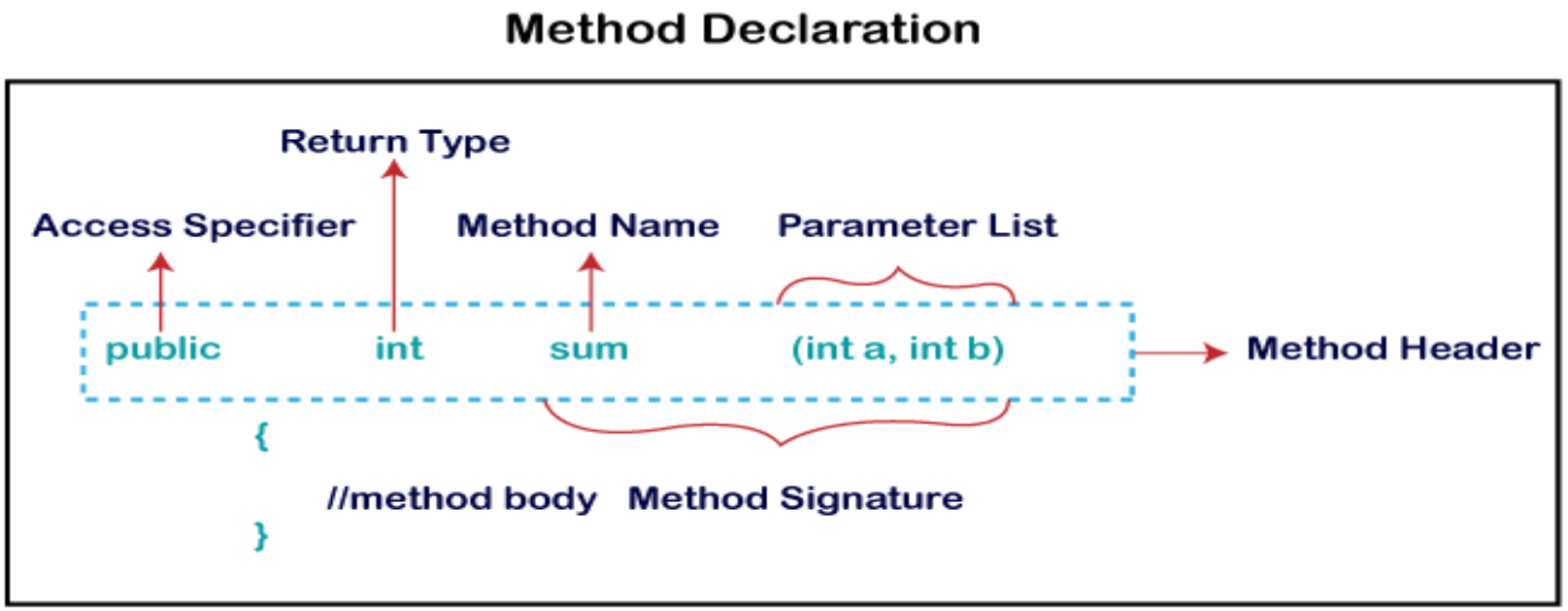
- A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation.
- It is used to achieve the **reusability** of code.
- We write a method once and use it many times.

Method cont…

- We do not require to write code again and again. It also provides the **easy modification** and **readability** of code, just by adding or removing a chunk of code.
- The method is executed only when we call or invoke it.
- The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments.

Method cont…

- It has six components that are known as **method header**, as we have shown in the following figure.



Access Control

Access Control

- As you know, encapsulation links data with the code that manipulates it.
- However, encapsulation provides another important attribute: *access control.*
- *Through encapsulation,* you can control what parts of a program can access the members of a class.
- By controlling access, you can prevent misuse.

Access Control

- How a member can be accessed is determined by the *access specifier that modifies its* declaration.
- Java supplies a rich set of access specifiers.
- Some aspects of access control are related mostly to inheritance or packages. (A *package is, essentially, a grouping of classes.).* These parts of Java's access control mechanism will be discussed later.

Access Control

- Java's access specifiers are public, private, and protected.
- Java also defines a default access level.
- protected applies only when inheritance is involved.

