Java Programming

V Semester B. Sc Computer Science

Reference:

The Complete Reference: Java2

By

Herbert Schildt

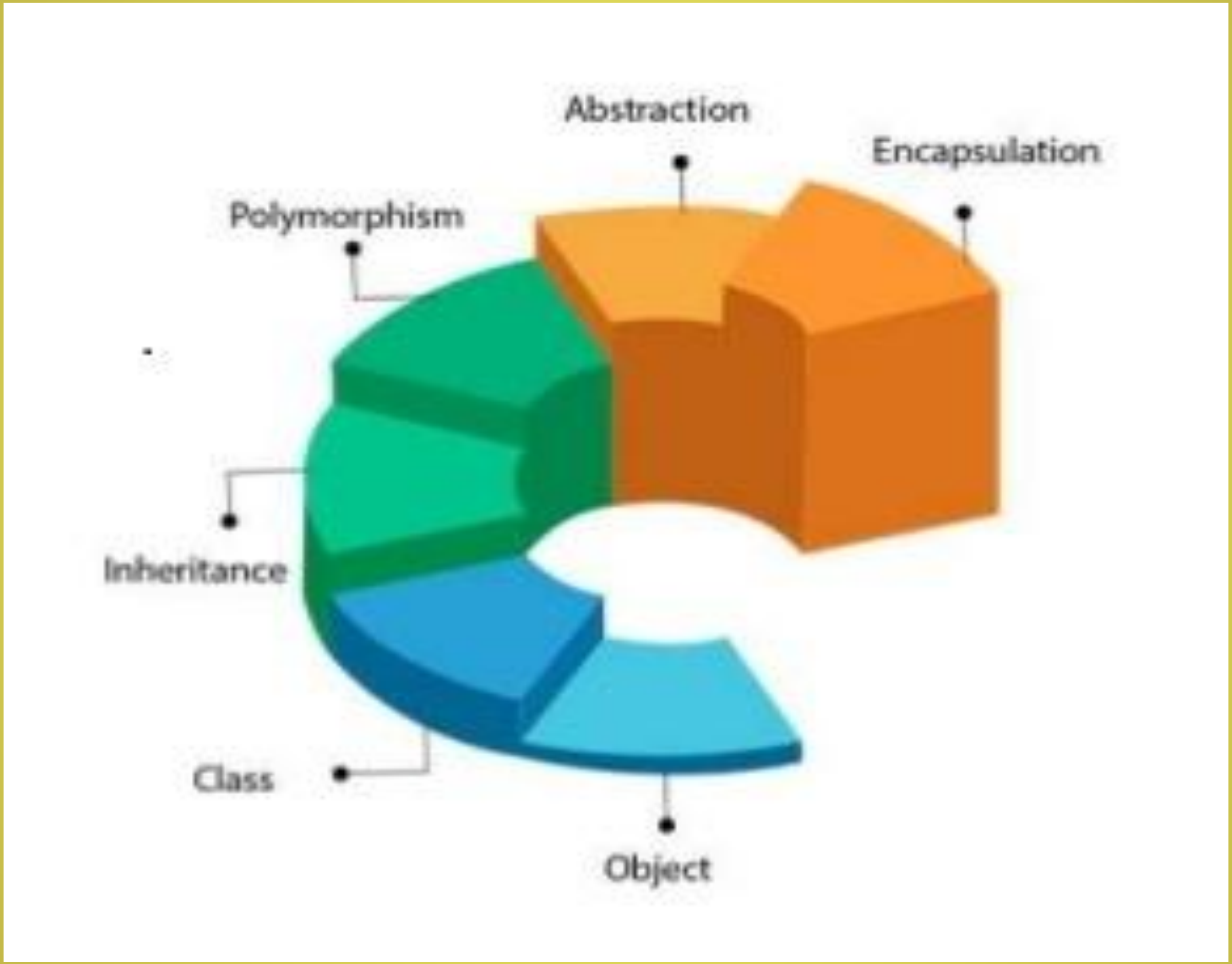
Rincy T A
Assistant Professor
Department of Computer Science
Prajyoti Niketan College, Pudukad

OBJECT ORIENTED PROGRAMMING

- The object-oriented paradigm is a programming methodology that promotes the efficient design and development of software systems using reusable components that can be quickly and easily assembled into larger systems.

Aim

- The main aim of object-oriented programming is to implement real-world concepts like
  - Object -> real world entity
  - Classes -> blue print
  - Abstraction -> Visibility Controls
  - Inheritance -> Parent Child Relation
  - Polymorphism -> Many forms


Abstraction
Encapsulation
Polymorphism
Inheritance
Class
Object

Why OOP

- Applications are more manageable and predictable
- Code Reusability
- Model real things more easily

Principles of OOP

- Class
  - A class is a group of objects which have some common properties. It is a blue print from which objects are created.
- Object
  - An object is an instance of a class. Any entity that has state and behaviour is known as an object
    - Eg: Bench, Per, Car, Table etc..

- Inheritance
  - Inheritance is a mechanism in which one object acquires all the properties and behaviours of a parent object.
  - It is a Parent- Child Relationship (IS-A Relationship)
    - Animal is a Mammal, Reptiles or Birds
  - Terms Used:
    - Sub class/ Child class/Derived class
    - Super class/ Parent class/ Base class

- Code Reusability
  - We can reuse the existing class fields and methods when you create a new class.
  - Types of Inheritance
    - Single : Class Y -> Class X
    - Multilevel : Class Z -> Class Y -> Class X
    - Hierarchical -> Class Z-> Class X, Class Y -> Class X
    - Multiple : Class Y -> Class X, Class Y -> Class Z (not supported by Java)
    - Hybrid

- Polymorphism
  - Ability to take more than one form
  - If one task is performed by different ways.
    - Compile Time Polymorphism
      - Overloading
    - Run Time Polymorphism
      - Overriding
  - Eg:

    getPrice()

    getPrice(String name)

- Encapsulation
  - It is the integration of data and operations into a class
  - Data Hiding
    - Eg: A Capsule
      - Can you drive a bus?
        - » Yes, I Can
      - So, how does acceleration work?
        - » Huh?

- Abstraction
  - It is giving the access to the functionality of a class while hiding the implementation details.

Overview

- Java programming language was originally developed by Sun Microsystems which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems' Java platform (Java 1.0 [J2SE]).
- Java is guaranteed to be Write Once, Run Anywhere.

Java is:

- Object Oriented: In Java, everything is an Object. Java can be easily extended since it is based on the Object model.
- Platform Independent: Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on.

- Simple: Java is designed to be easy to learn. If you understand the basic concept of OOP Java, it would be easy to master.
- Secure: With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.

- Architecture-neutral: Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.
- Portable: Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable.

- Robust: Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
- Multithreaded: With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that can run smoothly.

- Interpreted: Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.
- High Performance: With the use of Just-In-Time compilers, Java enables high performance.

- Distributed: Java is designed for the distributed environment of the internet.
- Dynamic: Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

History of Java

- James Gosling initiated Java language project in June 1991 for use in one of his many settop box projects. The language, initially called ‘Oak’ after an oak tree that stood outside Gosling's office, also went by the name ‘Green’ and ended up later being renamed as Java, from a list of random words.
- Sun released the first public implementation as Java 1.0 in 1995. It promised Write Once, Run Anywhere (WORA), providing no-cost run-times on popular platforms.
- On 13 November, 2006, Sun released much of Java as free and open source software under the terms of the GNU General Public License (GPL).
- On 8 May, 2007, Sun finished the process, making all of Java's core code free and opensource, aside from a small portion of code to which Sun did not hold the copyright.

Try It

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

```
public class MyFirstJavaProgram
{
    /* This is my first java program.
     This will print 'Hello World' as the output */
    public static void main(String []args)
    {
    System.out.println("Hello World");
        // prints Hello World
    }
}
```

- Open notepad and add the code as above.
- Save the file as: MyFirstJavaProgram.java.
- Open a command prompt window and go to the directory where you saved the class. Assume it's C:\.
- Type 'javac MyFirstJavaProgram.java' and press enter to compile your code. If there are no errors in your code, the command prompt will take you to the next line (Assumption : The path variable is set).
- Now, type ' java MyFirstJavaProgram ' to run your program.
- You will be able to see ' Hello World ' printed on the window.

```
C:\> javac MyFirstJavaProgram.java
C:\> java MyFirstJavaProgram
     Hello World
```

About Java programs, it is very important to keep in mind the following points.

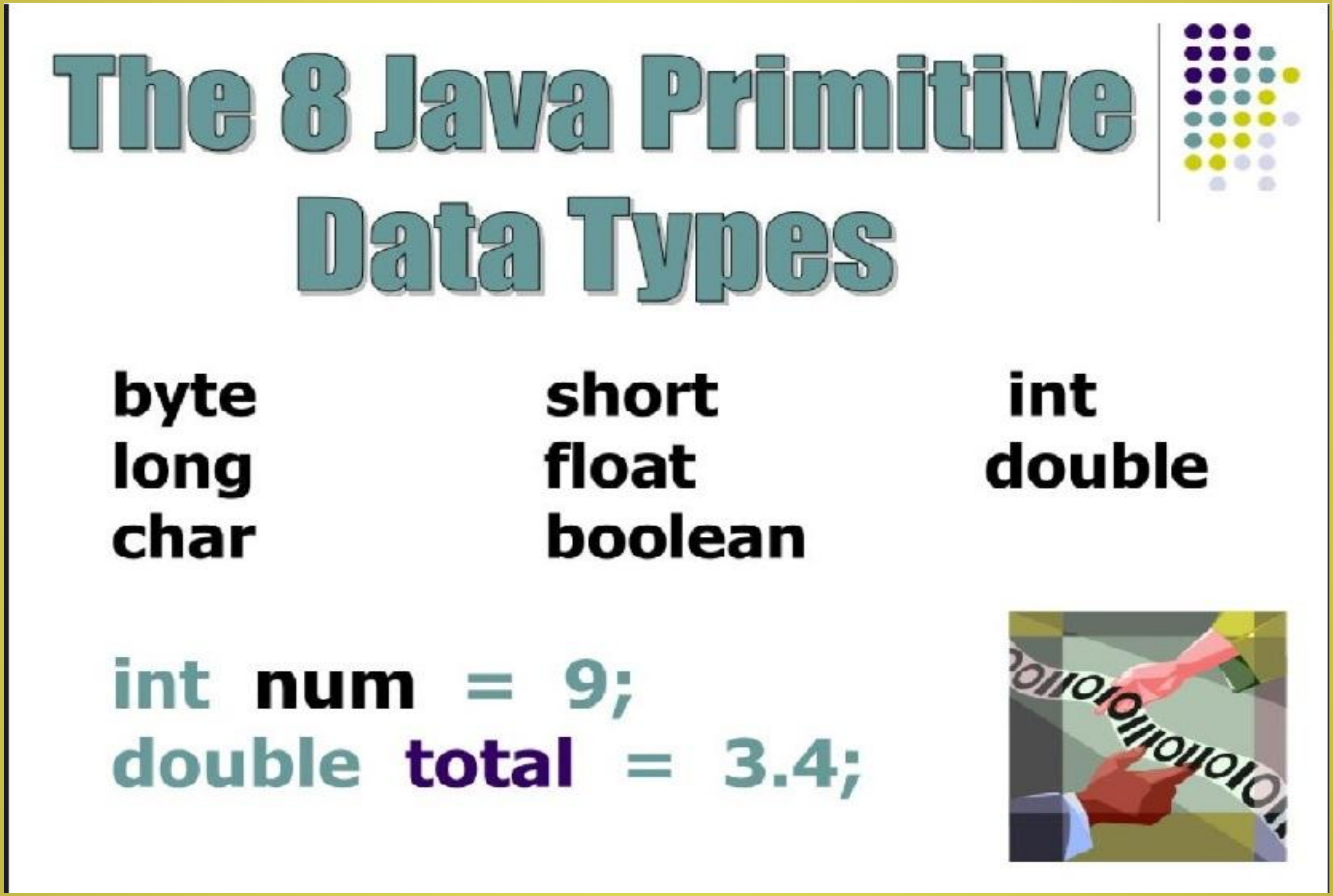
- Case Sensitivity - Java is case sensitive, which means identifier Hello and hello would have different meaning in Java.
- Class Names - For all class names the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case. Example: class MyFirstJavaClass

- Method Names - All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case. Example: public void myMethodName()

- Program File Name - Name of the program file should exactly match the class name. When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and the class name do not match, your program will not compile). Example: Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as 'MyFirstJavaProgram.java‘

- public static void main(String args[]) - Java program processing starts from the main() method which is a mandatory part of every Java program.

