The page features three large, overlapping blue circles of varying sizes, each with a gradient from light to dark blue. These circles are arranged in a diagonal line from the top right towards the bottom right. Thin, light blue lines intersect at the top left and extend diagonally across the page, framing the central text and circles.

**DRONACHARYA**  
College of Engineering

**LAB MANUAL OF WD**

## Index

1. Software and hardware requirements
2. Study about Java Language
3. List of Programs
4. Programs with output

### **Software and hardware requirements**

Software: jdk1.4

Hardware: PROCESSOR PIV

SPEED 1.8GHz

RAM 256MB

HDD 40GB

## STUDY ABOUT JAVA LANGUAGE

### *Objects*

### **Classes**

Java has *nested* classes that are declared within the body of another class or interface. A class that is not a nested class is called a *top level* class. An inner class is a non-static nested class.

Classes can be declared with the following modifiers:

**abstract** – cannot be instantiated. Only interfaces and abstract classes may contain abstract methods. A concrete (non-abstract) subclass that extends an abstract class must override any inherited abstract methods with non-abstract methods. Cannot be final.

**final** – cannot be subclassed. All methods in a final class are implicitly final. Cannot be abstract.

Note that Java classes do not need to be terminated by a semicolon (";"), which is required in C++ syntax.

---

**Method overloading** is a feature found in various object oriented programming languages such as C++ and Java that allows the creation of several functions with the same name which differ from each other in terms of the type of the input and the type of the output of the function.

An example of this would be a square function which takes a number and returns the square of that number. In this case, it is often necessary to create different functions for integer and floating point numbers.

Method overloading is usually associated with statically-typed programming languages which enforce type checking in function calls. When overloading a method, you are really just making a number of different methods that happen to

have the same name. It is resolved at compile time which of these methods are used.

Method overloading should not be confused with ad-hoc polymorphism or virtual functions. In those, the correct method is chosen at runtime.

\*\*\*\*\*

**Method overriding**, in object oriented programming, is a language feature that allows a subclass to provide a specific implementation of a method that is already provided by one of its superclasses. The implementation in the subclass overrides (replaces) the implementation in the superclass.

A subclass can give its own definition of methods which also happen to have the same signature as the method in its superclass. This means that the subclass's method has the same name and parameter list as the superclass's overridden method. Constraints on the similarity of return type vary from language to language, as some languages support covariance on return types.

Method overriding is an important feature that facilitates polymorphism in the design of object-oriented programs.

Some languages allow the programmer to prevent a method from being overridden, or disallow method overriding in certain core classes. This may or may not involve an inability to subclass from a given class.

In many cases, abstract classes are designed — i.e. classes that exist only in order to have specialized subclasses derived from them. Such abstract classes have methods that do not perform any useful operations and are meant to be overridden by specific implementations in the subclasses. Thus, the abstract superclass defines a common interface which all the subclasses inherit.

### *Examples*

This is an example in Python. First a general class ("Person") is defined. The "self" argument refers to the instance object. The Person object can be in one of three states, and can also "talk".

```
class Person:  
def __init__(self):  
self.state = 0
```

```
def talk(self, sentence):  
    print sentence
```

```
def lie_down(self):  
    self.state = 0
```

```
def sit_still(self):  
    self.state = 1
```

```
def stand(self):  
    self.state = 2
```

Then a "Baby" class is defined (subclassed from Person). Objects of this class cannot talk or change state, so exceptions (error conditions) are raised by all methods except "lie\_down". This is done by overriding the methods "talk", "sit\_still" and "stand"

```
class Baby(Person):  
    def talk(self, sentence):  
        raise CannotSpeakError, 'This person cannot speak.'
```

```
    def sit_still(self):  
        raise CannotSitError, 'This person cannot sit still.'
```

```
    def stand(self):  
        raise CannotStandError, 'This person cannot stand up.'
```

Many more methods could be added to "Person", which can also be subclassed as "MalePerson" and "FemalePerson", for example. Subclasses of Person could then be grouped together in a data structure (a list or array), and the same methods could be called for each of them regardless of the actual class; each object would respond appropriately with its own implementation or, if it does not have one, with the implementation in the superclass.

An **interface** in the Java programming language is an abstract type which is used to specify an interface (in the generic sense of the term) that classes must implement. Interfaces are introduced with the **interface** keyword, and may only contain function signatures and constant declarations (variable declarations which are declared to be both static and final).

---

As interfaces are *abstract*, they cannot be instantiated. Object references in Java may be specified to be of interface type; in which case they must be bound to null, or an object which *implements* the interface.

The primary capability which interfaces have, and classes lack, is multiple inheritance. All classes in Java (other than `java.lang.Object`, the root class of the Java type system) must have exactly one base class (corresponding to the `extends` clause in the class definition; classes without an `extends` clause are defined to inherit from `Object`); multiple inheritance of classes is not allowed. However, Java classes may *implement* as many interfaces as the programmer desires (with the `implements` clause). A Java class which implements an interface, but which fails to implement all the methods specified in the interface, becomes an **abstract base class**, and must be declared `abstract` in the class definition.

## Defining an Interface

Interfaces must be defined using the following formula (compare to Java's class definition).

```
[visibility] interface Interface Name [extends other interfaces] {  
  constant declarations  
  abstract method declarations  
}
```

The body of the interface contains abstract methods, but since all methods in an interface are, by definition, abstract, the `abstract` keyword is not required.

Thus, a simple interface may be

```
public interface Predator {  
  public boolean chasePrey(Prey p);  
  public void eatPrey(Prey p);  
}
```

## Implementing an Interface

The syntax for implementing an interface uses this formula:

... implements *interface name* [, *another interface*, *another*, ...] ...