Access Control

- Let's begin by defining public and private.
- When a member of a class is specified as public , then that member can be accessed by any other class.
- When a member of a class is specified as private, then that member can only be accessed by other members of its class.

Access Control

- Now you can understand why main( ) has always been preceded by the public specifier.
- It is called by code that is outside the program—that is, by the Java run-time system.
- When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.

Access Control

- In the classes developed so far, all members of a class have used the default access mode, which is essentially public.
- However, this is not what you will typically want to be the case.

Access Control

- Usually, you will want to restrict access to the data members of a class—allowing access only through methods.
- Also, there will be times when you will want to define methods that are private to a class.

Access Control

- An access specifier precedes the rest of a member's type specification.
- That is, it must begin a member's declaration statement. Here is an example:

Access Control

```
/* This program demonstrates the difference
   between public and private. */
class Test {
int a; // default access
public int b; // public access
private int c; // private access
```

Access Control

```
// methods to access c
void setc(int i)
{ // set c's value
c = i;
}
int getc() {
// get c's value
return c;
}
}
```

Access Control

```
class AccessTest {
public static void main(String args[]) {
Test ob = new Test();
ob.a = 10;
ob.b = 20;
//ob.c = 100; // Error!
```

Access Control

```
ob.setc(100); // OK
System.out.println("a, b, and c: " + ob.a + " " +
ob.b + " " + ob.getc());
}
}
```

Access Control

- As you can see, inside the Test class, variable a uses default access, which for this example is the same as specifying public. b is explicitly specified as public.
- Member c is given private access.
- This means that it cannot be accessed by code outside of its class.
- So, inside the AccessTest class, c cannot be used directly.

Access Control

- It must be accessed through its public methods: setc( ) and getc( ).
- If you were to remove the comment symbol from the beginning of the following line,

  // ob.c = 100; // Error!

- then you would not be able to compile this program because of the access violation.

This key word

This key word

- Usage of java this keyword
- Here is given the 6 usage of java this keyword.
  - this can be used to refer current class instance variable.
  - this can be used to invoke current class method (implicitly)
  - this() can be used to invoke current class constructor.
  - this can be passed as an argument in the method call.
  - this can be passed as argument in the constructor call.
  - this can be used to return the current class instance from the method.

This key word Cont….

```
class Student
{
        int rollno;
        String name;
        float fee;
        Student(int rollno,String name,float fee)
        {
                this.rollno=rollno;
                this.name=name;
                this.fee=fee;
        }
```

This key word Cont….

```
void display()
        {
                System.out.println(rollno+" "+name+" "+fee);
        }
}
class TestThis2
{
        public static void main(String args[])
        {
                Student s1=new Student(111,"ankit",5000f);
                Student s2=new Student(112,"sumit",6000f);
                s1.display();
                s2.display();
        }
}
```

Garbage Collection

Garbage Collection

- Since objects are dynamically allocated by using the new operator, you might be wondering how such objects are destroyed and their memory released for later reallocation.
- In some languages, such as C++, dynamically allocated objects must be manually released by use of a delete operator.

Garbage Collection

- Java takes a different approach; it handles deallocation for you automatically.
- The technique that accomplishes this is called *garbage collection*.
- *It works like this: when no* references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.

Garbage Collection

- There is no explicit need to destroy objects as in C++.
- Garbage collection only occurs sporadically (if at all) during the execution of your program.
- It will not occur simply because one or more objects exist that are no longer used.

Garbage Collection

- Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

Overloading methods

Overloading Method

- In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different.
- When this is the case, the methods are said to be *overloaded, and the process is referred to as method overloading.*
- *Method* overloading is one of the ways that Java supports **polymorphism**.

Overloading Method-cont…

- If you have never used a language that allows the overloading of methods, then the concept may seem strange at first.
- But as you will see, method overloading is one of Java's most exciting and useful features.

Overloading Method-cont…

- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.
- Thus, overloaded methods must differ in the type and/or number of their parameters.