Primitive Data Types



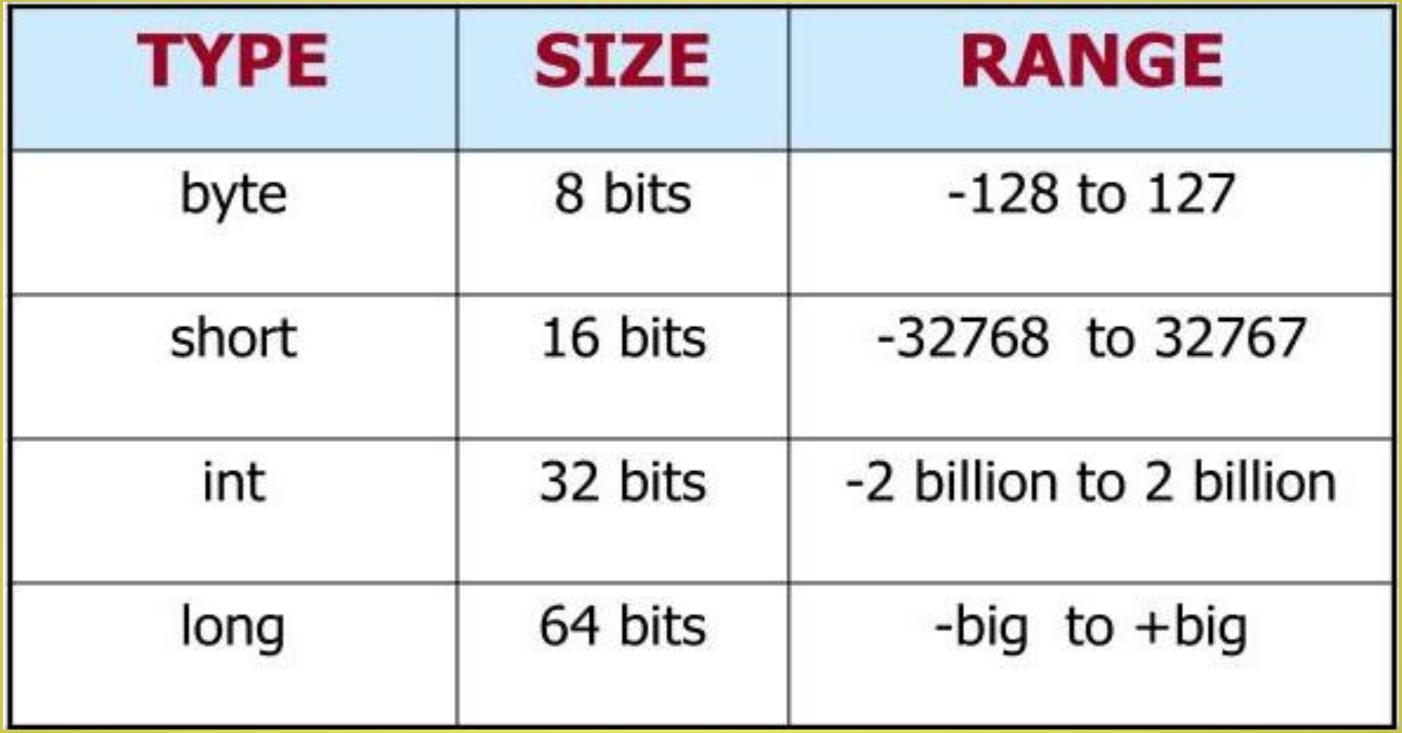
The 8 Java Primitive Data Types

byte	short	int
long	float	double
char	boolean	

```
int num = 9;
double total = 3.4;
```

Integers

- byte
- short
- int
- long
- All these are signed, positive and negative values.



TYPE	SIZE	RANGE
byte	8 bits	-128 to 127
short	16 bits	-32768 to 32767
int	32 bits	-2 billion to 2 billion
long	64 bits	-big to +big

Example:

```
byte b,c;
short s;
short t;
int lightspeed;
long days;
long seconds;
```

Floating-Point Types

- They are known as Real numbers
- Used when evaluating expressions that require fractional precision

Name	Width in bits
double	64
float	32

Example:

```
double pi,r,a;
```

Characters

- Java uses Unicode to represent characters.
- char is a 16-bit type
- Range is from 0 to 65536.

Example:

```
char ch1,ch2;
ch1=88; //code for X
ch2=‘Y’;
```

Booleans

- Possible values: true or false

Example:

```
    boolean b;
    b=false;
```

Lexical Issues

- Whitespace
  - Java is a free-form language
  - You do not need to follow any special indentation rules.
  - In Java, whitespace is a space, tab or newline.

- Identifiers
  - Used for class names, method names, and variable names.
  - May be any descriptive sequence of letters, numbers, or the underscore and dollar-sign characters.
  - Must not begin with a number.

- Literals
  - A constant value in java is created by using a literal representation of it.

Example:

100   98.6   'X' "This is test"

- Character Literals
  - Character Escape Sequences

Escape Sequence	Description
\ddd	Octal Character
\uxxxx	Heaxadecimal Unicode character
\’	Single Quote
\”	Double Quote
\\	Backslash
\r	Carriage Return
\n	New Line
\t	Tab
\b	Backspace

- Separators

()	Parentheses
{}	Braces
[]	Brackets
;	Semicolon
,	Comma
.	Period

- Keywords
  - There are 49 reserved keywords
  - Keywords cannot be used as names for a variable, class or method.

Example:

break	else	new	short
static	class	super	while

Variables

- Basic unit of storage in a Java Program
- All variables must be declared before they can be used.
- All variables have scope
- Syntax for variable declaration:

```
type identifier[= value][, identifier[= value]...];
```

Example:

```
int a,b,c;
int d=3,e,f=5;
char x='Y';
double pi=3.14159;
```

Dynamic Initialization

- Java allows variables to be initialised dynamically, using any expression valid at the time the variable is declared.

```
class Example2
{
      public static void main(String []args)
      {
            int num;
            num=100;
            System.out.println("This is num: "+num);
            num=num*2;
            System.out.println("The value of num*2 is "+num);
            double a=3.0,b=4.0;
            double c=Math.sqrt(a*a + b*b);
            System.out.println("Hypotenuse is "+c);
      }
}
```

Output:

```
This is num: 100
The value of num*2 is 200
Hypotenuse is 5.0
```

Scope and Lifetime of variables

- A block defines 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.
- Scopes can be nested.

```
class Example3
{
        public static void main(String []args)
        {
                int num;
                num=100;
                if(num==100)
                {
                        int y=10;
                        System.out.println("This is num*y: "+num*y);
                }
                //y=100; //Error
                num=num*2;
                System.out.println("The value of num*2 is "+num);
        }
}
```

Output:

```
This is num*y: 1000
The value of num*2 is 200
```

Type conversion and casting

- It is fairly common to assign a value of one type to a variable of another type.
- If the two types are compatible, then Java will perform the conversion automatically.
- It is possible to assign an int value to a long variable.
- There is no automatic conversion from double to byte.

Automatic conversions

- It will take place if the following conditions are met:
  - The two types are compatible
  - The destination type is larger than the source type
- If the conditions are met, then a widening conversion takes place

Casting incompatible types

- Suppose you want to assign an int value to a byte variable.
- Its a narrowing conversion
- A cast is simply an explicit type conversion

  (target-type) value

Note: target-type specifies the desired type to convert the specified value to.

- The following fragment casts an int to a byte

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

- When a floating point value is assigned to an integer type, truncation takes place
- 1.23 is truncated to 1

Automatic type promotion in expressions

Example:

```
byte a=40;
byte b=50;
byte c=100;
int d=a*b/c;
```

As useful as the automatic promotions are, they can cause confusing compile-time errors.

```
byte b=50;
b=b*2; //Error
```

But

```
byte b=50;
b=(byte)(b*2); //Correct
```

Operators

Arithmetic Operators

+	addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition Assignment,c+=a is equivalent to c=c+a
-=	Subtraction Assignment
*=	Multiplication Assignment
/=	Division Assignment
%=	Modulus Assignment
--	Decrement

- Modulus Operator:

  Returns the remainder of a division operator

```
class Modulus
{
        public static void main(String args[])
        {
                int x=42;
                double y=42.25;
                System.out.println(x%10);
                System.out.println(y%10);
        }
}
```

Output:

```
2
2.25
```

- Arithmetic assignment operators

  Any statement of the form

  ```
  var=var op expression;
  ```

  can be rewritten as

  ```
  var op=expression;
  ```

Example:

```
c += a*b;
```

is equivalent to

```
c= c + a*b;
```

- Increment and Decrement
  x=x+1 is equivalent to x++ (postfix)
  x=x-1 is equivalent to x- -

Example:

```
x=42;
y=++x; (Prefix)          // here y=43

x=42;
y=x++; (postfix)  //here y=42
```

Relational Operators

==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Boolean Logical Operators

&	Logical AND
|	Logical OR
^	Logical EXOR
||	Short-circuit OR
&&	Short-circuit AND
!	Logical Unary NOT
&=	AND Assignment
|=	OR Assignment
^=	XOR Assignment
==	Equal to
!=	Not Equal to
?:	Ternary if-then-else

- Logical Operators

```
a=true;
b=false;
a|b=true
a&b=false
a^b=true
!a&b | a&!b=true
!a=false
```

- Short-circuit logical operators:

  In the case of these operators, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left-operand alone.

Example:

If(denom !=0 && num/denom>10)

Assignment Operator

= Assignment

Syntax:

```
var=expression;

Int x,y,z;
x=y=z=100;
```

The ? operator

- Ternary operator

Syntax:

exp1?exp2:exp3;

exp1 can be any expression that evaluates to a boolean value.

Ratio= denom==0? 0 : num/denom;

Bitwise Operators

- These operators can be applied to the integer types, long, int, short, char and byte.

Operators:

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

```
int a=3;                //a=0011
int b=6;                //b=0110
int c=a | b;                //c=0111
int d=a & b;                //d=0010
int e=a ^ b;                    //e=0101
int f=(~a & b) | (a & ~b);   //f=0101
int g=~a & 0x0f;        //g=1100
int h=64;               //h=1000000
int i=h<<2;                 //i=256
int j=64;               //j=1000000
int k=j>>2;                 //k=16
```

The precedence of Java Operators

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

Arrays

- Java provides a data structure, the array, which stores a fixed-size sequential collection of elements of the same type.
- An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

- Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.