Classes may implement an interface. For example,

```
public class Cat implements Predator {  
  
    public boolean chasePrey(Prey p) {  
        // programming to chase prey p  
    }  
  
    public void eatPrey (Prey p) {  
        // programming to eat prey p  
    }  
}
```

If a class implements an interface and is not abstract, and does not implement a required interface, this will result in a compiler error. If a class is abstract, one of its subclasses is expected to implement its unimplemented methods.

Classes can implement multiple interfaces

```
public class Frog implements Predator, Prey { ... }
```

\*\*\*\*\*

A **Java package** is a mechanism for organizing Javaclasses into namespaces. Java packages can be stored in compressed files called JAR files, allowing classes to download faster as a group rather than one at a time. Programmers also typically use packages to organize classes belonging to the same category or providing similar functionality.

Java source files can include a **package** statement at the top of the file to designate the package for the classes the source file defines.

1. A package provides a unique namespace for the types it contains.
2. Classes in the same package can access each other's protected members.
3. A package can contain the following kinds of types.

Classes Interfaces Enumerated types Annotations

## ***Using packages***

In Java source files, the package that the file belongs to is specified with the `package` keyword.

```
package java.awt.event;
```

To use a package inside a Java source file, it is convenient to import the classes from the package with an import statement. The statement

```
import java.awt.event.*;
```

imports all classes from the `java.awt.event` package, while

```
import java.awt.event.ActionEvent;
```

imports only the `ActionEvent` class from the package. After either of these import statements, the `ActionEvent` class can be referenced using its simple class name:

```
ActionEvent myEvent = new ActionEvent();
```

Classes can also be used directly without an import statement by using the fully-qualified name of the class. For example,

```
java.awt.event.ActionEvent myEvent = new java.awt.event.ActionEvent();
```

doesn't require a preceding import statement.

## ***Package access protection***

Classes within a package can access classes and members declared with *default access* and class members declared with the *protected* access modifier. Default access is enforced when neither the `public`, `protected` nor `private` access modifier is specified in the declaration. By contrast, classes in other packages cannot access classes and members declared with default access. Class members declared as `protected` can only be accessed from within classes in other packages that are subclasses of the declaring class.

## *Package naming conventions*

Packages are usually defined using a hierarchical naming pattern, with levels in the hierarchy separated by periods (.) (pronounced "dot"). Although packages lower in the naming hierarchy are often referred to a "subpackages" of the corresponding packages higher in the hierarchy, there is no semantic relationship between packages. The Java Language Specification establishes package naming conventions in order to avoid the possibility of two published packages having the same name. The naming conventions describe how to create unique package names, so that packages that are widely distributed will have unique namespaces. This allows packages to be easily and automatically installed and catalogued.

In general, a package name begins with the top level domain name of the organization and then the organization's domain and then any subdomains listed in reverse order. The organization can then choose a specific name for their package. Package names should be all lowercase characters whenever possible.

For example, if an organization in Canada called MySoft creates a package to deal with fractions, naming the package `ca.mysoft.fractions` distinguishes the fractions package from another similar package created by another company. If a US company named MySoft also creates a fractions package, but names it `com.mysoft.fractions`, then the classes in these two packages are defined in a unique and separate namespace.

\*\*\*\*\*

```
// Hello.java

import java.applet.Applet;
import java.awt.Graphics;

public class Hello extends Applet {
    public void paint(Graphics gc) {
        gc.drawString("Hello, world!", 65, 95);
    }
}
```

```
}
```

The **import** statements direct the Java compiler to include the **java.applet.Applet** and **java.awt.Graphics** classes in the compilation. The import statement allows these classes to be referenced in the source code using the *simple class name* (i.e. Applet) instead of the *fully qualified class name* (i.e. java.applet.Applet).

The **Hello** class **extends** (subclasses) the **Applet** class; the Applet class provides the framework for the host application to display and control the lifecycle of the applet. The Applet class is an Abstract Windowing Toolkit (AWT) Component, which provides the applet with the capability to display a graphical user interface (GUI) and respond to user events.

The Hello class overrides the **paint(Graphics)** method inherited from the Containersuperclass to provide the code to display the applet. The paint() method is passed a **Graphics** object that contains the graphic context used to display the applet. The paint() method calls the graphic context **drawString(String, int, int)** method to display the "**Hello, world!**" string at a pixel offset of (**65, 95**) from the upper-left corner in the applet's display.

```
<!-- Hello.html -->
<html>
<head>
<title>Hello World Applet</title>
</head>
<body>
<applet code="Hello" width="200" height="200">
</applet>
</body>
</html>
```

An applet is placed in an HTML document using the **<applet>**HTML element. The applet tag has three attributes set: **code="Hello"** specifies the name of the Applet class and **width="200" height="200"** sets the pixel width and height of the applet. (Applets may also be embedded in HTML using either the object or embed element, although support for these elements by Web browsers is inconsistent.[5][6]) However, the applet tag is deprecated, so the object tag is preferred where supported.

The host application, typically a Web browser, instantiates the **Hello** applet and creates an AppletContext for the applet. Once the applet has initialized itself, it is

added to the AWT display hierarchy. The paint method is called by the AWT event dispatching thread whenever the display needs the applet to draw itself.

1. Drawing Lines
2. Drawing Other Stuff
3. Color- *introduces arrays*
4. Mouse Input- *introduces `showStatus( )` and `Vector`*
5. Keyboard Input
6. Threads and Animation- *introduces `System.out.println( )`*
7. Backbuffers- *introduces `Math.random( )` and `Graphics.drawImage( )`*
8. Painting
9. Clocks
10. Playing with Text- *introduces 2D arrays and hyperlinks*
11. 3D Graphics- *introduces classes*
12. Odds and Ends

\*\*\*\*\*

## **Arrays**

1. Java has array types for each type, including arrays of primitive types, class and interface types, as well as higher-dimensional arrays of array types.
2. All elements of an array must descend from the same type.
3. All array classes descend from the class `java.lang.Object`, and mirror the hierarchy of the types they contain.
4. Array objects have a read-only `length` attribute that contains the number of elements in the array.
5. Arrays are allocated at runtime, so the specified size in an array creation expression may be a variable (rather than a constant expression as in C).

