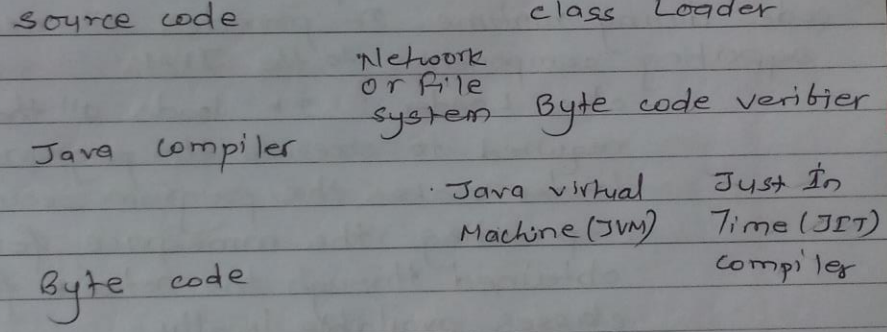
Java Architecture

Date: Page:

Java combines both the approaches of compilation and interpretation. At the first step, the Java compiler compiles the source code into bytecode, At the run time, Java Virtual Machine (JVM) interprets this bytecode and generates machine code which will be directly executed by the machine in which java program runs. Hence java is both compiled and interpreted language.



Java Virtual Machine (JVM): It is a component which provides an environment for running Java programs. JVM interprets the bytecode into machine code which will be executed in the machine in which the Java program runs. The JVM is unique for each platform.

Date: Page:

Bytecode : Platform independence is one of the main advantage of Java. It is the intermediate code that is produced by the Java compiler.

The bytecode is stored in class files. The bytecode is then interpreted by unique JVM to make it run in the desired devices.

Java Runtime Environment (JRE) : JRE contains JVM, class libraries and other supporting components. The bytecode that is stored in the class files will be loaded, verified and interpreated into the machine code during runtime. It provides all necessary supporting components to the JVM.

- class Loader : It loads all the class files required to execute the program. class loader makes the program secure by separating the namespace for the classes obtained through the network from the classes available locally.

- Bytecode verifier : once the bytecode is loaded successfully, then next step is bytecode verification by the bytecode verifier. once the code is verified and proven that there is no security issues with the code, the JVM converts the bytecode into machine code.

GURUKUL

Just In
which
faster.
It
particula
compiles
directly
program
by the
will

Adv

(i) Write
importa
write
be

(ii) Secu
over
Java

(iii) Netw
easy
to

Date: Page:

Just In Time (JIT) Compiler: This is a component which helps the program execution to happen faster.

If the JIT compiler library exists, when a particular bytecode is executed first time, JIT compiler compiles it into native machine code which can be directly executed by the machine in which the Java program runs. once the bytecode is recompiled by the JIT compiler, the execution time needed will be much lesser.

Advantages of Java :

(i) Write Once, Run Anywhere (WORA) : The most important advantage of Java is that you have to write the application code only once and you will be able to run anywhere.

(ii) Security : users can download untrusted code over a network and run it in a secure environment. Java also has highly configurable restrictions levels

(iii) Network-centric Programming: Java makes it easy to work with resources across a network and to create network based apps using multitier architecture

(iv) Dynamic, Extensible Programs: Java code is organized in modular object-oriented units called classes, classes are stored in separate files and are loaded into the Java interpreter only when needed.

This means that an application can decide what it needs and dynamically extend itself by loading the classes it needs to expand its functionality.

(v) platform Independence: Java source code can be run on any device regardless of its origin.

(vi) Performance: Java is portable, interpreted language; Java programs run almost as fast as native, non-portable C & C++ programs.

(vii) Reusability of code: since, Java is object oriented, we can create modular programs and the reusability of the code is also high.

(viii) Java is Robust (Reliable): Java puts lot of emphasis on early checking for possible errors. Java compilers are able to detect many problems that would first show up during execution time in other languages.

(ix) Java is
ading capab
perform se
program.