Declaring Array Variables

- Declare a variable to reference the array, and specify the type of array the variable can reference.

```
dataType[] arrayRefVar; // preferred way.
 or
dataType arrayRefVar[];
     // works but not preferred way.
```

- Example:

```
double[] myFruits;
      or
double myFruits[];
```

Creating Arrays

- You can create an array by using the new operator with the following syntax:

  arrayRefVar = new dataType[arraySize];

- The above statement does two things:
  - It creates an array using

    new dataType[arraySize];

  - It assigns the reference of the newly created array to the variable arrayRefVar

- Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below:

```
dataType[] arrayRefVar = new dataType[arraySize];
```

- Alternatively you can create arrays as follows:

```
dataType[] arrayRefVar = {value0, value1, ..., valuek};
```

The array elements are accessed through the index. Array indices are 0-based; that is, they start from 0 to arrayRefVar.length-1.

- Example:

```
double[] myFruits = new double[10];

int[] month_days=new int[12];
          or
int[] month_days;
month_days=new int[12];

month_days[0]=31;
month_days[1]=28;
…………
```

- Initialising an array:
  - Example:

```
int month_days[]={31,28,31,30};
System.out.println("April has "+month_days[3]+"days");
```

```
class Example2
{
	public static void main(String []args)
	{
		int month_days[]={31,28,31,30};
		System.out.println("April has "+month_days[3]+" days");
		int sum_of_days=0;
		for(int i=0;i<4;i++)
			sum_of_days=sum_of_days+month_days[i];

		System.out.println("Total days: "+sum_of_days);
	}
}
```

Output:

```
April has 30 days
Total days: 120
```

Multidimensional arrays

- Arrays of arrays
- Declaring a two-dimensional array

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

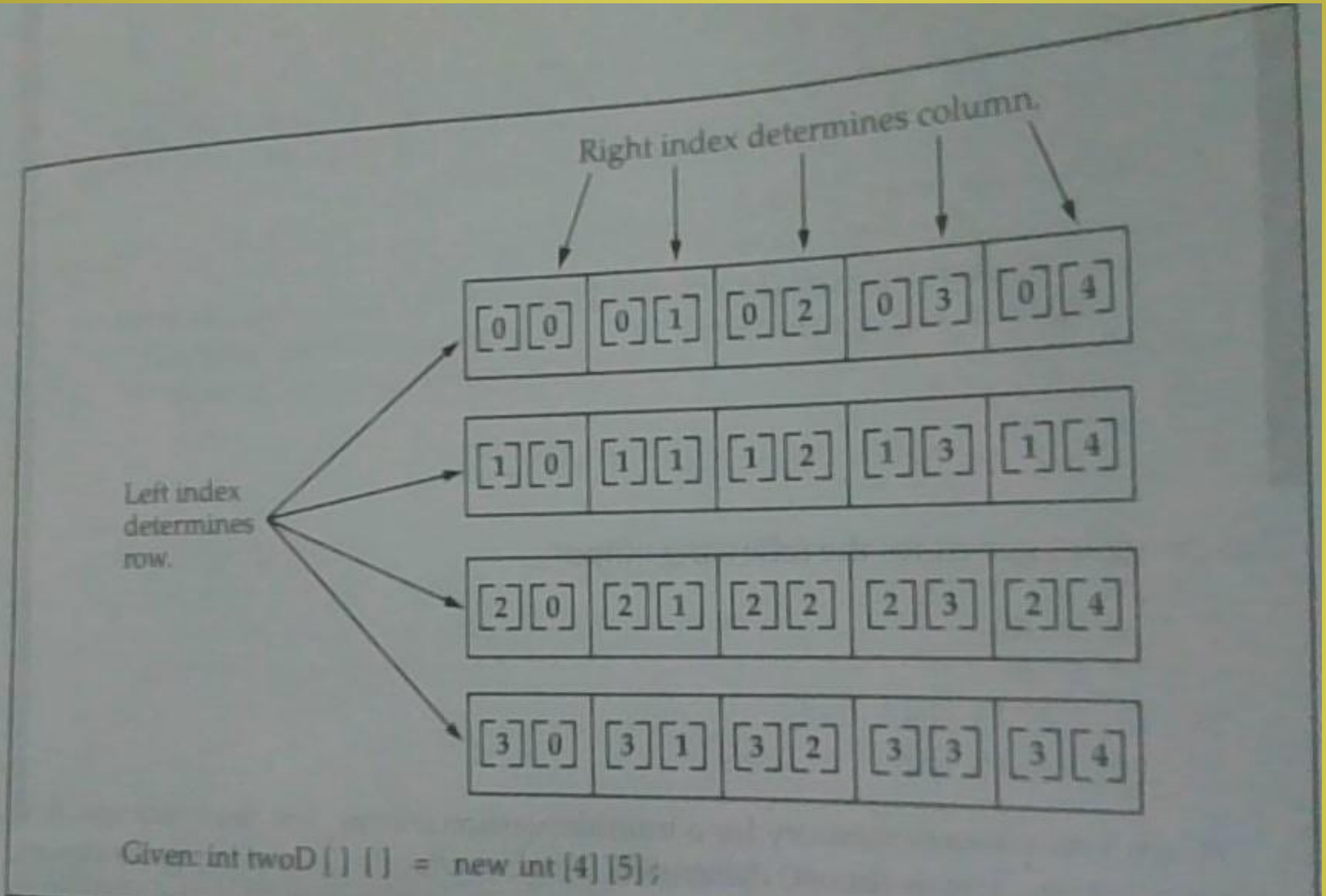


Figure 3-1 A conceptual view of a 4 by 5, two-dimensional array

```
class Example5
{
        public static void main(String []args)
        {
                int twoD[][]=new int[4][5];
                int i, j,k=0;
                for(i=0;i<4;i++)
                        for(j=0;j<5;j++)
                        {
                                twoD[i][j]=k;
                                k++;
                        }
                for(i=0;i<4;i++)
                {
                        for(j=0;j<5;j++)
                        {
                                System.out.print(twoD[i][j]+" ");
                        }
                        System.out.println();
                }
        }
}
```

```
Output:
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

- When you allocate memory for a multidimensional array, you need only specify the memory for the first dimension.
- Example:

```
int twoD[][]=new int[4][];
twoD[0]=new int[5];
twoD[1]=new int[5];
twoD[2]=new int[5];
twoD[3]=new int[5];
```

- We can create arrays with unequal second dimension.

```
int twoD[][]=new int[4][];
twoD[0]=new int[1];
twoD[1]=new int[2];
twoD[2]=new int[3];
twoD[3]=new int[4];
```

```
0
1   2
3   4   5
6   7   8   9
```

- Initialisation of multidimensional arrays:

```
double m[][]={
            {0*0, 1*0, 2*0, 3*0},
            {0*1, 1*1, 2*1, 3*1},
            {0*2, 1*2, 2*2, 3*2},
            {0*3, 1*3, 2*3, 3*3}
             };
```

Reading Console Input

- Console input is accomplished by reading from System.in

- To obtain a character-based stream that is attached to the console, you wrap System.in in a BufferedReader object, to create a character stream.

- Reader is an abstract class. One of its concrete subclass is InputStreamReader, which converts bytes to characters.

- Putting it altogether, the following line of code creates a BufferedReader that is connected to the keyboard.

  ```
  BufferedReader br= new BufferedReader(new InputStreamReader(System.in));
  ```

  After this statement executes, br is a character-based stream that is linked to the console through System.in

Reading characters

- The version used for read() is:

  int read() throws IOException

  Each time that read() is called, it reads a character from the input stream and returns as an integer value. It returns -1 when the end of the stream is encountered. It can throw IOException.

Example Program:

```
import java.io.*;
class BRRead
{
	public static void main(String []args) throws IOException
	{
		char c;
		BufferedReader br= new BufferedReader(new
	InputStreamReader(System.in));
		System.out.println("Enter characters, 'q' to quit.");
		do
		{
			c=(char)br.read();
			System.out.print(c);
		}while(c!='q');
	}
}
```

```
C:\>java BRRead
Enter characters, 'q'
to quit.
ghjghq
ghjghq
```

Reading Strings

- The version used for readLine() is:

  String readLine() throws IOException

  It reads a String from the keyboard, and it returns a String object.

Example Program:

```
import java.io.*;
class BRReadLines
{
	public static void main(String []args) throws IOException
	{
		String str;
		BufferedReader br= new BufferedReader(new 
   InputStreamReader(System.in));
		System.out.println("Enter Lines of text, ‘stop' to quit.");
		do
		{
			str=br.readLine();
			System.out.println(“String is:”+str);
		}while(!str.equals(“stop”));
	}
}
```

```
C:\>java BRReadLines
Enter Lines of text, 'stop'
to quit.
Hello
String is:Hello
How are you?
String is:How are you?
stop
String is:stop
```

Reading Integers

```
import java.io.*;
class ReadInt
{
        public static void main(String []args) throws IOException
        {
                int a;
                BufferedReader br= new BufferedReader(new
    InputStreamReader(System.in));
                System.out.println("Enter numbers, -1 to quit.");
                do
                {
                        a=Integer.parseInt(br.readLine());
                        System.out.println("Value is:"+a);
                }while(a!= -1);
        }
}
```

```
C:\>java ReadInt
Enter numbers, -1 to quit.
3
Value is:3
5
Value is:5
-1
Value is:-1
```

Class Fundamentals

- •Class defines a new data type.
- •This new type can be used to create objects of that type.
- •A class is a template for an object, and an object is an instance of a class.
- •It defines the shape and nature of an object.

The General Form of a Class

```
class classname{
type instance-variable1;
type instance-variable2;
.
.
type instance-variableN;
type methodname1(parameter-list){
//body of method
}
type methodname2(parameter-list){
//body of method
}
.
.
}
}
```

- **A class contains data(instance variables) and the code that operate on the data(methods).**
- **A class code defines the interface to its data.**
- **Each instance of the class contains its own copy of the data variables.**

A simple Class

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

To create an object of this class, use the statement

```
Box mybox=new Box();
Box yourbox=new Box();
```

A simple Class

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

- Each object of this class contain its own copies of the instance variables defined by the class.
- To access these variables use the dot(.) operator.

```
mybox.width=100;
mybox.height=50;
mybox.depth=20;
yourbox.width=80;
```

An Example Program

```
class Box
{
        double width;
        double height;
        double depth;
}
class BoxDemo
{
        public static void main(String args[])
        {
                Box mybox=new Box();
                double vol=0;
                mybox.width=100;
                mybox.height=50;
                mybox.depth=20;
                vol= mybox.width*mybox.height*mybox.depth;
                System.out.println("Volume is " + vol);
        }
}
```

An Example Program

```
class Box
{
	double width;
	double height;
	double depth;
}
class BoxDemo
{
	public static void main(String args[])
	{
		Box mybox1=new Box();
		Box mybox2=new Box();
		double vol=0;
```

```
                mybox1.width=10;
                mybox1.height=20;
                mybox1.depth=15;
                mybox2.width=3;
                mybox2.height=6;
                mybox2.depth=9;
                vol=mybox1.width*mybox1.height*mybox1.depth;
                System.out.println(“Volume is “ + vol);
                vol=mybox2.width*mybox2.height*mybox2.depth;
                System.out.println(“Volume is “ + vol);
        }
}
```

Declaring objects

Obtaining objects of a class is a two step process.

First, declare a variable of the class type.

Second, acquire an actual, physical copy of the object and assign it to that variable(new operator).

The new operator dynamically allocates memory for an object and returns a reference to it.