6. Java arrays have a single dimension. Multi-dimensional arrays are supported by the language, but are treated as arrays of arrays.

```
// Declare the array - name is "myArray", element type is references to
"SomeClass"
SomeClass[] myArray = null;
// Allocate the array
myArray = new SomeClass[10];
// Or Combine the declaration and array creation
SomeClass[] myArray = new SomeClass[10];
// Allocate the elements of the array (not needed for simple data types)
for (inti = 0; i<myArray.length; i++)
myArray[i] = new SomeClass();

*****
*
```

## LOOPS

### For loop

```
for (initial-expr; cond-expr; incr-expr) {
statements;
}
```

### For-each loop

J2SE 5.0 added a new feature called the for-each loop, which greatly simplifies the task of iterating through every element in a collection. Without the loop, iterating over a collection would require explicitly declaring an iterator:

```
publicintsumLength(Set<String>stringSet) {
int sum = 0;
Iterator<String>itr = stringSet.iterator();
while (itr.hasNext())
sum += itr.next().length();
return sum;
}
```

The for-each loop greatly simplifies this method:

```
public int sumLength(Set<String> stringSet) {
    int sum = 0;
    for (String s : stringSet)
        sum += s.length();
    return sum;
}
```

\*\*\*\*\*

## Exception handling

**Exception handling** is a programming language construct or computer hardware mechanism designed to handle the occurrence of some condition that changes the normal flow of execution. The condition is called an **exception**. Alternative concepts are signal and event handler.

In general, current state will be saved in a predefined location and execution will switch to a predefined handler. Depending on the situation, the handler may later resume the execution at the original location, using the saved information to restore the original state. For example, an exception which will usually be resumed is a page fault, while a division by zero usually cannot be resolved transparently.

From the processing point of view, hardware interrupts are similar to resumable exceptions, except they are usually not related to the current program flow.

## *Exception support in programming languages*

### Exception handling syntax

Many computer languages, such as Ada, C++, Common Lisp, D, Delphi, Eiffel, Java, Objective-C, OCaml, PHP (as of version 5), Python, REALbasic, ML, Ruby, and all NET languages have built-in support for exceptions and exception handling. In those languages, the advent of an exception (more precisely, an exception handled by the language) unwinds the stack of function calls until an exception handler is found. That is, if function  $f$  contains a handler  $H$  for exception  $E$ , calls function  $g$ , which in turn calls function  $h$ , and an exception  $E$  occurs in  $h$ , then functions  $h$  and  $g$  will be terminated and  $H$  in  $f$  will handle  $E$ .

Excluding minor syntactic differences, there are only a couple of exception handling styles in use. In the most popular style, an exception is initiated by a special statement (throw, or raise) with an exception object. The scope for exception handlers starts with a marker clause (try, or the language's block starter such as begin) and ends in the start of the first handler clause (catch, except, rescue). Several handler clauses can follow, and each can specify which exception classes it handles and what name it uses for the exception object.

A few languages also permit a clause (else) that is used in case no exception occurred before the end of the handler's scope was reached. More common is a related clause (finally, or ensure) that is executed whether an exception occurred or not, typically to release resources acquired within the body of the exception-handling block. Notably, C++ lacks this clause, and the Resource Acquisition Is Initialization technique is used to free such resources instead.

In its whole, exception handling code might look like this (in pseudocode):

```
try {
line = console.readLine();
if (line.length() == 0) {
throw new EmptyLineException("The line read from console was empty!");
}
console.println("Hello %s!" % line);
} catch (EmptyLineException e) {
console.println("Hello!");
} catch (Exception e) {
console.println("Error: " + e.message());
} else {
console.println("The program ran successfully");
} finally {
console.println("The program terminates now");
}
```

As a minor variation, some languages use a single handler clause, which deals with the class of the exception internally.

### Checked exceptions

The designers of Java devised<sup>[1][2]</sup> *checked exceptions*<sup>[3]</sup> which are a special set of exceptions. The checked exceptions that a method may raise constitute part of the type of the method. For instance, if a method might throw an IOException instead

of returning successfully, it must declare this fact in its method header. Failure to do so raises a compile-time error.

This is related to exception checkers that exist at least for OCaml. The external tool for OCaml is both transparent (i.e. it does not require any syntactic annotations) and facultative (i.e. it is possible to compile and run a program without having checked the exceptions, although this is not suggested for production code).

The CLU programming language had a feature with the interface closer to what Java has introduced later. A function could raise only exceptions listed in its type, but any leaking exceptions from called functions would automatically be turned into the sole runtime exception, `failure`, instead of resulting in compile-time error. Later, Modula-3 had a similar feature.<sup>[4]</sup> These features don't include the compile time checking which is central in the concept of checked exceptions and hasn't as of 2006 been incorporated into other major programming languages than Java.<sup>[5]</sup>

## Pros and cons

Checked exceptions can, at compile time, greatly reduce (but not entirely eliminate) the incidence of unhandled exceptions surfacing at runtime in a given application; the unchecked exceptions (`RuntimeExceptions` and `Errors`) can still go unhandled.

However, some see checked exceptions as a nuisance, syntactic salt that either requires large throws declarations, often revealing implementation details and reducing encapsulation, or encourages the (ab)use of poorly-considered try/catchblocks that can potentially hide legitimate exceptions from their appropriate handlers.

Others do not consider this a nuisance as you can reduce the number of declared exceptions by either declaring a superclass of all potentially thrown exceptions or by defining and declaring exception types that are suitable for the level of abstraction of the called method, and mapping lower level exceptions to these types, preferably wrapped using the exception chaining in order to preserve the root cause.