Overloading Method-cont…

- While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.
- When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Overloading Method

Here is a simple example that illustrates method overloading:

```
// Demonstrate method overloading.
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
// Overload test for one integer parameter.
void test(int a) {
System.out.println("a: " + a);
}
```

Overloading Method

```
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// overload test for a double parameter
double test(double a) {
System.out.println("double a: " + a);
return a*a;
}
}
```

Overloading Method

```
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
double result;
```

Overloading Method

```
// call all versions of test()
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of ob.test(123.25): " +
   result);
}
}
```

Overloading Method

This program generates the following output:

No parameters

a: 10

a and b: 10 20

double a: 123.25

Result of ob.test(123.25): 15190.5625

Overloading Method

- test( ) is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one double parameter.
- The fact that the fourth version of test( ) also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution.

Overloading Method

- When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters.
- However, this match need not always be exact.
- In some cases, Java's automatic type conversions can play a role in overload resolution.

Overloading Method

- For example, consider the following program:

```
// Automatic type conversions apply to
   overloading.
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
```

Overloading Method

```
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// overload test for a double parameter
void test(double a) {
System.out.println("Inside test(double) a: " + a);
}
}
```

Overloading Method

```
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
int i = 88;
ob.test();
ob.test(10, 20);
ob.test(i); // this will invoke test(double)
ob.test(123.2); // this will invoke test(double)
}
}
```

Overloading Method

This program generates the following output:

No parameters

a and b: 10 20

Inside test(double) a: 88

Inside test(double) a: 123.2

Overloading Method

- this version of OverloadDemo does not define test(int).
- Therefore, when test( ) is called with an integer argument inside Overload, no matching method is found.
- However, Java can automatically convert an integer into a double, and this conversion can be used to resolve the call.

Overloading Method

- Therefore, after test(int) is not found, Java elevates i to double and then calls test(double). Of course, if test(int) had been defined, it would have been called instead.
- Java will employ its automatic type conversions only if no exact match is found

Overloading Method

- When you overload a method, each version of that method can perform any activity you desire.
- There is no rule stating that overloaded methods must relate to one another.
- However, from a stylistic point of view, method overloading implies a relationship.
- Thus, while you can use the same name to overload unrelated methods, you should not.

Overloading Method

- For example, you could use the name sqr to create methods that return the *square of an integer* and the *square root of a floating-point value.*
- *But these two operations are fundamentally* different.
- Applying method overloading in this manner defeats its original purpose.
- In practice, you should only overload closely related operations.

Parameter passing

Call by value

- In general, there are two ways that a computer language can pass an argument to a subroutine (method).
- The first way is *call-by-value.*
- *This approach copies the value of an argument into the formal* parameter of the subroutine.
- Therefore, changes made to the parameter of the subroutine have no effect on the argument.

Call by value

- The second way an argument can be passed is *call-by-reference*.
- In this approach, a reference to an argument (not the value of the argument) is passed to the parameter.
- Inside the subroutine, this reference is used to access the actual argument specified in the call.

Call by value

- This means that changes made to the parameter will affect the argument used to call the subroutine.
- As you will see, Java uses both approaches, depending upon what is passed.

Call by value

- In Java, when you pass a primitive type to a method, it is passed by value.
- Thus, what occurs to the parameter that receives the argument has no effect outside the method.

Call by value

- For example, consider the following program:

```
// Primitive types are passed by value.
class Test {
void meth(int i, int j) {
i *= 2;
j /= 2;
}
}
```

Call by value

```
class CallByValue {
public static void main(String args[]) {
Test ob = new Test();
int a = 15, b = 20;
```

Call by value

```
System.out.println("a and b before call: " +
a + " " + b);
ob.meth(a, b);
System.out.println("a and b after call: " +
a + " " + b);
}
}
```

Call by value

- The output from this program is shown here:

a and b before call: 15 20

a and b after call: 15 20

Call by reference

- When you pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference.
- Keep in mind that when you create a variable of a class type, you are only creating a reference to an object.

Call by reference

- Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument.
- This effectively means that objects are passed to methods by use of call-by-reference.
- Changes to the object inside the method *do affect the object used as an* argument.