PATH & C

PATH: on
it is requir
variable t
Javac.exe
any direct
path of

The pat
operatin
executabl
terminal

However, y
any issu
variable
PATH

CLASSPA
used by
compiler

GURUKUL

(ix) Java is Multithreaded: Java has the multithreading capability that allows a program to perform several tasks simultaneously within a program.

PATH & CLASSPATH Variables:

PATH: once Java is installed on a machine, it is required to set the PAHT environment variable to conviniently run the executables (Javac.exe, javadoc.exe, java.exe and so on) from any directory without having to type the full path of the command.

The path is the system variable that our operating system uses to locate needed executables from the command line or from terminal window.

However, you can also run Java apps with no any issues without setting the PATH environment variable but it is convenient & easy once PATH is set, to use Java.

CLASSPATH: It is a system environment variable used by the Java compiler and JVM. Java compiler and JVM uses CLASSPATH to determine

the location of required class files.

The 'CLASSPATH' variable is one way to tell applications, including the JDK tools, where to look for user classes.

IF 'classpath' is not set the user will get a "CLASSPATH: undefined variable" error or simply %CLASSPATH%.

The default value of class path is "." means only the current directory the -cp command line switch overrides this value.

Date: Page:

1.2: classes and objects

creating class:

A class is declared by use of the 'class' keyword. once a class is defined, you declare it's exact form and nature.

A class defines a new data type, once defined, this new type can be used to create objects of that type, Thus, a class is a template for an object, an object is an instance of a class.

Any concept you wish to implement in Java program must be encapsulated within a class.

only

General form of class:

```
class classname{
    type-instance-variable1;
    type-instance-variable2;
    //....
    type instance-variable N;
    type methodname 1(parameter-list){
      // body of method 1 }

    type methodname 2(parameter-list){
      // body of method 2 }
    //.....

    type methodnameN(Parameter-list){
      // body of method N }
}
```

GURUKUL

The data or variables, defined within a class are called instance variables. The code is contained within methods. collectively, the methods and variables defined within a class are called members of the class.

Example of class:

```
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 Demo{
    ~~Pub~~ Public static void main(String args[]){
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;
```

GURUKUL

```
myboxs.setDim(10,20,15);
mybox2.setDim(3,6,9);

vol = mybox1.volume();
System.out.println("volume is" +vol);

vol = mybox2.volume();
System.out.println("volume is" +vol);
}
}
```

creating objects:

once we create a class, we create a new data type we can use this type to declare objects of that type. creating an object is a two step process. Firstly, we must declare a variable of the class type. This variable doesnot define an object. It is a simple variable that can refer to an object

Secondly, we must acquire an actual, physical copy of the object and assign it to that variable. ye can do this by using the new operator. The new operator dynamically allocates memory for an object and returns a reference to it.

The general form is;

```
class variable = new class();
```

```
mybox1.setDim(10,20,15);
mybox2.setDim(3,6,9);

vol = mybox1.volume();
System.out.println("volume is" +vol);

vol = mybox2.volume();
System.out.println("volume is" +vol);
    }
}
```

creating objects:

once we create a class, we create a new data type. we can use this type to declare objects of that type. creating an object is a two step process. Fitstly, we must declare a variable of the class type. This variable dosnot define an object. It is a simple variable that can refer to an object.

Secondly, we must acquire an actual, physical copy of the object and assign it to that variable. ye can do this by using the new operator.

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

The general form is;

```
class variable = new class();
```

Eg: Box mybox = new Box();
The above statement combines the two steps just described. It can be rewritten like this as,

```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

In creating Interfaces

Using the interface, we can fully abstract a class interface from its implementation i.e by using interface, we can specify what a class must do, but not how it does it.

Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. once an Interface is defined any number of classes can implement an Interface.

To implement an interface a class must create the complete set of methods defined by the interface. By providing the 'interface' keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.

Interfaces are designed to support dynamic method resolution at run time.

GURUKUL

Defining

class. This

access inte
return
return

}

Here, acce
no access
of same pa
by any o
is the no
identifier
variables c
declarations
value.

Implemen

interface

GURUKUL

Date: Page:

Defining an Interface:

An interface is defined much like a class. This is the general form of an interface

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

Here, access is either public or not used. When no access specifier is included, then only the members of same package can access it. else it can be used by any other member if public is used. name is the name of the interface and can be any valid identifier.

variables can be declared inside of interface declarations and need to be initialized with a constant value.

Implementing Interface.