A simple throws Exception declaration or catch (Exception e) is always sufficient to satisfy the checking. While this technique is sometimes useful, it effectively circumvents the checked exception mechanism, so it should only be used after careful consideration.

One prevalent view is that unchecked exception types should not be handled, except maybe at the outermost levels of scope, as they often represent scenarios that do not allow for recovery: RuntimeExceptions frequently reflect programming defects<sup>[7]</sup>, and Errors generally represent unrecoverable JVM failures. The view is that, even in a language that supports checked exceptions, there are cases where the use of checked exceptions is not appropriate.

## Questions and Exercises: Object-Oriented Programming Concepts

### Questions

- Real-world objects contain \_\_\_ and \_\_\_.
- A software object's state is stored in \_\_\_.
- A software object's behavior is exposed through \_\_\_.
- Hiding internal data from the outside world, and accessing it only through publicly exposed methods is known as data \_\_\_.
- A blueprint for a software object is called a \_\_\_.
- Common behavior can be defined in a \_\_\_ and inherited into a \_\_\_ using the \_\_\_ keyword.
- A collection of methods with no implementation is called an \_\_\_.
- A namespace that organizes classes and interfaces by functionality is called a \_\_\_.
- The term API stands for \_\_\_?

### LIST OF PROGRAMS

1. Program to calculate area using class and object.
2. Program to take input from command line
3. Program to take the input data from user using class BufferedReader.
4. Program in JAVA using constructor overloading to calculate volume.
5. Program in JAVA to calculate volume using single inheritance.
6. Program in JAVA to implement multiple inheritance using Interface
7. Program to create and import a package to calculate marks and print the grade of student.
8. Program to set the priority of a thread in Multithreading.
9. Program for handling uncaught exception using finally.
10. Program to show a face on an Applet

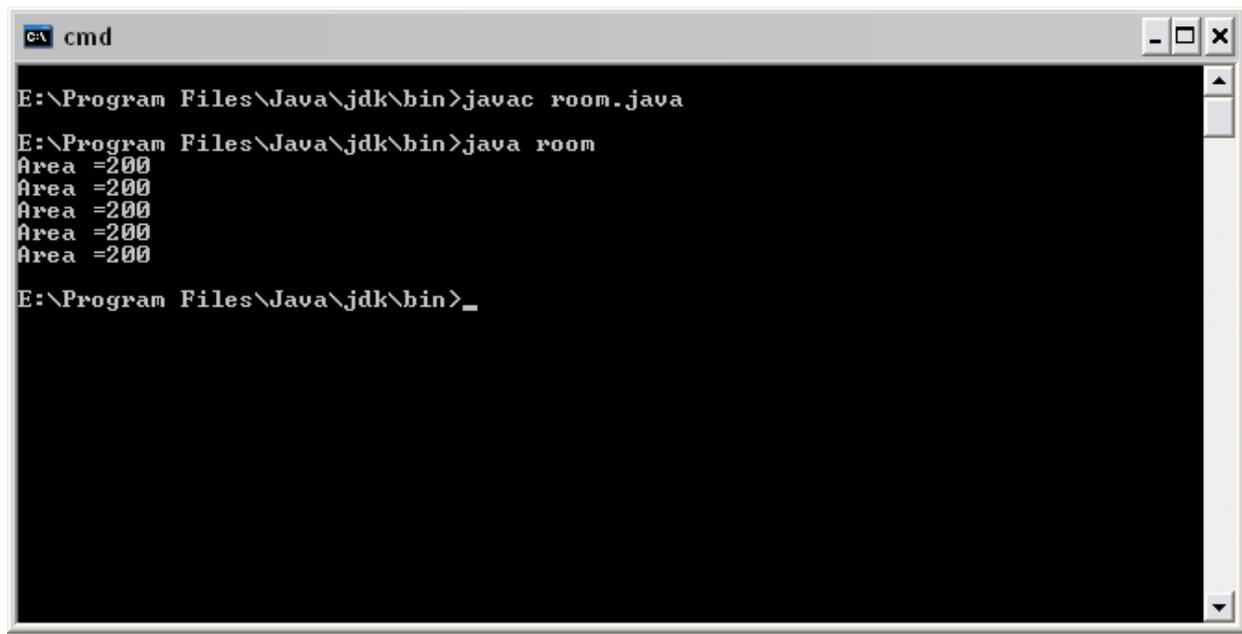
### PROGRAM NO - 1

- Write a program to calculate area using class and object.

```
classroomarea
{
intlength,breadth,area;
voidgetdata(intl,int b)
{
length=l;
breadth=b;
}
intcalarea()
{
area=length*breadth;
System.out.println("Area =" +area);
return area;
}
}
```

```
class room
{
public static void main(String args[])
{
roomarea r=new roomarea();
r.getdata(10,20);
r.calarea();
System.out.println("Area =" +r.calarea());
intaa=r.calarea();
System.out.println("Area =" +aa);
}
}
```

## OUTPUT



```
c:\ cmd
E:\Program Files\Java\jdk\bin>javac room.java
E:\Program Files\Java\jdk\bin>java room
Area =200
Area =200
Area =200
Area =200
Area =200
E:\Program Files\Java\jdk\bin>_
```

- Write a program to take input from command line.

```
classcommandline
{
public static void main(String args[])
{
intcount,i=0;
String st;
count=args.length;
System.out.println("No of arguments:"+count);
while(i<count)
{
st=args[i];
i=i+1;
System.out.println(st);
}
}
}
```

OUTPUT

```
cmd
Microsoft Windows [Version 5.2.3790]
(C) Copyright 1985-2003 Microsoft Corp.

E:\Program Files\Java\jdk\bin>javac commandline.java

E:\Program Files\Java\jdk\bin>java commandline i am inderjeet singh bahl
No of arguments is:5
i
am
inderjeet
singh
bahl

E:\Program Files\Java\jdk\bin>_
```