LTC:Call by reference

- For example, consider the following program:

```
// Objects are passed by reference.
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
```

Call by reference

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

Call by reference

```
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);
}
}
```

Call by reference

- This program generates the following output:

ob.a and ob.b before call: 15 20

ob.a and ob.b after call: 30 10

- As you can see, in this case, the actions inside meth( ) have affected the object used as an argument.

Call by reference

REMEMBER

1. *When a primitive type is passed to a method, it is done by use of call-by-value.*
2. *Objects are implicitly passed by use of call-by-reference.*

Recursion

Recursion

- Java supports *recursion.*
- *Recursion is the process of defining something in terms of itself.*
- *As* it relates to Java programming, recursion is the process that allows a method to call itself.
- A method that calls itself is said to be *recursive.*

Recursion

- The classic example of recursion is the computation of the factorial of a number.
- The factorial of a number *N is the product of all the whole numbers between 1 and N.*
- *For* example, 3 factorial is 1 × 2 × 3, i.e., 6.
- Here is how a factorial can be computed by use of a recursive method:

Recursion

```
// A simple example of recursion.
class Factorial {
// this is a recursive method
int fact(int n) {
int result;
if(n==1) return 1;
result = fact(n-1) * n;
return result;}}
```

Recursion

```
class Recursion {
public static void main(String args[]) {
Factorial f = new Factorial();
System.out.println("Factorial of 3 is " + f.fact(3));
System.out.println("Factorial of 4 is " + f.fact(4));
System.out.println("Factorial of 5 is " + f.fact(5));
}
}
```

Recursion

- The output from this program is shown here:

Factorial of 3 is 6

Factorial of 4 is 24

Factorial of 5 is 120

Recursion

- fact( ) is called with an argument of 1, the function returns 1; otherwise, it returns the product of fact(n–1)*n.
- To evaluate this expression, fact( ) is called with n–1.
- This process repeats until n equals 1 and the calls to the method begin returning.

Recursion

- To better understand how the fact( ) method works, let's go through a short example.
- When you compute the factorial of 3, the first call to fact( ) will cause a second call to be made with an argument of 2.
- This invocation will cause fact( ) to be called a third time with an argument of 1.

Recursion

- This call will return 1, which is then multiplied by 2 (the value of n in the second invocation).
- This result (which is 2) is then returned to the original invocation of fact( ) and multiplied by 3 (the original value of n).

Recursion

- This yields the answer, 6.
- You might find it interesting to insert println( ) statements into fact( ), which will show at what level each call is and what the intermediate answers are.

Recursion

- The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives.

Exploring String

Exploring String Cont…

- In Java, string is basically an object that represents sequence of char values.
- An array of characters works same as Java string. For example:

```
char[] ch={'j','a','v','a','t','p','o','i','n','t'};
String s=new String(ch);
```

- Is same as:

```
String s="javatpoint";
```

Exploring String Cont…

- **Java String** class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

Exploring String Cont…

- Examples

```
package r22;

 class Sc {
    public static void main(String[] args)
{
String s1 = new String("Hello");
String s2 = new String("Hello");
System.out.println("string length is: "+s1.length());
System.out.println(s1.substring(1,4));
System.out.println(s1.equals(s2));
System.out.println(s1.equals(s2));
```

Exploring String Cont…

```
System.out.println(s1.compareTo(s2));
String str5 = s1.concat(s2);
System.out.println(str5);
String replaceString=s1.replace('o','i');
System.out.println(replaceString);
String replaceString1=s1.replaceAll("Hello","Hi");
System.out.println(replaceString1);
```

Exploring String Cont…

```
 System.out.println(s1.startsWith("He"));
 System.out.println(s1.trim()+" sasi");
 String s3=s1.toLowerCase();
 System.out.println(s3);
String s1upper=s1.toUpperCase();
System.out.println(s1upper);
  String substr = s1.substring(3); // Starts with 0 and goes to end
      System.out.println(substr);
}
}
```

Exploring String Cont…

OUTPUT
string length is: 5
ell
true
true
0
HelloHello
Helli
Hi
true
Hello sasi
hello
HELLO
lo