```
interface callback{
        void callback(int param);
```

```
class client implements callback {

    Public void callback(int p) {
    System.out.println("callback called with" + p);
    }
}

class Anotherclient implements callback {

    Public void callback(int p) {
    System.out.println("Another version of callback");
    System.out.println("p squared is" +(p*p));
    }
}


class TestInt {
    public static void main(String args[]) {
      client c1 = new client();
      Anotherclient c2 = new Anotherclient();
      c1.callback(42);
      c2.callback(42);
    }
}
```

output:

callback called with 42
Another version of callback
p squared is 1764.

Acce

The three
and Protected
the many le

* Anything
from

* Anything
outside

* Anything
current
your

Same class
Same package subc
same " non-sub
Different package su
" " non-s

Date: Page:

Access Modifiers

The three access modifiers, Private, Public and Protected provide a variety of ways to produce the many levels of access required by these categories.

* Anything declared 'public' can be accessed from anywhere.

* Anything declared private cannot be seen outside of it's class

* Anything declared protected can be seen outside current package but only to classes that subclass your class directly.

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

Fig: class Member Access

GURUKUL

Date: Page:

Arrays:

In Java arrays are implemented as objects.

* one dimensional array can be declared as follows;

```
int a1[] = new int[10];
int a2[] = {3,5,7,9,11,-10};
int a3[] = {4,3,2};
```

* Multidimensional arrays can be declared as;

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

above declaration generates two dimensional array twoD of size 4 by 5.

Example:

```
class Array {
    Public static void main (String args[]){
    int num[] = {1,2,3,4,5};
    for (i=0; i<5; i++)
        system.out.println("Number:" + num[i]);
    }
}
```

output:

```
Number: 1
Number: 2
   :
Number: 5
```

GURUKUL

elas
f
}
output.
0
5
10
15
GURUKUL

```
class twodarray {
    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
```

Packages:

Packages ar
are used to keep th
talized.
For ex: when you cre
a package, the pack
collide with other c
The package
control mechanism or
inside a package th
any code outside tha
we can also define
exposed only to other

Defining a Package:

The general form of the

Package name;
ex: package Packg1; whe

Java uses file system
for exaple; any
are stored in direct

You can also cre
can simply separa
period. The ge
Package PK
GURUKUL Pkg

class superf
int i,j;
void showij()f
system.out.println("i&j" +

}

class Sub extends superf
int k;
void showk()f
system.out.println("k:"+k);
}

void sum()f
System.out.println("Sum:

}

}

Simplefnh ()f
public static void main(

Super superob = new
Sub subob = new

Date: Page:

Packages:

Packages are containers for classes. They are used to keep the class name space compartmentalized.

Per ex: when you create a class named 'List' in a package, the package ensures that it will not collide with other classes having same name 'List'.

The package is both a naming and visibility control mechanism. one can define classes inside a package that are not accersible by any code outside that package.

we can also define class members that are exposed only to other members of the same package.

Defining a Package:

The general form of the package statement is

Package name;

ex: package Packg1; where packg1 is name of package.

Java uses file system directories to store packages for example; any classes declared within packg1 are stored in directory called packg1.

You can also create a hierarchy of packages, we can simply separate each package name by using period. The general form is;

Package ~~Pkg1[.Pkg2.[~~

Pkg1[.Pkg2[.Pkg3]];

A package
file system
for eg. a

Packa
stored
environ

Inheri

oriented pr
hierarchial

By
class that
items. This
other, more s
are unique
The class
superclass
A subclass
It inherits
and adds

To inh
defini
exten

A package hierarchy must be reflected in the file system of Java development system.
for eg. a package declared as

Package java.awt.image; needs to be stored in ~~java/~~ java/awt/image in a windows environment.

Inheritance:

It is one of the cornerstones of object oriented programming because it allows the creation of hierarchial classifications.

By using inheritence we can create a general class that defines attributes common to a set of items. This ~~das~~ class can then be inherited by other, more specific classes, adding attributes that are unique to the class.

The class that is inherited by other class is called superclass and class inheriting is called a subclass.
A subclass is a specialized version of a superclass. It inherits all the instance variables of the superclass and adds it's own.

To inherit a class, you simply incorporate the definition of one class into another by using the extends keyword.