```
 Box mybox1=new Box();
```

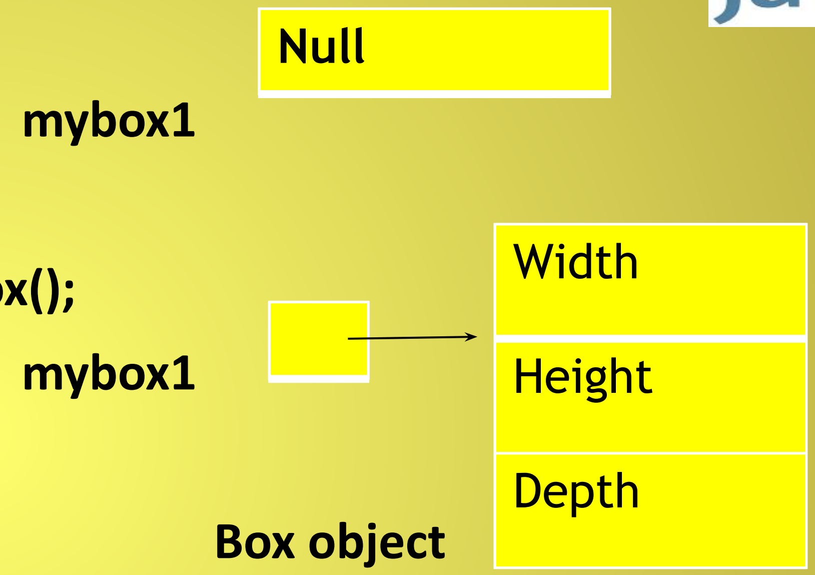
can be rewritten as

```
Box mybox1;
mybox1=new Box();
```

Statement **Effect**

Box mybox1;

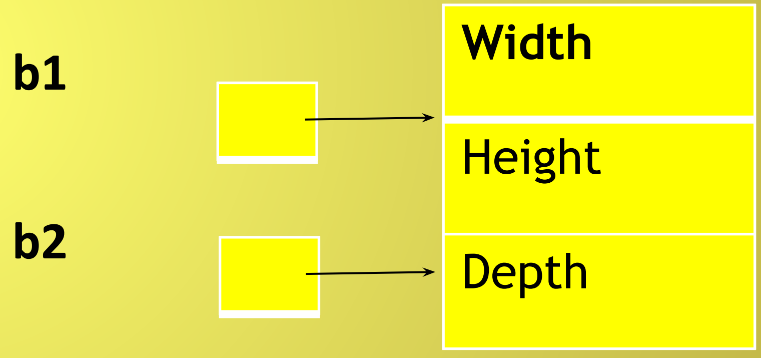
mybox1=new Box();



Assigning object reference variable

Consider the following fragment of code:

```
Box b1=new Box();
Box b2=b1;
```



Box object

Introducing methods

- Methods have so much power and flexibility in Java
- General Form:

```
type methodname(parameter-list)
{
//body of method
}
```

- type specifies the type of data returned by the method.
- If the method does not return a value- void.
- Parameter-list is a sequence of type and identifier pairs separated by commas.
- No parameters- list is empty.
- Methods return value using the syntax:

```
return value;
```

Adding a method to the Box class

```
class Box
{
    double width;
    double height;
    double depth;
    void volume()
    {
        System.out.print(“Volume is “);
        System.out.println(width*height*dept
  h);
    }
}
```

```
class BoxDemo3
{
    public static void main(String args[])
    {
        Box mybox=new Box();
        mybox.width=100;
        mybox.height=50;
        mybox.depth=20;
        mybox.volume();
    }
}
```

Returning a value

- A better way to implement volume() is to have it compute the volume of the box and return the result to the caller.

```
class Box
{
      double width;
      double height;
      double depth;
      double volume()
      {
            return width*height*depth;
      }
}
```

```
class BoxDemo4
{
    public static void main(String args[])
    {
        Box mybox=new Box();
        double vol=0;
        mybox.width=10;
        mybox.height=20;
        mybox.depth=15;
        vol=mybox.volume();
        System.out.print(“Volume is “+vol);
    }
}
```

Adding a method that takes parameters

- A parameterized method can operate on a variety of data.
- Parameterless method:

```
int square()
{
    return 10*10;
}
```

- A method with one parameter

```
int square(int i)
{
        return i*i;
}
```

square() is now a general-purpose method that can compute the square of any integer value.

```
int x,y;

x=square(5);

x=square(9);

y=2;

x=square(y);
```

A better approach

```
class Box
{
      double width;
      double height;
      double depth;
      double volume()
      {
            return width*height*depth;
      }
      void setDim(double w, double h, double d)
      {
            width=w;
            height=h;
            depth=d;
      }
}
```

```
class BoxDemo5
{
    public static void main(String args[])
    {
        Box mybox=new Box();
        double vol=0;
        mybox.setDim(10,20,15);
        vol=mybox.volume();
        System.out.print("Volume is "+vol);
    }
}
```

Recursion

- Recursion is the process of defining something in terms of itself.
- It allows a method to call itself.
- A method that calls itself is said to be recursive.

Example Program

```
class Factorial
{
        int fact(int n)
        {
                int result;
                if(n==1)
                    return 1;
                result=fact(n-1)*n;
                return result;
        }
}
class Example5
{
        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 5 is: "+f.fact(5));
                System.out.println("Factorial of 7 is: "+f.fact(7));
        }
}
```

```
C:\>java Example5
Factorial of 3 is: 6
Factorial of 5 is: 120
Factorial of 7 is: 5040
```

- When 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.
- When a method calls itself, new local variables and parameters are allocated storage on the stack, and method code is executed with these new variables from start.

Introducing access control

- Encapsulation provides an important attribute: access control
- You can control what parts of the program can access the members of the class. Thus you can prevent the misuse of data.
- Java's access specifiers are:
  - private
  - public
  - protected

- public
  - When a member of a class is modified by the public specifier, then that member can be accessed by any other code.
- private
  - When a member of the class is specified as private, then that member can only be accessed by other members of its class.
- Example:

```
	public int i;
	private double j;
	private int myMethod(int a, char b){…..}
```

Example program

```
class Test
{
    int a;
    public int b;
    private int c;
    void setc(int
  i)
    {
            c=i;
    }
    int getc()
    {
        return c;
    }
}
```

```
class Example5
{
    public static void main(String []args)
    {
        Test ob=new Test();
        ob.a=1;
        ob.b=20;
        //ob.c=100;  //Error
        ob.setc(100);
        System.out.println("a, b, and c:
"+ob.a+
                "  "+ob.b+" "+ob.getc());
    }
}
```

```
C:\>java Example5
a, b, and c: 1 20
100
```

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, a constructor is automatically called immediately after the object is created, before the new operator completes.
- They have no return type, not even void.

Example Program

```
class Box
{
        double width;
        double height;
        double depth;
        Box()
        {
                System.out.println(“Constructing Box”);
                width=10;
                height=10;
                depth=10;
        }
        double volume()
        {
                return width*height*depth;
        }
}
```

```
class BoxDemo
{
    public static void main(String []args)
    {
        Box mybox=new Box();
        Box yourbox=new Box();
        double vol=0;
        vol=mybox.volume();
        System.out.println("Volume is "+vol);
        vol=yourbox.volume();
        System.out.println("Volume is "+vol);
    }

}
```

Parameterized Constructors

- In the previous example, all boxes have the same dimension.
- Suppose we want to construct Box objects with different dimensions.
- The easy solution is to add parameters to constructors.

Example Program

```
class Box
{
        double width;
        double height;
        double depth;
        Box(double w, double h, double d)
        {
                width=w;
                height=h;
                depth=d;
        }
        double volume()
        {
                return width*height*depth;
        }
}
```

```
class BoxDemo
{
    public static void main(String []args)
    {
        Box mybox=new Box(10,20,15);
        Box yourbox=new Box(12,14,16);
        double vol=0;
        vol=mybox.volume();
        System.out.println("Volume is "+vol);
        vol=yourbox.volume();
        System.out.println("Volume is "+vol);
    }

}
```

This keyword

- this can be used inside any method to refer to the current object.
- this is always a reference to the object on which the method was invoked.
- Example:

```
// a redundant use of this
Box(double w, double h, double d)
{
      this.width=w;
      this.height=h;
      this.depth=d;
}
```

- Instance variable hiding:
  - It is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes.
  - You can have local variables, including formal parameters to methods, which overlap with the names of the class instance variables.
  - However, when a local variable has the same name as the instance variable, local variable, hides the instance variable.
  - this lets you to refer directly to the object, you can use it to resolve any name space collisions that might occur between instance variables and local variables.

```
Box(double width, double height,
  double depth)
    {
        this.width=width;
        this.height=height;
        this.depth=depth;
    }
```

Using objects as parameters

- So far we have only been using simple types as parameters to methods
- It is common to use objects as parameters
- There are two ways to pass an argument to a subroutine
- Call-by-value:
  - The method copies the value of an argument into the formal parameter of the subroutine.
  - Therefore the changes made to the parameter of the subroutine have no effect on the argument.
  - When you pass a simple type to a method, it is passed by value.

- Call-by-reference
  - Changes made to the parameter will effect the argument used to call the subroutine.
  - When you pass an object to a method, the situation changes dramatically, because objects are passed by reference.
- Example program involves passing objects as arguments to methods as well as to constructors.

Example program

```
class Box
{
        double width;
        double height;
        double depth;
        Box()
        {
                width=10;
                height=10;
                depth=10;
        }
        Box(double w, double h, double d)
        {
                width=w;
                height=h;
                depth=d;
        }
```

```
    Box(Box ob)
    {
            width=ob.width;
            height=ob.height;
            depth=ob.depth;
    }
    double volume()
    {
            return width*height*depth;
    }
    boolean equals(Box o)
    {
            if(o.width==width && o.height==height &&
  o.depth==depth)
                    return true;
            else
                    return false;
    }
}
```

```
class Example5
{
	public static void main(String []args)
	{
		Box mybox1=new Box();
		Box mybox2=new Box(10,20,15);
		Box myclone=new Box(mybox1);
		double vol;
		vol=mybox1.volume();
		System.out.println("Volume of mybox1 is "+vol);
		vol=mybox2.volume();
		System.out.println("Volume of mybox2 is "+vol);
		System.out.println("mybox1==myclone:"+mybox1.equals(mycl
  one));
		System.out.println("mybox2==myclone:"+mybox2.equals(mycl
  one));
	}
}
```

```
C:\>java Example5
Volume of mybox1 is 1000.0
Volume of mybox2 is 3000.0
mybox1==myclone:true
mybox2==myclone:false
```

Returning Objects

- A method can return any type of data, including class types that you create.
- Example Program:

```
class Box
{
    double width;
    double height;
    double depth;
Box(double w,double h,double d)
    {
        width=w;
        height=h;
        depth=d;
    }
    Box incrByTen()
    {
        Box temp=new Box(width+10, height+10, 
  depth+10);
        return temp;
    }
}
```

```
class Example5
{
    public static void main(String []args)
    {
        Box mybox1=new Box(2,3,4);
        Box mybox2;
        mybox2=mybox1.incrByTen();
        System.out.println("Mybox2 dimensions are:"+
  mybox2.width+    ","+
  mybox2.height+","+mybox2.depth);
    }
}
```

```
C:\>java Example5
Mybox2 dimensions
are:12.0,13.0,14.0
```

Introducing nested and inner classes

- It is possible to define a class within another class.
- The scope of a nested class is bounded by the scope of its enclosing classes.
- Thus class B is defined within class A, then B is known to A, but not outside of A.
- A nested class has access to the members, including private members, of the class in which it is nested.
- Enclosing class does not have access to the members of the nested class.
- You can define inner classes within any block scope(Eg: within the body of the for loop)

- The most important type of nested class is the inner class.