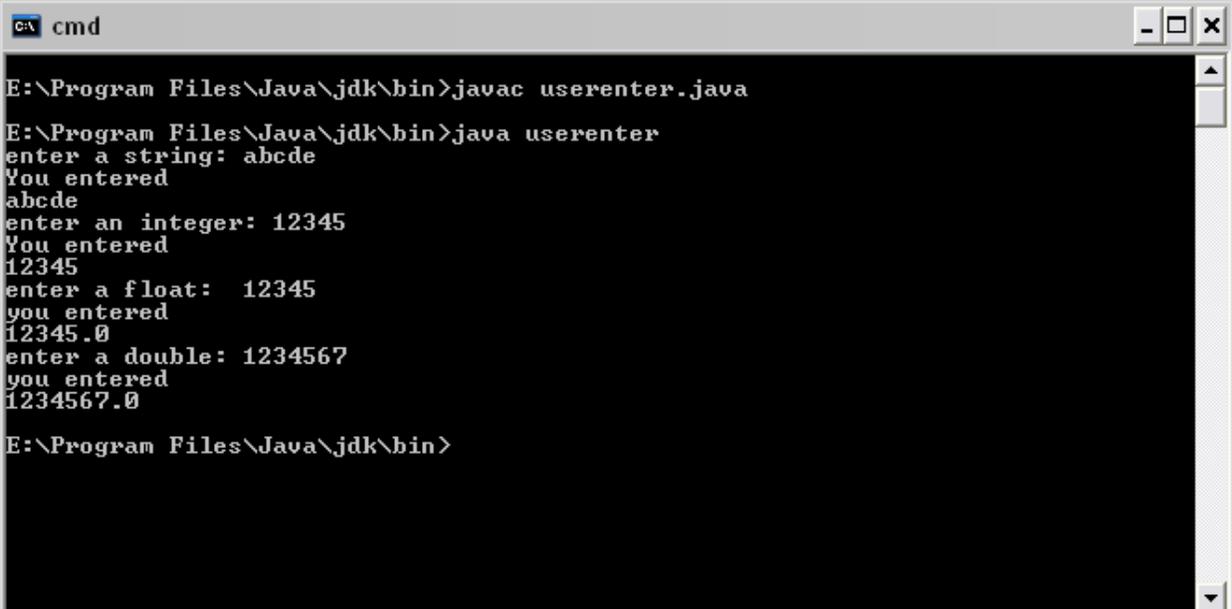
PROGRAM NO - 3

- Write a program to take the input data from user using class bufferedReader.

```
import java.io.*;
class userenter
{
public static void main(String args[])throws IOException
{
BufferedReader br=new BufferedReader(new

InputStreamReader(System.in));
System.out.print("enter a string:\t");
String str=br.readLine();
System.out.println("You entered");
System.out.println(str);
System.out.print("enter an integer: ");
inti=Integer.parseInt(br.readLine());
System.out.println("You entered");
System.out.println(i);
System.out.print("enter a float:\t");
float f=Float.parseFloat(br.readLine());
System.out.println("you entered");
System.out.println(f);
System.out.print("enter a double:\t");
double d=Double.parseDouble(br.readLine());
System.out.println("you entered");
System.out.println(d);
}
}
```

## OUTPUT



```
cmd
E:\Program Files\Java\jdk\bin>javac userenter.java
E:\Program Files\Java\jdk\bin>java userenter
enter a string: abcde
You entered
abcde
enter an integer: 12345
You entered
12345
enter a float: 12345
you entered
12345.0
enter a double: 1234567
you entered
1234567.0
E:\Program Files\Java\jdk\bin>
```