```
lass  super{
      int i,j;
      void showij(){
          system.out.println(" i & j :" +i + " " + j);
      }
  }

class  sub extends super{
       int k;
       void showk(){
         system.out.println(" k:"; +k);
       }

       void sum(){
          system.out.println(" sum is:"; +(i+j+k));
       }
  }

  class  simpleInh (){
       public static void main(string args[]){

          super superob = new super();
          sub   subob = new sub();

          superob.i = 20;
          superob.j = 10;
          system.out.println(" contents of superclass");
          superob.showij();
```

}

outpu

```
        subob.i = 7
        subob.j = 8
        subob.k = 9
        System.out.println("contents of subclass");
        subob.showij();
        subob.showk();
        System.out.println("Sum in subclass:"+(i+j+k));
        subob.sum();
    }
}
```

output:

```
contents of superclass
i & j : 10, 20
contents of subclass
i & j : 7, 8
  k : 9
Sum in subclass
sum is : 24
```

Date: Page:

1.3 Exception Handling and Threading.

Exception Handling

An exception is an abnormal condition that arises in code sequence at runtime. In languages that donot support exception handling, errors must be checked and handled manually.

In Java exception handling is managed via five keywords; try, catch, throw, throws, and finally.

* Program statements to monitor for exceptions are contained within try block.

* If an exception occurs within the try block, it is thrown and it is catched by using 'catch'

* To throw an exception, use keyword throw.

* If a method is capable of causing an exception that it doesnot handle you specify this behaviour by using 'throws' in the method's declaration.

* use 'finally' to create a block of code that will be executed either exception occurs or not.

GURUKUL

1.3 Exception Handling and Threading.

Exception Handling

An exception is an abnormal condition that arises in code sequence at runtime. In languages that donot support exception handling, errors must be checked and handled manually.

In Java exception handling is managed via five keywords; try, catch, throw, throws, and finally.

* Program statements to monitor for exceptions are contained within try block.
* If an exception occurs within the try block, it is thrown and it is catched by using 'catch'

To throw an exception, use keyword throw.

class

{

super

sub

Superob.i = 2.

superob.j = 10;

System.out.println(

Superob.showij();

causing an exception
specify this
in the method's

out of code
exception occurs or

GURUKUL

General form of exception handling block.

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

Simple program:

```
class Exc {
    public static void main (String args[]){
    int d, a ;

    try {
      d = 0 ;
      a = 42/d ;
      System.out.println(" Answer is error." ); }
    catch (ArithmeticException e) {
    System.out.println("Division by zero") ; }
```

```
        System.out.println("After catch statement");
    }
}
```

Date: Page:

output:

Division by zero
After catch statement.

{Explain program written above}

Importance of Exception Handling:

- Separating Error-handling code from Regular code:
  Exception handling provides the means to separate the details of what to do when something out of the ordinary happens from the main logic of program.

- Ability to propagate Errors up the call stack.
  Exception handling in java has the ability to propagate error reporting up the call stack of methods. It is useful when error/exception is found in a series of nested method calls.

- Grouping and Differentiating Error Types
  Java provides several super classes and sub classes that group exceptions based on their type. subclasses & Superclasses provide functionality to handle specific exception types.

GURUKUL

- Meaningf
  are obje
  overrides
  obtain
  of stri
  println(

Multit
  ded progr
  two or
  part of
  each thr
  Hence, m

  enviror
  of disp
  progra
  eg. a
  print

Multitas
  multita
  own s
  limited
  thread

Date: Page:

• Meaningful Error Reporting:

The exceptions thrown in a java program are objects of a class. Since the Throwable class overrides the toString() method, we can easily obtain a description of an exception in form of string and display the description using a println() statement.

Multithreading:

Java has built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines separate path of execution. Hence, multithreading is specialized form of multitasking.

In a thread-based multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. eg. a text editor may format text same time it is printing.

Multitasking threads require less overhead than multitasking processes while processes require own separate address spaces, Ipc is expensive & limited and context switching. on the other hand threads share same address space, Ipc is

Date: Page:

In-expensive and context switching is lower in cost

Multithreading enables us to write efficient programs that make use of the procesing power available in the system, because idle time can be kept to a minimum.

creating a Thread

In the most general sense, we can create a thread by instantiating an object of type Thread we can create a thread in two ways.

- You can implement the Runnable interface
- You can extend the Thread class, itself

creating Multithreaded program:

```
class NewThread implements Runnable{
    String name;
    Thread t;

    NewThread (String threadname) {
        name = threadname;
        t = new Thread (this, name);
        System.out.println("New thread: " +t);
        t.start();
    }
```

GURUKUL

```
Public void run (){
  try{
     for(int i=5; i>0; i--){
        System.out.println(name + ": " +i);
        Thread.sleep(1000);
        }
     }

  catch (InterruptedException e){
     System.out.println(name + " Interrupted");
     }
     system.out.println(name + "exiting.");
  }
}

class MultiThreadDemo{
  public static void main (String args[]){
     new NewThread ("one");
     new NewThread ("Two");
     new NewThread ("Three");

  try{
     Thread.sleep(10000);
     }
  catch (InterruptedException e){
     System.out.println("Main thread Interrupted");
     }
     System.out.println("Main thread exiting.");
  }
}
```

Sample output (May vary depending on execution environment)

```
New thread : Thread[one, 5, main]
New thread : Thread[Two, 5, main]
New thread : Thread[Three, 5, main]
One : 5
Two : 5
Three : 5
one : 4
Two : 4
Three : 4
one : 3
Three : 3
Two : 3
one : 2
Two : 2
Three : 2
Two : 1
one : 1
Three : 1
Two exiting.
one exiting.
Three exiting.
Main Thread exiting.
```

[Explain program if needed]

In, above program 'sleep(10000)' in main() causes the main thread to sleep for ten seconds & causes it to finish last

Date: Page:

Thread Life cycle.

Two ways to determine whether a thread has finished or ~~not~~ cause it to finish last is to use 'isAlive()' and 'join()' keywords

The 'isAlive()' method returns true if the thread is still running else returns false.

The general form is

final boolean isAlive()

final void join() throws InterruptedException

Exauple:

```
class NewThread implements Runnable{
  String name;
  Thread t;

  NewThread (String threadname){
    name = threadname;
    t = new Thread(this, name);
    System.out.println("New thread:" + t);
    t.start();
  }

  Public void run(){
    try{
      for(int i=5; i>0; i--){
        system.out.println("name+" : " +i);
        Thread.sleep(1000);
      }
    }
```

GURUKUL

```
catch (InterruptedException e){
System.out.println(name+": "Interrupted.");
}
System.out.println (name + "exiting.");
}
}

class Demolite{
public static void main (String args[]){
NewThread ob1 = new NewThread("one");
NewThread ob2 = new NewThread("Two");
NewThread ob3 = new NewThread("Three");

System.out.println("Thread one alive:"
+ob1.t.isAlive());
system.out.println("Thread two alive:"
+ob2.t.isAlive());
system.out.println("Thread three Alive:"
+ob3.t.isAlive());

try{
System.out.println("waiting threads to finish");
ob1.t.join();
ob2.t.join();
ob3.t.join();
}

catch (InterruptedException e){
system.out.println("Main thread Interrupted");
}
```

```
        System.out.println("Thread one alive"
                + obj.t.isAlive());
        System.out.println("Thread two alive:"
                + obj.t.isAlive());
        System.out.println("Thread three alive:"
                + obj.t.isAlive());
        System.out.println("Main thread exiting.");
    }
}
```

Sample output: (May depend on execution environment)