```
class Outer
{
	int a=42;
	void test()
	{
			Inner inner=new Inner();
			inner.display();
	}
	class Inner
	{
		int y=10;
		void display()
		{
		System.out.println("Outer-a: " +a);
		}
	}
```

```
void showy()
	{
	//System.out.println(y); //Error
	}
}
class InnerClassDemo
{
	public static void main(String[] args)
	{
		Outer outer=new Outer();
		outer.test();
	}
}
```

```
C:\>java  InnerClassDemo
Outer-a: 42
```

Using Command Line Arguments

- Sometimes you want to pass information into a program when you run it.
- A command line argument is the information that directly follows the program's name on the command line when it is executed.
- They are stored as strings in the String array passed to main().

```
class CommandLine
{
public static void main(String args[])
 {
for(int i=0;i<args.length;i++)
    System.out.println(“args[“ + i +”]: ” +
  args[i]);
}
}
```

```
C:\java CommandLine this is a test 100
-1
args[0] : this
args[1] : is
args[2] : a
args[3] : test
args[4] : 100
args[5] : -1
```

```
import java.io.*;
import java.util.Scanner;
class Demorgan {
int x,y,i,j;
void readNumbers() {
Scanner Sc = new Scanner(System.in);
System.out.println("x : ");
x = Sc.nextInt();
System.out.println("y : ");
y = Sc.nextInt();
System.out.println("i : ");
i = Sc.nextInt();
System.out.println("j : ");
j = Sc.nextInt();
}
```

```
void FirstLaw() {
 boolean result;
 result = !(!(x<5) && !(y>=7));
 System.out.println("!(AB) =
"+result);
 result = ((x<5) || (y>=7));
 System.out.println("!A + !B =
"+result);
 }
```

```
void SecondLaw() {
 boolean result;
 result = !((i>4) || (j <=6));
 System.out.println("!(A+B) = "+result);
 result = (!(i>4) && !(j <=6));
 System.out.println("!A!B = "+result);
 }
 }
```

```
public class Demor {
public static void
main(String[] args) {
Demorgan obj = new
Demorgan();
obj.readNumbers();
obj.FirstLaw();
obj.SecondLaw();
}
}
```

```
C:\>java Demor
x :
3
y :
4
i :
5
j :
6
!(AB) = true
!A + !B = true
!(A+B) = false
!A!B = false
```

Understanding static

- Static data members are used independently of any object of class.
- When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.
- Most common example of a static method is main(). main() is declared as static because it must be called before any objects exist.
- Static variables are global variables
- All instances of the class share the same static variable.

- Methods declared as static have several restrictions:
  - They can only call other static methods
  - They must only access static data
  - They cannot refer to this or super in any way

Example

```
class Example5
{
    static int a=3;
    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("Sta
tic block
initialised ");
        b=a*4;
  }
  public static void
main(String []args)
  {
        meth(42);}}
```

- As soon as the class is loaded, all of the static statements are run. First a is set to 3, then the static block executes, and finally, b is initialised to a*4 or 12. The main() is called, which calls meth(), passing 42 to x.

```
C:\>java Example5
Static block initialised
x= 42
a= 3
b= 12
```

- Outside the class in which they are defined, static methods and variables can be used independently of any object.
- You need only specify the name of their class followed by the dot operator.
- Syntax:

```
classname.method()
```

- classname is the name of the class in which the static method is declared.

Example program

```
class StaticDemo
{
    static int a=42;
    static int b=99;
    static void callme()
    { System.out.println("a= "+a);}
}
class Example5{
    public static void main(String[] args)
  {
        StaticDemo.callme();
        System.out.println("b =
  "+StaticDemo.b);
    }
}
```

```
C:\>java Example5
a= 42
b = 99
```

Introducing final

- A variable can be declared as final.
- Doing so prevents its contents from being modified.
- This means you must initialize a final variable when it is declared.

```
final int FILE_NEW=1;
final int FILE_OPEN=2;
final int FILE_SAVE=3;
```

- final variable is essentially a constant and it is a common convention to use uppercase identifiers for final variables

Overloading methods

- 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.
- The methods are said to be overloaded, and the process is referred to as method overloading.
- It is a way in which Java implements compile time Polymorphism.

Example Program

```
class OverLoadDemo
{
    void test()
    {
        System.out.println("No Parameters");
    }
    void test(int a)
    {
        System.out.println("a: "+a);
    }
    void test(int a, int b)
    {
        System.out.println("a and b: "+a+" "+b);
    }
    double test(double a)
    {
        System.out.println("double a: "+a);
        return a*a;
    }
}
```

```
class OLD
{
    public static void main(String []args)
    {
        OverLoadDemo ob=new OverLoadDemo();
        double result;
        ob.test();
        ob.test(10);
        ob.test(10,20);
        result=ob.test(123.25);
        System.out.println("Result is "+result);
    }
}
```

```
C:\>java OLD
No Parameters
a: 10
a and b: 10 20
double a: 123.25
Result is
15190.5625
```

Automatic type promotion while overloading

- Suppose a class have a the following methods
  - void test()
  - void test(int a, int b)
  - void test(double a)
- Suppose in the main(), we have a statement like

```
int i=88;
ob.test(i);
```

- We know that the above program does not define test(int), therefore no matching method is found.
- However, java can automatically convert an integer into double, and this conversion can be used to resolve the call.
- And it calls test(double).

- 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.
- The return type alone is insufficient to distinguish two versions of a method.

Overloading Constructors

```
class Box
{
	double width;
	double height;
	double depth;
	Box()
	{
		width=10;
		height=10;
		depth=10;
	}
	Box(double w, double h, double d)
	{
		width=w;
		height=h;
		depth=d;
	}
	Box(double len)
	{
		width=height=depth=len;
	}
double volume()
      {
              return width*height*depth;
      }
}
class BoxDemo
{
      public static void main(String []args)
      {
              Box mybox1=new Box();
              Box mybox2=new Box(10,20,15);
              Box mycube=new Box(7);
              double vol;
              vol=mybox1.volume();
              System.out.println("Volume of mybox1 is "+vol);
              vol=mybox2.volume();
              System.out.println("Volume of mybox2 is "+vol);
              vol=mycube.volume();
              System.out.println("Volume of mycube is "+vol);
      }
}
```

```
C:\>javac
BoxDemo.java

C:\Rincy>java Example5
Volume of mybox1 is
1000.0
Volume of mybox2 is
3000.0
Volume of mycube is
343.0
```

INHERITANCE

- Inheritance allows the creation of hierarchical classifications.
- Using inheritance you can create a general class that defines properties common to a set of related items.
- A class that is inherited is called a superclass.
- The class that does the inheriting is called a subclass.
- It inherits all the instance variables and methods defined by a superclass and adds its own, unique elements.

Inheritance Basics

- To inherit a class, use extends keyword

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 Example5
{
    public static void
  main(String[] args) {
        A superOb=new A();
        B subOb=new B();
        superOb.i=10;
        superOb.j=20;
        System.out.println("Conte
  nts of                 superOb:");
        superOb.showij();
        System.out.println();
        subOb.i=7;
        subOb.j=8;
        subOb.k=9;
        System.out.println("Conte
nts of                   subOb:");
        subOb.showij();
        subOb.showk();
        System.out.println();
        System.out.println("Sum
of          i, j and k in
subOb:");
        subOb.sum();
        }
}
```

```
C:\>java Example5
Contents of superOb:
i and j: 10  20

Contents of subOb:
i and j: 7  8
k: 9

Sum of i, j and k in subOb:
i+j+k= 24
```

- Even though A is a super class of B, it is also a stand alone, completely independent class.
- A subclass can be a super class of another class.
- Java does not support multiple inheritance.
- A subclass cannot access those members of the super class that have been declared as private.

```
class A
{
        int i;
        private int j;
        void showij()
        {
                System.out.println("i and j: "+i+"  "+j);
        }
}
class B extends A
{
        int k;
        void showijk()
        {
                System.out.println("i: "+i);
                //System.out.println("j: "+j);
                System.out.println("k: "+k);
        }
}
```

```
class Example5
{
    public static void main(String[] args) {
        B subOb=new B();
        subOb.i=7;
        //subOb.j=8; //Error
        subOb.k=9;
        System.out.println("Contents of superOb:");
        subOb.showij();
        System.out.println();
        System.out.println("SubOb: ");
        subOb.showijk();
    }
}
```

```
C:\>java Example5
Contents of subOb:
i and j: 7  0

SubOb:
i: 7
k: 9
```

Example

```
class Box
{
    double width;
    double height;
    double depth;
    double volume()
    {
        return width*height*depth;
    }
}
```

```
class BoxWeight extends Box
{
        double weight;
        BoxWeight(double w,double h,double d, double m)
        {
                width=w;
                height=h;
                depth=d;
                weight=m;
        }
}
class Example5
{
        public static void main(String[] args)
        {
                BoxWeight mybox1=new BoxWeight(10,20,30,40);
                BoxWeight mybox2=new BoxWeight(1,2,3,4);
                System.out.println("mybox1 volume= "+ mybox1.volume());
                System.out.println("mybox2 volume= "+ mybox2.volume());
        }
}
```

```
C:\>java Example5
mybox1 volume= 6000.0
mybox2 volume= 6.0
```

A superclass variable can reference a subclass object

- A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.
- It is very useful in a variety of situations.

```
class Box
{
    double width;
    double height;
    double depth;
    Box(){}
    double volume()
    {
        return width*height*depth;
    }
}
```

```
class BoxWeight extends Box
{
    double weight;
    BoxWeight(double w,double h,double d,
  double m)
    {
        width=w;
        height=h;
        depth=d;
        weight=m;
    }
}
```

```
class Example5
{
        public static void main(String[] args)
        {
                BoxWeight weightbox=new BoxWeight(10,20,30,40);
                Box plainbox=new Box();
                double vol;
                vol=weightbox.volume();
                System.out.println("weightbox volume= "+ vol);
                System.out.println("Weight of weightbox is: "+weightbox.weight);
                plainbox=weightbox;
                vol=plainbox.volume();
                System.out.println("plainbox volume= "+ vol);
                //System.out.println("Weight of plainbox is: "+plainbox.weight); //invalid
    statement
        }
}
```

- When a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass.
- This is why plainbox can't access weight even when it refers to a BoxWeight object.

```
C:\>java Example5
weightbox volume= 6000.0
Weight of weightbox is: 40.0
plainbox volume= 6000.0
```

Using super

- super has two general forms
- The first calls the superclass constructor
- The second is used to access a member of the superclass that has been hidden by a member of a subclass.

Using super to call superclass constructors

- A subclass can call a constructor method defined by its superclass by use of the following form of super:

  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 suclass' constructor.

```
class Box
{
        double width;
        double height;
        double depth;
        Box(double w,double h,double d)
        {
                width=w;
                height=h;
                depth=d;
        }
        double volume()
        {
                return width*height*depth;
        }
}
```

```
class BoxWeight extends Box
{
        double weight;
        BoxWeight(double w,double h,double d, double m)
        {
                super(w,h,d);
                weight=m;
        }
}
class Example5
{
        public static void main(String[] args)
        {
                BoxWeight weightbox=new BoxWeight(10,20,30,40);
                double vol;
                vol=weightbox.volume();
                System.out.println("weightbox volume= "+ vol);
        }
}
```

```
C:\>java Example5
weightbox volume= 6000.0
```

A second use for super

Syntax: super.member

It always refers to the superclass of the subclass in which it is used.

Here, a member can be either a method or an instance variable.