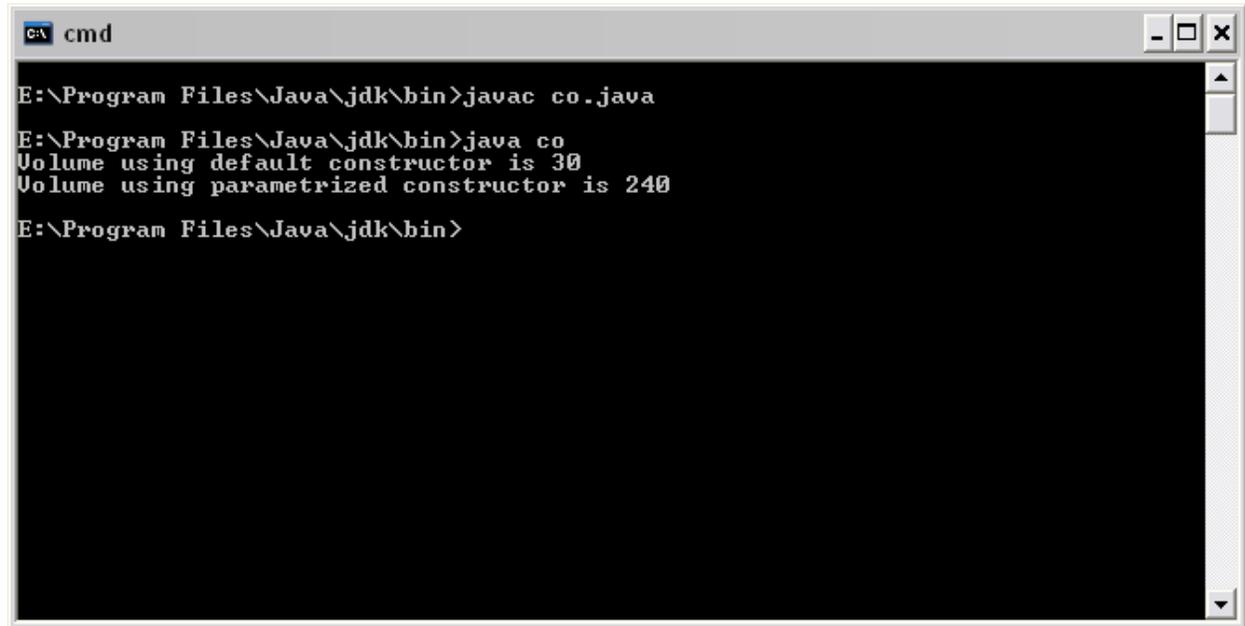
## PROGRAM NO - 4

- Write a program in JAVA using constructor overloading to calculate volume.

```
class const1
{
int l,b,h;
const1()
{
l=2;
b=3;
h=5;
}
const1(int l1,int b1,int h1)
{
l=l1;
b=b1;
h=h1;
}
}
class co
{
public static void main(String args[])
{
const1 v=new const1();
const1 v1=new const1(3,8,10);
int volume=v.l*v.b*v.h;
int volume1=v1.l*v1.b*v1.h;
System.out.println("Volume using default constructor is "+volume);
System.out.println("Volume using parametrized constructor is
"+volume1);
```

```
    }  
}
```

## OUTPUT



```
c:\ cmd  
E:\Program Files\Java\jdk\bin>javac co.java  
E:\Program Files\Java\jdk\bin>java co  
Volume using default constructor is 30  
Volume using parametrized constructor is 240  
E:\Program Files\Java\jdk\bin>
```

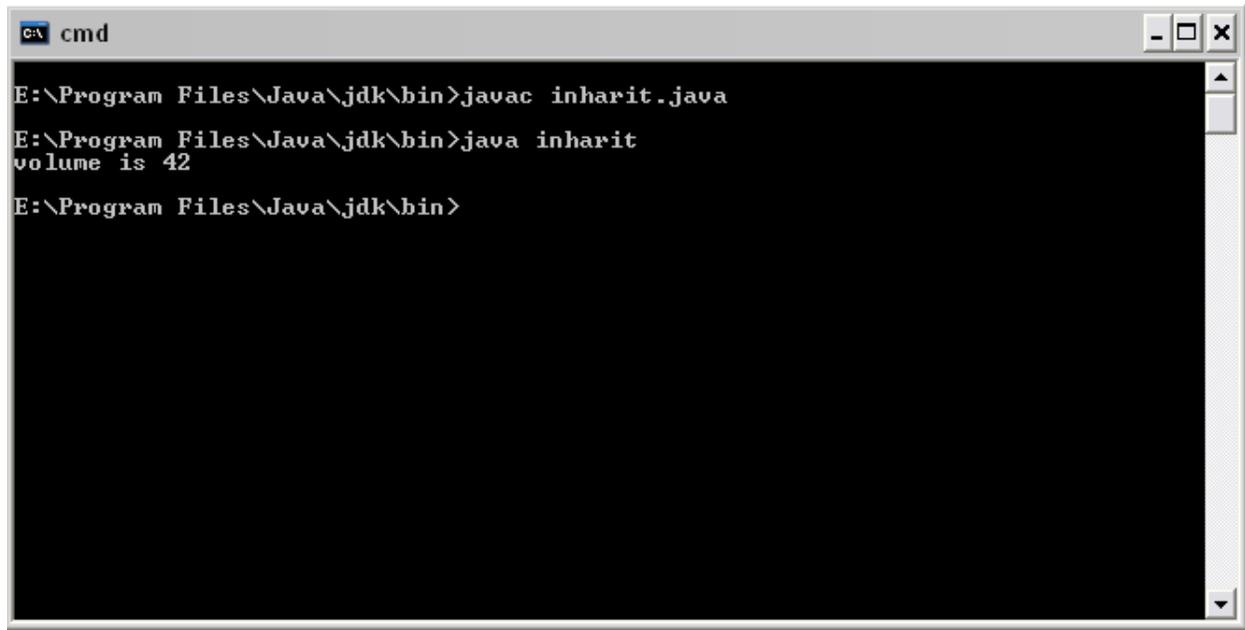
- Write a program in JAVA to calculate volume using single inheritance.

```
class abc
{
    int l,b,h;
    abc()
    {
        l=2;
        b=3;
        h=7;
    }
}

class xyz extends abc
{
}

class inherit
{
    public static void main(String args[])
    {
        xyz v=new xyz();
        int volume=v.b*v.l*v.h;
        System.out.println("volume is "+volume);
    }
}
```

OUTPUT



```
cmd
E:\Program Files\Java\jdk\bin>javac inharit.java
E:\Program Files\Java\jdk\bin>java inharit
volume is 42
E:\Program Files\Java\jdk\bin>
```