```
New Thread : Thread[one, 5, main]
New Thread : Thread[Two, 5, main]
New Thread : Thread[Three, 5, main]
Thread one alive: True
Thread two alive: True
Thread three alive: True
waiting for threads to finish.
one : 5
Two : 5
Three 5
one : 4
Two : 4
Three 4
one : 3
Three : 3
Two : 3
```

```
        System.out.println("Thread one alive"
                + ob1.t.isAlive());
        System.out.println("Thread two alive:"
                + ob2.t.isAlive());
        System.out.println("Thread three alive:"
                + ob3.t.isAlive());
        System.out.println("Main thread exiting.");
    }
}
```

Sample output: (May depend on execution environment)

```
New Thread : Thread[one, 5, main]
New Thread : Thread[Two, 5, main]
New Thread : Thread[Three, 5, main]
Thread one alive: True
Thread two alive: True
Thread three alive: True
waiting for threads to finish.
one : 5
Two : 5
Three : 5
one : 4
Two : 4
Three : 4
one : 3
Three : 3
Two : 3
```

Date: Page:

One : 2
Three : 2
Two : 2
one : 1
Two : 1
Three : 1
one exiting
Two exiting
Three exiting.
Thread one alive : false
Thread Two alive : false
Thread Three alive : false
Main thread exiting.

Exception Handling : Keywords used
[Try catch finally Throws, Throw]

Using Try & catch :
[Defination & exaple from FRONT]

using Throw :
The throw keyword allows an program to throw an exception explicitly. The general form of throw is ;

throw ThrowableInstance;

Here, ThrowableInstance must be an object of type Throwable or subclass of Throwable primitive types. int, char, string cannot be used.

The flow of execution stops immediately after the throw statement; any subsequent statements are not executed.
The nearest try block is inspected to see if it has a catch statement that matches the type of exception. If no match is found next try block is inspected.

```
class Throwdemo {
    static void demoproc () {
        try {
            throw new NullPointerException ("demo");
        }
        catch (NullPointerException e) {
            System.out.println("Inside demoproc");
            throw e;
        }
    }

    Public static void main (String args[]) {
        try {
            demoproc ();
        }
        catch (NullPointerException e) {
        System.out.println ("Recaught : " + e);
        }
    }
}
```

[Explain as Needed]

The flow of execution stops immediately after the throw statement; any subsequent statements are not executed.
The nearest try block is inspected to see if it has a catch statement that matches the type of exception. If no match is found next try block is inspected

```
class Throwdemo {
    static void demoproc () {
      try {
          throw new NullPointerException ("demo");
        }
      catch (NullPointerException e) {
          system.out.println("Inside demoproc");
          throw e;
        }
    }

    Public static void main (String args[]) {
       try {
          demoproc ();
          }
       catch (NullPointerException e) {
        system.out.println ("Recaught : " + e);
          }
    }
}
```

[Explain as Needed]

Date: Page:

output:
caught inside demoproc.
Recaught: java.lang.NullPointerException: demo

using Throws

It a method is capable of causing an exception that it doesnot handle, it must specify this behaviour using 'throws' clause in method's declaration.
The general form is;

```
type method-name(parameter-list) throws exception-list
{
    // body of method
}
```

Here, exception-list is a comma-separated list of the exceptions that a method can throw.

Example:

```
class Throwsdemo{
    static void throwone() throws IllegalAccessException{
    System.out.println("Inside throwone");
    throw new IllegalAccessException("demo");
    }
    public static void main(string args[]){
    try{
```

GURUKUL

```
        throwOne();
    }
    catch(IllegalAccessException e){
        System.out.println("caught" +e);
    }
}
```

output:

```
inside throwOne
caught java.lang.IllegalAccessException : demo.
```

[Explain as needed]

using finally :

When exceptions are thrown, execution in a method is altered. In some cases this behaviour could be a problem because an exception may cause the method to return prematurely.

If we need to execute a block of code either or not any exception occurs we use 'finally' clause.

finally clause makes the code execute even if any catch statement doesnot match the exception.

Example:

GURUKUL