This second form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

```
class A
{
        int i;
}
class B extends A
{
        int i;
        B(int a, int b)
        {
        super.i=a;
        i=b;
        }
        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();
        }
}
```

```
C:\>java useSuper
i in superclass: 1
i in subclass: 2
```

When constructors are called

- In what order are the constructors for the classes that make up the hierarchy are called?
  - In a class hierarchy, constructors are called in order of derivation, from superclass to subclass
  - Since super must be the first statement executed in a subclass constructor, this order is the same whether or not super() is used.
  - If super() is not used, then the default parameterless constructor for each superclass will be executed.

```
class Test {
  Test()
  {
         System.out.println("Inside Test Constructor");
  }}
class TestSub extends Test{
         TestSub()
         {
super();
                  System.out.println("Inside TestSub Constructor");
         }}
class TestSubNew extends TestSub{
TestSubNew(){
super();
         System.out.println("Inside TestSubNew Constructor");
}}
class Main {
   public static void main(String args[]) {
     TestSubNew t=new TestSubNew();
}}
```

```
C:\>java Main
Inside Test Constructor
Inside TestSub Constructor
Inside TestSubNew Constructor
```

Using Abstract Classes

- There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.
- Sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.
- To declare an abstract method, use the general form

```
abstract type name(parameter-list);
```

- No method body is present.

- Any class that contains one or more abstract methods must also be declared abstract.
- To declare a class abstract, you simply use the abstract keyword in front of the class keyword at the beginning of the class declaration.
- There can be no objects of an abstract class
- That is, an abstract class cannot be directly instantiated with the new operator
- You cannot declare abstract constructors, or abstract static methods
- Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared abstract.

Example

```
abstract class A {
	abstract void callme();
	void callmetoo()
	{
		System.out.println("This is a concrete method");
	}
}
class B extends A{
	void callme()
	{
		System.out.println("B's implementation of callme");
	}
}
class Main
{
	public static void main(String[] args) {
		B b=new B();
		b.callme();
		b.callmetoo();
	}
}
```

```
C:\>Java Main
B's implementation of callme
This is a concrete method
```

- Although abstract classes cannot be used to instantiate objects, they can be used to create object references , because java’s approach to run-time polymorphism is implemented through the use of the superclass references.

Example:

```
abstract class Figure{
        double dim1;
        double dim2;
        Figure(double a, double b)
        {
                dim1=a;
                dim2=b;
        }
        abstract double area();
        }

class Rectangle extends Figure{
        Rectangle(double a,double b)
        {
                super(a,b);
        }
        double area()
        {
                System.out.println("Inside area of Rectangle");
                return dim1*dim2;
        }
}
```

```
class Triangle extends Figure
{
	Triangle(double a,double b)
	{
		super(a,b);
	}
	double area()
	{
		System.out.println("Inside area of Triangle");
		return dim1*dim2/2;
	}
}
class Main
{
	public static void main(String[] args) {
		Rectangle r=new Rectangle(9,8);
		Triangle t=new Triangle(10,8);
		System.out.println("Volume of Rectangle : "+r.area());
		System.out.println("Volume of Triangle : "+t.area());
	}
}
```

```
Inside area of
Rectangle
Volume of Rectangle :
72.0
Inside area of Triangle
Volume of Triangle :
40.0
```

Using final with Inheritance

- The keyword final has three uses
- First, it can be used to create the equivalent of a named constant.
- The other two uses of final apply to inheritance.

Using final to prevent Inheritance

- To allow a method from being overriden, specify final as a modifier at the start of its declaration.
- Methods declared as final cannot be overriden.

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

Using final to prevent Inheritance

- Sometimes you will want to prevent a class from being inherited.
- Precede the class declaration with final
- It implicitly declares all of its methods to final too.
- It is illegal to declare a class both as final and abstract.

```
final class A
{
//…………………
}

class B extends A {   //Error
//………………
}
```

Method Overriding

- In a class hierarchy, when a method in a 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 overriden 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.

Example: 1

```
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 useSuper
{
public static void main(String
args[])
{
        B subOb=new B(1,2,3);
        subOb.show();
}
}
```

```
C:\java
useSuper
k: 3
```

Example: 2

```
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()
        {
        super.show();
        System.out.println("k: "+
k);
        }
}
class useSuper
{
public static void main(String
args[])
{
        B subOb=new B(1,2,3);
        subOb.show();
}
}
```

```
C:\java useSuper
i and j: 1 2
k: 3
```

Example: 3(Method overloading)

```
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(String msg)
    {
    System.out.println("k: "+
  k);
    }
}
```

```
class useSuper
{
public static void
main(String args[])
{
B subOb=new
B(1,2,3);
subOb.show("This is
k");
subOb.show();
}
}
```

```
C:\java useSuper
This is k: 3
i and j: 1 2
```

Dynamic Method Dispatch

- It is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- It’s a method to implement run-time polymorphism.
- A superclass reference variable can refer to a subclass object.
- Java uses this fact to resolves calls to overriden methods at run time.

- When an overriden method is called through a superclass reference, Java determines which version of the method to execute based on the type of object being referred to at the time the call occurs.
- It is the type of the object being referred to that determines which version of an overriden method will be executed.

Example Program

```
class A
{
        void callme()
        {
                System.out.println("Inside A's callme method");
        }
}
class B extends A
{
        void callme()
        {
                System.out.println("Inside B's callme method");
        }
}
```

```
class C extends A
{
        void callme()
        {
                System.out.println("Inside C's callme method");
        }
}
class useSuper
{
        public static void main(String args[])
        {
                A a=new A();
                B b=new B();
                C c=new C();
                A r;
                r=a;
                r.callme();
                r=b;
                r.callme();
                r=c;
                r.callme();
        }
}
```

```
Inside A's callme
method
Inside B's callme
method
Inside C's callme
method
```

Why overriden methods

- Overriden methods allow Java to support runtime polymorphism.
- It allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
- Using overriden methods Java implements “The one interface, multiple methods” aspect of polymorphism.

Packages

- Packages are containers for classes that are used to keep the class name space compartmentalised.
- 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.

- A package is both a naming and visibility control mechanism.
- You can define classes inside a package that are not accessible by code outside that package.

Defining a package

- Simply include a package command as the first statement in a java source file.
- Any classes declared within that file belongs to the specified package.
- General form:

```
package pkg;
```

Example: package myPackage;

Example: package MyPackage;

the .class files for any classes you declare to be part of MyPackage must be stored in a directory called MyPackage

- More than one file can include the same package statement.
- You can create a hierarchy of packages.
- Separate each package name from the one above it by use of a period.
- General form:

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

Example:

```
package java.awt.image;
```

Finding packages and CLASSPATH

- Packages are mirrored by directories.
- 1) Java Run-time system uses the current working directory as its starting point. Thus if your package is in the current working directory, or a subdirectory of the current working directory, it will be found.
- 2) You can specify a directory path or paths by setting the CLASSPATH environmental variable.

Example

```
package MyPack;
class useSuper1
{
          static int noOfObjects = 0;
   useSuper1()
   {
          noOfObjects += 1;
   }
}
class useSuper
{
          public static void main(String args[])
   {
      useSuper1 t1 = new useSuper1();
      useSuper1 t2 = new useSuper1();
      useSuper1 t3 = new useSuper1();
      System.out.println(useSuper1.noOfObjects);
   }
}
```

```
C:\Programs>cd MyPack

C:\Programs\MyPack>javac
useSuper.java

C:\Programs\MyPack>cd..

C:\Programs>java
MyPack.useSuper
3
```

Access Protection

- Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods.
- Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:
  - Subclasses in the same package
  - Non-subclasses in the same package
  - Subclasses in different packages
  - Classes that are neither in the same package nor subclasses

Class Member Access

	Private	No Modifier	Protected	Public
Same Class	Yes	Yes	Yes	Yes
Same Package Subclass	No	Yes	Yes	Yes
Same Package Non-Subclass	No	Yes	Yes	Yes
Different Package Subclass	No	No	Yes	Yes
Different Package Non-Subclass	No	No	No	Yes

Protection.java

```
package p1;
public class Protection{
        int n=1;
        private int n_pri=2;
        protected int n_pro=3;
        public int n_pub=4;
        public Protection()
        {
                System.out.println("base constructor");
                System.out.println("n= "+n);
                System.out.println("n_pri= "+n_pri);
                System.out.println("n_pro= "+n_pro);
                System.out.println("n_pub= "+n_pub);
        }

}
```

Derived.java

```
package p1;
class Derived extends Protection
{
    Derived()
    {
        System.out.println("Derived
  constructor");
        System.out.println("n= "+n);
        //System.out.println("n_pri=
  "+n_pri);
        System.out.println("n_pro= "+n_pro);
        System.out.println("n_pub=
  "+n_pub);
    }
}
```

SamePackage.java

```
package p1;
class SamePackage
{
    SamePackage()
    {
    Protection p=new Protection();
    System.out.println("same package
  constructor");
    System.out.println("n= "+p.n);
    //System.out.println("n_pri= "+p.n_pri);
    System.out.println("n_pro= "+p.n_pro);
    System.out.println("n_pub= "+p.n_pub);
    }
}
```

Protection2.java

```
package p2;
class Protection2 extends p1.Protection{
    Protection2()
    {
        System.out.println("Derived other Package Constructor");
        //System.out.println("n= "+n);
        //System.out.println("n_pri= "+n_pri);
        System.out.println("n_pro= "+n_pro);
        System.out.println("n_pub= "+n_pub);
    }
}
```

OtherPackage.java

```
package p2;
class OtherPackage
{
    OtherPackage()
    {
        p1.Protection p=new p1.Protection();
        System.out.println("Other Package
  Constructor");
        //System.out.println("n= "+p.n);
        //System.out.println("n_pri= "+p.n_pri);
        //System.out.println("n_pro= "+p.n_pro);
        System.out.println("n_pub= "+p.n_pub);
    }
}
```

Importing Packages

- All of the built-in classes are stored in packages
- Java includes the import statement to bring certain classes, or entire packages, into visibility.
- Once imported, a class can be referred to directly, using only its name.
- Import statements occur immediately following the package statement and before any class definition.
- General form:

  import pkg1[.pkg2].(classname|*);

- Here, pkg1 is the name of the top-level package, and pkg2 is the name of the subordinate package inside the outer package separated by a dot(.)
- Finally you specify either an explicit classname or a star(*)
- Example:

```
import java.util.*;
import java.io.*;
```

- All the java classes included with java are stored in a package called java
- The basic language functions are stored in a package inside of java package java.lang
- It is implicitly imported by the compiler

- Any place you use a class name, you can use its fully qualified name, which includes its full package hierarchy.
- Example:

```
        import java.util.*;
        class MyDate extends Date
        {
        }
```

OR

```
        class MyDate extends java.util.Date
        {}
```

Example: Figure.java

```
package p1;
public class Figure{
        double dim1;
        double dim2;
        public Figure(double a, double b)
        {
                dim1=a;
                dim2=b;
        }
        public void show(){
                System.out.println(dim1+" "+dim2);
        }
}
```

Test.java

```
import p1.*;
class Test
{
    public static void main(String[] args) {
        Figure tst=new Figure(1,2);
        tst.show();
    }
}
```

Interfaces

- Using the keyword interface, you can fully abstract a class' interface from its implementation.
- That is, using interface, you can specify what a class must do, but not how it does it.
- Intefaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.
- Once it is defined, any number of classes can implement an interface.
- Also one class can implement any number of interfaces.

- To implement an interface, a class must create the complete set of methods defined by the interface.
- Each class is free to determine the details of its own implementation
- By this method, Java allows you to determine the “One interface, Multiple methods” aspect of polymorphism.

Defining Interface

```
access interface name
{
return_type method_name1(parameter_list);
return_type method_name2(parameter_list);
type final-varname1=value;
type final-varname2=value;
//….
}
```

- Here, access is either public or not used
- When, no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared.
- When it is declared as public, it can be used by any other code.
- Example:

```
interface Callback
{
void call(int param);
}
```

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
}
```

- Here, access is either public or not used.
- The methods that implement an interface must be declared public.
- Type signature of the implementing method must match exactly the type signature specified in the interface definition.