### PROGRAM NO - 6

- Write a program in JAVA to implement multiple inheritance using Interface

```
class student{
```

```
introllno;
voidgetno(int r){
rollno=r;}
voidputno(){
System.out.println("Roll no="+ rollno);
}}
class marks extends student
{
int sub1,sub2;
voidgetmarks(int s1,int s2)
{
sub1=s1;
sub2=s2;
}
voidputmarks(){
System.out.println("Subject 1 marks="+sub1);
System.out.println("Subject 2 marks="+sub2);
}}
interface weight{
intwt=60;
voidputwt();
}
class result extends marks implements weight
```

```
{
int total;

public void putwt()
{
System.out.println("Weight="+wt);}

void display(){
total=sub1+sub2+wt;

putno();

putmarks();

putwt();

System.out.println("Total="+total);}
}

class final1
{
public static void main(String args[]){

result r1=new result();

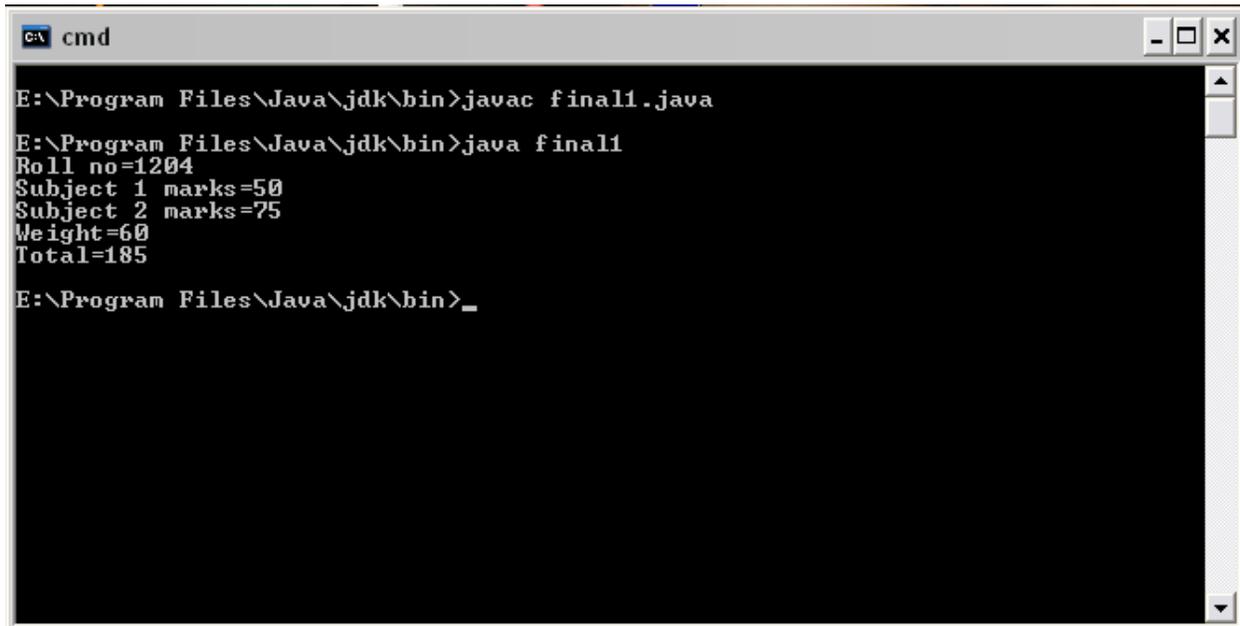
r1.getno(1204);

r1.getmarks(50,75);

r1.display();

}}
```

OUTPUT



```
cmd
E:\Program Files\Java\jdk\bin>javac final1.java
E:\Program Files\Java\jdk\bin>java final1
Roll no=1204
Subject 1 marks=50
Subject 2 marks=75
Weight=60
Total=185
E:\Program Files\Java\jdk\bin>_
```

PROGRAM NO - 7

- Write a program to create and import a package to calculate marks and print the grade of student.

```
package pack;

public class student
{
int sub1,sub2;

public void getmarks(intx,int y)
{
sub1=x;
sub2=y;
}

public void putmarks()
{
int tot=sub1+sub2;
System.out.println("total="+tot);
}
}
```

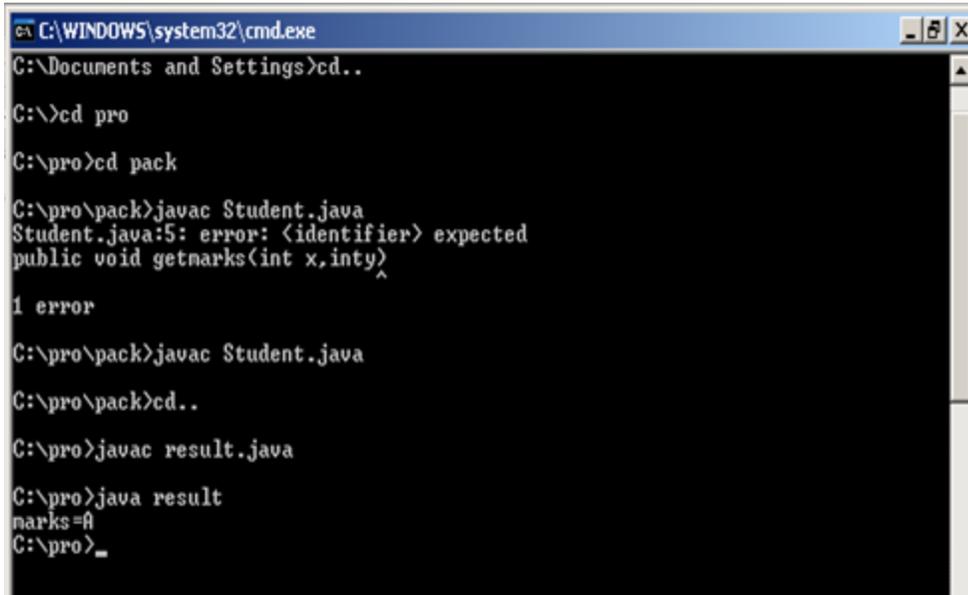
---

```
import pack.*;

class grade extends student
{
char gr;
```

```
grade(char p)
{
gr=p;
}
voidgetresult()
{
putmarks();
System.out.println("grade="+gr);
}
}
class result
{
public static void main(String args[])
{
grade r=new grade('A');
r.getmarks(60,80);
r.getresult();
}
}
```

## OUTPUT



```
C:\WINDOWS\system32\cmd.exe
C:\Documents and Settings>cd..
C:\>cd pro
C:\pro>cd pack
C:\pro\pack>javac Student.java
Student.java:5: error: <identifier> expected
public void getmarks(int x,inty)
                             ^
1 error
C:\pro\pack>javac Student.java
C:\pro\pack>cd..
C:\pro>javac result.java
C:\pro>java result
marks = 0
C:\pro>_
```