```
class Client implements Callback
{
public void call(int p)
{
System.out.println("Callback with" + p);
}
void non_ifaceMeth()
{
System.out.println("Can define other members");
}
}
```

Accessing Implementations through Interface References

- You can declare variables as object references that use an interface rather than a class type.
- Any instance of any class that implements the declared interface can be referred to by such a variable.
- When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to.

Example

```
class TestIFace
{
    public static void main(String args[])
    {
    Callback c=new Client();
    c.call(42);
    }
}
```

Partial Implementations

- If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as abstract.
- Example:

```
abstract class Incomplete implements Callback
{
int a, b;
void show(){
      System.out.println(a+ " " +b);
}
//....
}
```

Variables in Interfaces

```
import java.util.Random;
interface SC{
int No=0;
int Yes=1;
}
class MCQ implements SC{
	Random rand=new Random();
	int ask()
	{
		int prob=(int) (100*rand.nextDouble());
		if(prob<30)
			return No;
		else
			return Yes;
	}
}
```

```
class Test implements SC{
        static void answer(int result)
        {
                switch(result)
                {
                        case No:
                                System.out.println("No");
                        case Yes:
                                System.out.println("Yes");
                }
        }
        public static void main(String[] args) {
                MCQ tst=new MCQ();
                answer(tst.ask());
                answer(tst.ask());
                answer(tst.ask());
                answer(tst.ask());
        }
}
```

```
C:\>java Test
Yes
Yes
No
Yes
No
Yes
```

Interfaces can be extended

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

```
interface A{
void meth1();
void meth2();
}
interface B extends A
{
void meth3();
}
//.......
```

Exception Handling

- An exception is an abnormal condition that arises in a code sequence at run time.
- It is a run time error.
- A java exception is an object that describes an exceptional condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error.
- The method may choose to handle the exception itself, or pass it on.
- At some point, the exception is caught and processed.

- Exceptions
  - Generated by the Java Runtime System
  - Manually generated by your code

Java Exception Handling

- try
  - Program statements that you want to monitor
- catch
  - Your code can catch the exception and handle it
- throw
  - To manually throw an exception
- throws
  - Any exception that is thrown out of a method must be specified as such by a throws clause
- finally
  - Any code that absolutely must be executed is put in finally block

General form of Exception Handling

```
try
{
//block of code to monitor errors
}
catch(ExceptionType1 exOb)
{
//exception handler for ExceptionType1
}
catch(ExceptionType2 exOb)
{
//exception handler for ExceptionType2
}
//…
finally
{
//block of statements to be executed before try block ends
}
```

ExceptionType is the type of exception that has occurred.

Exception Types and classes

- Throwable
  - Exception
    - RuntimeException
  - Error

Exception: This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types.

RuntimeException examples: Division by zero, Invalid array indexing

Error: Exceptions that are not expected to be caught under normal circumstances by your program.
Example: Stack Overflow.

Uncaught Exceptions

```
class exc0{
public static void main(String args[]){
int d=0;
int a=42/d;
}}
```

- Java Runtime System constructs a new exception object and throws this exception.
- This causes the Exc0 to stop, because once the exception is thrown, it must be caught by an exception handler and dealt with it immediately.
- The default handler displays a string describing the exception :

```
        java.lang.ArithmeticException: / by zero at
Exc0.main (Exc0.java: 4)
```

The stack trace will always show the sequence of method invocations that led up to the error

Using try and catch

- If you handle an exception by yourself
  - It allows you to fix an error
  - It prevents the program from automatically terminating
  - Example:

```
class Main
{
        public static void main(String[] args) {
                int d,a;
                try
                {
                        d=0;
                        a=42/d;
                        System.out.println("This will not be printed");
                }
                catch(ArithmeticException e)
                {
                        System.out.println("Division by zero");
                }
                System.out.println("After catch statement");
        }
}
```

```
C:\>java Main
Division by zero
After catch
statement
```

- Once an exception is thrown, program control transfers out of the try block into the catch block.
- Once the catch statement has executed, program control continues with the next line in the program following the entire try/catch mechanism.
- You can display the description of an exception by using the println() statement (Throwable overrides the toString() method)
  - Example:

```
catch(ArithmeticException   e)
{
        System.out.println(" Exception : " + e);
}

Output:
Exception: java.lang.ArithmeticException: / by zero
```

Multiple catch clauses

- More than one exception could be raised by a single piece of code.
- When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed.
- After one catch statement executes, the others are bypassed, and execution continues after the try/catch block.

```
class Main
{
	public static void main(String[] args) {
		try{
			int a=args.length;
			System.out.println("a="+a);
			int b=42/a;
			int c[]={1};
			c[42]=99;
		}
		catch(ArithmeticException e){
			System.out.println("Division by zero "+e);
		}
		catch(ArrayIndexOutOfBoundsException e){
			System.out.println("Array Index oob "+e);
		}
		System.out.println("After multiple catch
   statements");
	}
}
```

```
C:\>java Main
a=0
Division by zero
java.lang.ArithmeticException: /
by zero
After multiple catch statements

C:\>java Main 12
a= 1
Array Index oob
java.lang.ArrayIndexOutOfBoundsE
xception: Index 42 out of bounds
for length 1
After multiple catch statements
```

Nested try statements

- The try statement can be nested.
- A try statement can be inside the block of another try.
- If an inner try statement does not have a catch handler for a particular exception, the next try statement's catch handlers are inspected for a match.
- This continues until one of the catch statement succeeds, or until all of the nested try statements are exhausted.
- If no catch statement matches, then the Javarun-time system will handle the exception.

```
class Main{
    public static void main(String[] args) {
        try{ int a=args.length;
            System.out.println("a= "+a);
            int b=42/a;
            try{ if(a==1)
                    a=a/(a-a);
                if(a==2){
                    int c[]={1};
                    c[42]=99;
            } }
            catch(ArrayIndexOutOfBoundsException e){
                System.out.println("Array Index oob "+e);
        } }
        catch(ArithmeticException e){
            System.out.println("Division by zero  "+e);
        } } }
```

```
C:\>java Main
a= 0
Division by zero
java.lang.ArithmeticException: /
by zero

C:\>java Main 12
a= 1
Division by zero
java.lang.ArithmeticException: /
by zero

C:\>java Main 12 24
a= 2
Array Index oob
java.lang.ArrayIndexOutOfBounds
Exception: Index 42 out of
bounds for length 1
```

throw

- So far, you have only been catching exceptions that are thrown by the Java run-time system.
- It is possible for your program to throw an exception explicitly, using the throw statement.
- General form:

```
throw ThrowableInstance;
```

- ThrowableInstance must be an object of type Throwable or a subclass of Throwable.
- There are two ways you can obtain a Throwable object:
  - Using a parameter into a catch clause
  - Creating an object with new

throw

- The flow of execution stops immediately after the throw statement: any subsequent statements are not executed.
- The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception.
- Outer try statements are inspected in sequence until it finds a match or it goes to the default handler.

Example

```
class Main
{
	static void demoProc()
	{
		try{
			throw new NullPointerException("demo");
		}
		catch(NullPointerException e){
			System.out.println("Caught inside demoProc");
			throw e;
		}
	}
	public static void main(String[] args) {
		try{
			demoProc();
		}
		catch(NullPointerException e){
			System.out.println("Recaught  "+e);
		}
	}
}
```

```
C:\>java Main
Caught inside demoProc
Recaught  java.lang.NullPointerException:
demo
```

throws

- If a method is capable of causing an exception that it does not handle, it must specify this behaviour so that callers of the method can guard themselves against that exception.
- Do this by including a throws clause in the methods declaration.
- A throws clause lists the type of exceptions that a method might throw.
- This is necessary for all exceptions, except those of type Error and RuntimeException, or any of their subclasses.
- All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile time error happens.

throws

- General form:

```
type method_name(parameter-list) throws exception_list
{
// body of method
}
```

Exception list is a comma seperated list of the exceptions that a method can throw.

An incorrect program that tries to throw an exception that it does not catch

```
class Main
{
    static void demoProc()
    {
        System.out.println("Inside Throwone");
        throw new NullPointerException("demo");
    }
    public static void main(String[] args) {
        demoProc();
    }}
```

```
C:\>java Main
Inside Throwone
Exception in thread "main"
java.lang.NullPointerExceptio
n: demo at
Main.demoProc(Main.java:6)
 at Main.main(Main.java:11)
```

Correct Program

```
class Main
{
        static void demoProc() throws NullPointerException
        {
                System.out.println("Inside Throwone");
                throw new NullPointerException("demo");
        }
        public static void main(String[] args) {
                try
                {
                        demoProc();
                }
                catch(NullPointerException e)
                {
                        System.out.println("caught  "+e);
                }
        }
}
```

```
C:\>java Main
Inside Throwone
caught
java.lang.NullPointerException
: demo
```

finally

- finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block.
- The finally block will execute whether or not an exception is thrown.
- If an exception is thrown, the finally block will execute even if no catch statement matches the exception.
- finally clause is executed just before the method returns.

Example

```
class Main
{
    static void procA()
    {
        try
        {
        System.out.println("Inside procA");
        throw new NullPointerException("demo");
        }
        finally
        {
        System.out.println("ProcA's finally");
        }
    }
```

```
static void procB()
{
try
{
    System.out.println("Inside procB");
    return;
}
finally
{
System.out.println("ProcB's finally");
}
}
```

```
static void procC()
    {
        try
        {
        System.out.println("Inside
    procC");
        return;
        }
        finally
        {
        System.out.println("ProcC's
    finally");
        }
    }
```

```
public static void main(String[] args) {
try
{
procA();
}
catch(NullPointerException e)
{
System.out.println("caught  "+e);
}
procB();
procC();
}
}
```

```
C:\>java Main
Inside procA
ProcA's finally
caught  java.lang.NullPointerException: demo
Inside procB
ProcB's finally
Inside procC
ProcC's finally
```

Java’s Built-in Exceptions

- Inside java.lang, java defines several exception classes.
- Exceptions derived from RuntimeException are automatically available. These are called unchecked exceptions because the compiler does not check to see if a method handles or throws these exceptions.

Java’s unchecked RunTimeException Subclasses

ArithmeticException	•Arithmetic Error, such as Divide by Zero
ArrayIndexOutOfBoundsException	•Array Index is out of bounds
ClassCastException	•Invalid cast
NullPointerException	•Invalid use of Null reference
NumberFormatException	•Invalid conversion of a string to a numeric format
StringIndexOutOfBoundsException	•Attempt to index outside the bounds of a string

Java’s checked exceptions

ClassNotFoundException	•Class not found
IllegalAccessException	•Access to a class is denied
InterruptedException	•One thread has been interrupted by another thread
NoSuchMethodException	•A requested method does not exist
....	

Creating your own exception subclasses

- Define a subclass of Exception. It is their existence in the type system that allows you to use them as exceptions.
- The Exception class does not define any methods of its own.
- Exception class is a subclass of Throwable

```
class MyException extends Exception{
        private int detail;
        MyException(int a)
        {
                detail=a;
        }
        public String toString(){
                return "MyException["+detail+"]";
        }
}
class Main
{
        static void compute(int a) throws MyException{
        System.out.println("Called compute("+a+")");
                if(a>10){
                        throw new MyException(a);
                }
                        System.out.println("Normal Exit");
        }
```

```
        public static void main(String[] args)
{
        try
        {
                compute(1);
                compute(20);
        }
        catch(MyException e)
        {
                System.out.println("caught
"+e);
        } } }
```

```
C:\Rincy\Programs>java Main
Called compute(1)
Normal Exit
Called compute(20)
caught  MyException[20]
```

Using class and objects, Write a java program to find the sum of two complex numbers (Hint: Use object as parameter to function).

```
import
  java.util.Scanner;
class Complex
{
float real;
float imag;
Complex(float r,float i)
{
real=r;
imag=i;
}
```

```
void sum(Complex c1,Complex

c2)

{

real=c1.real+c2.real;

imag=c1.imag+c2.imag;

System.out.println("Added

Number is "+real+"+"+imag+"i");

}

}
```

```
class AddComplex
{
public static void main(String args[])
{
Scanner sc=new Scanner(System.in);
System.out.println("Enter Complex Number 1:");
int rl=sc.nextInt();
int im=sc.nextInt();
Complex ob1=new Complex(rl,im);
System.out.println("Enter Complex Number 2:");
rl=sc.nextInt();
im=sc.nextInt();
Complex ob2=new Complex(rl,im);
Complex ob3=new Complex(0,0);
ob3.sum(ob1,ob2);
}
}
```

```
C:\>java AddComplex
Enter Complex Number 1:
2
3
Enter Complex Number 2:
4
5
Added Number is 6.0+8.0i
```

Threads

- Multithreaded program: different parts (threads) run concurrently
- Multithreading is a form of multitasking
- Process based: run two or more programs concurrently
- Thread based: a single program performs two or more tasks simultaneously
- Thread States
  - Running
  - Ready: waiting for CPU
  - Suspended
  - Resumed after suspension
  - Blocked: waiting for resources
  - Terminated: halts execution; cannot resume

Thread Priority

- Context switch: thread priority is used to decide when to switch from one running thread to the next
- Voluntarily relinquish the control: sleeping or blocking on I/O. Highest priority ready thread will get the CPU
- Preemptive multitasking: lower priority thread is preempted by a higher priority thread
- Monitor: used to protect a shared assets from being manipulated by more than one thread at a time

Thread & Runnable

Java's multithreading system is built upon Thread class, its

methods and Runnable interface

- Create a thread: extend Thread or implement Runnable interface Methods
  - getName(): obtain thread's name
  - getPriority(): obtain thread's priority
  - isAlive(): is the thread still running
  - join(): wait for a thread to terminate
  - run(): entry point for a thread
  - sleep(): suspend a thread
  - start(): Start a thread by calling run

The Main Thread

- When a Java program starts up, one thread begins running immediately. (main thread)
- The main thread is important for two reasons
  - It is the thread from which other "child" threads will be spawned
  - It must be the last thread to finish execution because it performs various shutdown activities
  - It can be controlled through a Thread object
  - You can do so by calling a method currentThread()

```
class Main{
public static void main(String args[]){
Thread t=Thread.currentThread();
System.out.println("Current Thread:"+t);
t.setName("My Thread");
System.out.println("After name change"+t);
try{
for(int n=5;n>0;n--){
System.out.println(n);
Thread.sleep(1000);
}
}
catch(InterruptedException e){
System.out.println("Main thread interrupted");
}
}
}
```

```
C:\>java Main
Current
Thread:Thread[main,5,main]
After name
changeThread[My
Thread,5,main]
5
4
3
2
1
```

- static void sleep(long milliseconds) throws InterruptedException
  - Causes the thread from which it is called to suspend execution for a specified period of milliseconds.
- static void sleep(long millisecons, int nanoseconds) throws InterruptedException
- final void setName(Sring threadName)
- final String getName()

Runnable Interface

Construct a thread on any object that implements Runnable

- To implement Runnable, class should contain run()

  public void run()

- Instantiate an object of type Thread within the class

  Thread(Runnable threadOb, String threadName)

- run() establishes the entry point for another, concurrent thread
- thread start running with start() –> start()

  executes a call to run()

```
class ThreadEg implements Runnable {
Thread t;
ThreadEg() {
t = new Thread(this, "Thread Example"); //create second thread
System.out.println("Child thread" + t);
t.start(); }
public void run() {
try {
for(int i=5;i>0;i--) {
System.out.println("Child thread" + i);
Thread.sleep(500); }
} catch(InterruptedException e) {
System.out.println("Child interrupted"); }
System.out.println("Exit child thread"); }
}
```

```
class Main {
public static void main(String args[]) {
new ThreadEg();
try {
for(int i=5;i>0;i--) {
System.out.println("Main thread" + i);
Thread.sleep(1000); }
} catch(InterruptedException e) {
System.out.println("Main thread
interrupted");}
System.out.println("Main thread
exit"); }
}
```

```
C:\>java Main
Child
threadThread[Thread
Example,5,main]
Main thread5
Child thread5
Child thread4
Main thread4
Child thread3
Child thread2
Main thread3
Child thread1
Exit child thread
Main thread2
Main thread1
Main thread exit
```

Extending Thread

- Create a class that extends Thread, and then

  Create an instance of the class
- Override run()
- super() invokes Thread constructor

  public Thread(String threadName)

```
class ThreadEg extends Thread {
ThreadEg() {
super("Thread Example"); //create second thread
System.out.println("Child thread" + this);
start(); }
public void run() { //entry point for second thread
try {
for(int i = 5; i > 0; i--) {
System.out.println("Child thread" + i);
Thread.sleep(500); }
} catch(InterruptedException e) {
System.out.println("Child interrupted"); }
System.out.println("Exit child thread"); }
}
```

```
class Main{
public static void main(String args[]) {
new ThreadEg(); //create a thread
try {
for(int i = 5; i > 0;i--) {
System.out.println("Main thread" + i);
Thread.sleep(1000); }
} catch(InterruptedException e) {
System.out.println("Main thread
interrupted");}
System.out.println("Main thread
exit"); }
}
```

```
C:\Rincy\Programs>java Main
Child threadThread[Thread
Example,5,main]
Main thread5
Child thread5
Child thread4
Main thread4
Child thread3
Child thread2
Main thread3
Child thread1
Exit child thread
Main thread2
Main thread1
Main thread exit
```

Using isAlive() and join()

How can one thread know when other thread has ended?

- Two ways as solution
- First you can call isAlive() on the thread.
- Defined in Thread class
    - final boolean isAlive()

  Returns true if the thread upon which it is called is still running.
- Second the method that is commonly used to wait for a thread to finish is
    - final void join() throws InterruptedException

  This method waits until the thread on which it is called is terminates.

Suspend, Resume, isAlive, join

```
class ThreadEg implements
  Runnable
{
    String name;
    Thread t;
    ThreadEg(String
  threadName)
    {
        name = threadName;
        t = new Thread(this,
  name);
        System.out.println("Ne
  w Thread"+ t);
        t.start();
    }
```

```
public void run()
{
try
{
for(int i=5; i>0; i--)
{
System.out.println(name +
":" + i);
Thread.sleep(200);
}
}
```

```
catch(InterruptedException e)
                {
                System.out.println(name + "interrupted");
                }
                System.out.println(name + “ exiting");
        }
}
class Main
{
        public static void main(String args[])
        {
                ThreadEg ob1 = new ThreadEg("One");
                ThreadEg ob2 = new ThreadEg("Two");
                System.out.println(“Thread one is
                                Alive”+ob1.t.isAlive());
                try
                {
                        Thread.sleep(1000);
                        ob1.t.suspend();
```

```
System.out.println("Suspend
One");
Thread.sleep(1000);
ob1.t.resume();
System.out.println("Resume
One");
ob2.t.suspend();
System.out.println("Suspend
Two");
Thread.sleep(1000);
ob2.t.resume();
System.out.println("Resume
Two");
}
```

```
catch(InterruptedException e)
        {
        System.out.println("Main interrupted");
        }
        try
        {
        System.out.println("Wait for threads to finish");
            ob1.t.join();
            ob2.t.join();
        }
        catch(InterruptedException e)
        {
        System.out.println("Main interrupted");
        }
        System.out.println("Main thread exit");
    }
}
```

```
C:\Rincy\Programs>java Main
New ThreadThread[One,5,main]
New ThreadThread[Two,5,main]
Thread one is Alivetrue
One:5
Two:5
One:4
Two:4
One:3
Two:3
Two:2
One:2
Two:1
One:1
Suspend One
Two exiting
Resume One
Suspend Two
One exiting
Resume Two
Wait for threads to finish
Main thread exit
```

Thread Priorities

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.
- Higher priority threads get more CPU time than lower priority threads.
- A higher priority thread can also preempt a lower priority one.
- To set a thread's priority, use the setPriority() method, which is a member of Thread.
- General form:

```
final void setPriority(int level)
```

- Here, level specifies the new priority setting for the calling thread.
- The value of level must be within the range MIN_PRIORITY and MAX_PRIORITY. These values are 1 and 10, respectively.
- To return a thread to default priority, specify NORM_PRIORITY, which is currently 5.
- These values are defined as final variables within Thread.

```
class Clicker implements Runnable
{
        int click=0;
        Thread t;
        private volatile boolean
    running=true;
        public Clicker(int p)
        {
                t = new Thread(this);
                t.setPriority(p);
        }
        public void run()
        {
                while(running)
                {
                        click++;
                }
        }
```

```
public void stop()
        {
                running=false;
        }
        public void start()
        {
                t.start();
        }
}
```

```
class Main
{
        public static void main(String args[])
        {
                        Thread.currentThread().setPriority(Thread.MAX_PRI
    ORITY);
                        Clicker hi=new Clicker(Thread.NORM_PRIORITY+2);
                        Clicker lo=new Clicker(Thread.NORM_PRIORITY-2);
                        lo.start();
                        hi.start();
                        try
                        {
                                Thread.sleep(1000);
                        }
                        catch(InterruptedException e)
                        {
                                System.out.println("Main thread interrupted");
                        }
                        lo.stop();
                        hi.stop();
                        try
                        {
```

```
                        hi.t.join();
        lo.t.join();
        }
catch(InterruptedException
e)
{
System.out.println("Interrup
tedException caught");
}
System.out.println("Low-pri
ority thread "+lo.click);
System.out.println("High-pri
ority thread "+hi.click);
}
}
```

```
C:\>java Main
Low-priority thread
744800224
High-priority thread
745453003
```

Synchronization

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.
- Monitor/semaphore
- A monitor is an object that is achieved as a mutually exclusive lock
- Only one thread can own a monitor at a time
- When a thread acquires a lock, it is said to have entered the monitor.
- All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are aid to be waiting for the monitor.
- You can synchronize your code through
  - Synchronized methods

```
class Callme
{
        void call(String msg)
        {
                System.out.print("["+msg);
                try
                {
                        Thread.sleep(1000);
                }
                catch(InterruptedException e)
                {
                        System.out.println("Interrupted");
                }
                System.out.println("]");
        }
}
class Caller implements Runnable
{
        String msg;
        Callme target;
        Thread t;
```

```
public Caller(Callme targ, String s)
        {
                target=targ;
                msg=s;
                t=new Thread(this);
                t.start();
        }
        public void run()
        {
                target.call(msg);
        }
}
```

```
class Main
{
    public static void main(String args[])
    {
                Callme target=new Callme();
                Caller ob1=new Caller(target,"Hello");
                Caller ob2=new Caller(target,"Synchronised");
                Caller ob3=new Caller(target,"World");
                try
                {
                    ob1.t.join();
                    ob2.t.join();
                    ob3.t.join();
                }
                catch(InterruptedException e)
                {
                System.out.println("InterruptedException
  caught");
                }
    }
}
```

```
C:\>java Main
[World[Hello[Synchronised]
]
]
```

Make this change

```
public void run()
  {
      synchronized(t
arget){
      target.call(msg
);}
  }
```

```
C:\>java Main
[Hello]
[World]
[Synchronised]
```

Thank You