## PROGRAM NO - 8

- Write a program to set the priority of a thread in Multithreading.

```
class A extends Thread
{
public void run()
{
System.out.println("Thread A started");
for(int i=1;i<=3;i++)
{
System.out.println("Thread A"+i);
}
System.out.println("Exit Thread A");
}
}

class B extends Thread
{
public void run()
{
System.out.println("Thread B started");
for(int j=1;j<=3;j++)
{
System.out.println("Thread B"+j);
}
System.out.println("Exit Thread B");
}
}

class C extends Thread
{
public void run()
{
System.out.println("Thread C started");
for(int k=1;k<=3;k++)
{
System.out.println("Thread C"+k);
}
System.out.println("Exit Thread C");
}
}
```

```
}  
classtheadtest  
{  
public static void main(String args[])  
{  
A tha=new A();  
B thb=new B();  
C thc=new C();  
thc.setPriority(Thread.MAX_PRIORITY);  
thb.setPriority(tha.getPriority()+1);  
tha.setPriority(Thread.MIN_PRIORITY);  
tha.start();  
thb.start();  
thc.start();  
}  
}
```

OUTPUT

```
cmd
E:\Program Files\Java\jdk\bin>javac threadtest.java
E:\Program Files\Java\jdk\bin>java threadtest
Thread A started
Thread B started
Thread C started
Thread C1
Thread C2
Thread C3
Exit Thread C
Thread B1
Thread B2
Thread B3
Exit Thread B
Thread A1
Thread A2
Thread A3
Exit Thread A
E:\Program Files\Java\jdk\bin>_
```

## PROGRAM NO - 9

- Write a program for handling uncaught exception using finally.

```
import java.io.*;

classerrhandler

{

public static void main(String args[])

{

try

{

DataInputStreambr=new DataInputStream(System.in);

System.out.println("Enter the String: ");

String s=br.readLine();

System.out.println("Enter a Number: ");

inti=Integer.parseInt(br.readLine());

System.out.println("String: "+s);

System.out.println("Number: "+i);

int a=10;

int b=15;

int c=5;

int d=a/(b-c);

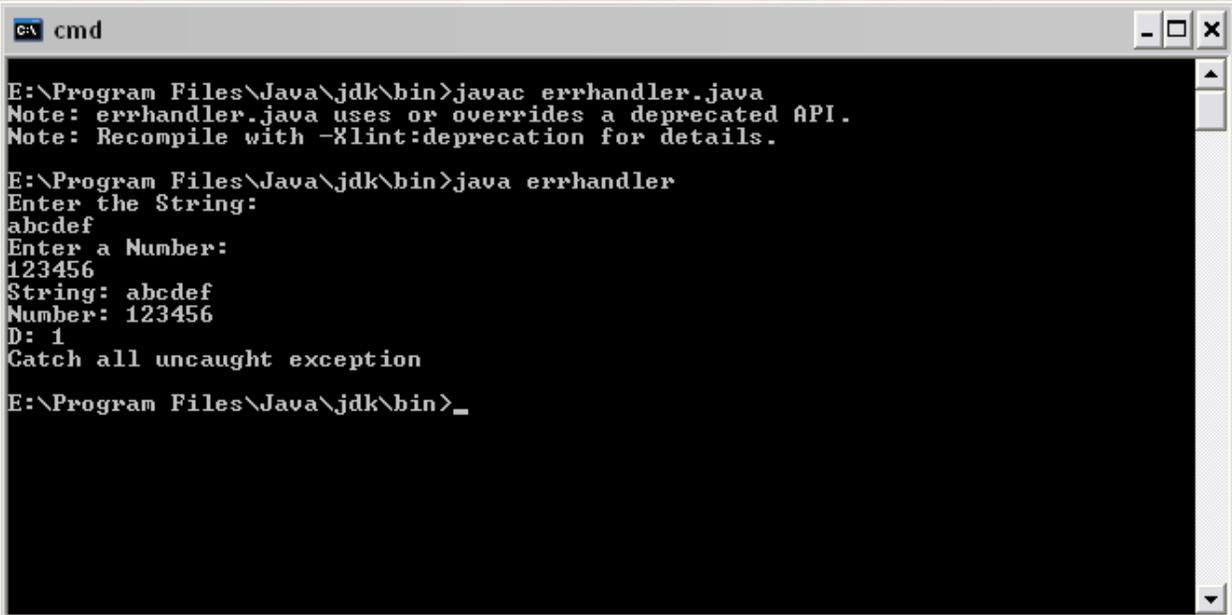
System.out.println("D: "+d);

}

catch(ArithmeticException e)
```

```
{  
System.out.println("Arithmetic Exception Catch");  
}  
catch(IOException e)  
{  
System.out.println("IOException Catch");  
}  
finally  
{  
System.out.println("Catch all uncaught exception");  
}  
}  
}
```

## OUTPUT



```
cmd
E:\Program Files\Java\jdk\bin>javac errhandler.java
Note: errhandler.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

E:\Program Files\Java\jdk\bin>java errhandler
Enter the String:
abcdef
Enter a Number:
123456
String: abcdef
Number: 123456
D: 1
Catch all uncaught exception

E:\Program Files\Java\jdk\bin>_
```

## PROGRAM NO - 10

- Write a program to show a face on an Applet.

```
/* <applet
code=face.class
width=300
height=300
>
</applet>*/
import java.awt.*;
import java.applet.*;
public class face extends Applet
{
public void paint(Graphics g)
{
g.drawOval(40,40,120,150);
g.setColor(Color.blue);
g.drawOval(57,75,30,20);
g.drawOval(110,75,30,20);
g.fillOval(68,81,10,10);
```

```
g.fillOval(121,81,10,10);  
g.drawOval(85,100,30,30);  
g.setColor(Color.red);  
g.fillArc(60,125,80,40,180,180);  
g.drawOval(25,92,15,30);  
g.drawOval(160,92,15,30);  
}  
}
```

## OUTPUT