```
        throwone();
    }
    catch(IllegalAccessException e){
        System.out.println("caught" +e);
    }
}
}
```

output:

```
inside throwone
caught java.lang.IllegalAccessException : demo.
```

[Explain on needed]

using finally :

When exceptions are thrown, execution in a method is altered. In some cases this behaviour could be a problem because an exception may cause the method to return prematurely.

If we need to execute a block of code either or not any exception occurs we use 'finally' clause.

finally clause makes the code execute even if any catch statement doesnot match the exception.

Example:

GURUKUL

Date: Page:

```
class FinallyDemo{
    static void procA(){
        try{
            System.out.println("inside procA");
            throw new RuntimeException("demo");
        }
        finally{
            System.out.println("procA's finally");
        }
    }

    Public static void main(String args[]){
        try{
            procA();
        }
        catch (Exception e){
            System.out.println("Exception caught");
        }
    }
}
```

[Explain As Needed]

output:

```
Inside procA
procA's finally
Exception caught.
```

GURUKUL

Date: Page:

```
class finallyDemo{
  static void procA(){
    try{
      System.out.println("inside procA");
      throw new RuntimeException("demo");
    }
    finally{
      System.out.println("procA's finally");
    }
  }

  Public static void main(String args[]){
    try{
      procA();
    }
    catch(Exception e){
      System.out.println("Exception caught");
    }
  }
}
```

[Explain As Needed]

output:

Inside procA
procA's finally
Exception caught.

GURUKUL

1.4 : File IO

Write a simple java program to read from and write to files.

```
import java.io.*;

Public class FileReadWrite
{
   Public static void main(string args[]) throws IOException
   {
      FileWriter Writer = new FileWriter("D:/hello.txt");
      Writer.write("This \n is \n an \n exauple");
      Writer.close();

      FileReader fr = new FileReader("D:/hello.txt");
      char a[] = new char[50];
      fr.read(a);
      for(char c=a)
         system.out.println(c);
      fr.close();
   }
}
```

output:

This
is
an
Exauple.

classes to handle files in Java:

i) FileInputStream : It creates an InputStream that you can use to read bytes from a file.
Two commonly used constructors are:

FileInputStream (String filepath)
FileInputStream (File fileObj)

Both of these can throw a FileNotFoundException.
filepath is the full path name of a file
fileObj is a file object that describes file.

Example:

```
FileInputStream f0 = new FileInputStream ("/autoexec.bat")
File f = new File ("/autoexec.bat");
FileInputStream f1 = new FileInputStream(f);
```

ii) File output stream : It creates an OutputStream that allows you to write bytes to a file. It implements the AutoCloseable, closeable & Flushable interfaces.

FileOutputStream (String filepath)
FileOutputStream (File fileObj)
FileOutputStream (String filepath, boolean append)
File outputStream (File fileobj, boolean append)

* can throw FileNotFoundException.

GURUKUL

iii) Fil
be use
It's two
Fi
Fil

* Either

iv) FileWr
to writ
constru
Filev
Filev
Filew
Filew

* can t

v) Rando
random
interfac
RandomA
positio
It's

GURUK

iii) FileReader : It creates a reader that can be used to read the contents of a file.
It's two common constructors are;

FileReader (String filepath)
FileReader (File fileobj)

* Either of them can throw FileNotFoundException.

iv) FileWriter : It creates a writer which allows to write to a file. Four most commonly used constructors are:

FileWriter (String filePath)
FileWriter (File fileobj)
FileWriter (String filepath, boolean append)
FileWriter (File fileobj, boolean append)

* can throw IOException.

v) RandomAccessFile : It encapsulates a random access file. It implements the interfaces DataInput & DataOutput.
RandomAccessFile is special because it supports positioning requests.
It's basic two constructors are:

iii) FileReader : It creates a reader that can be used to read the contents of a file. It's two common constructors are;

FileReader(String filepath)
FileReader(File fileobj)

* Either of them can throw FileNotFoundException.

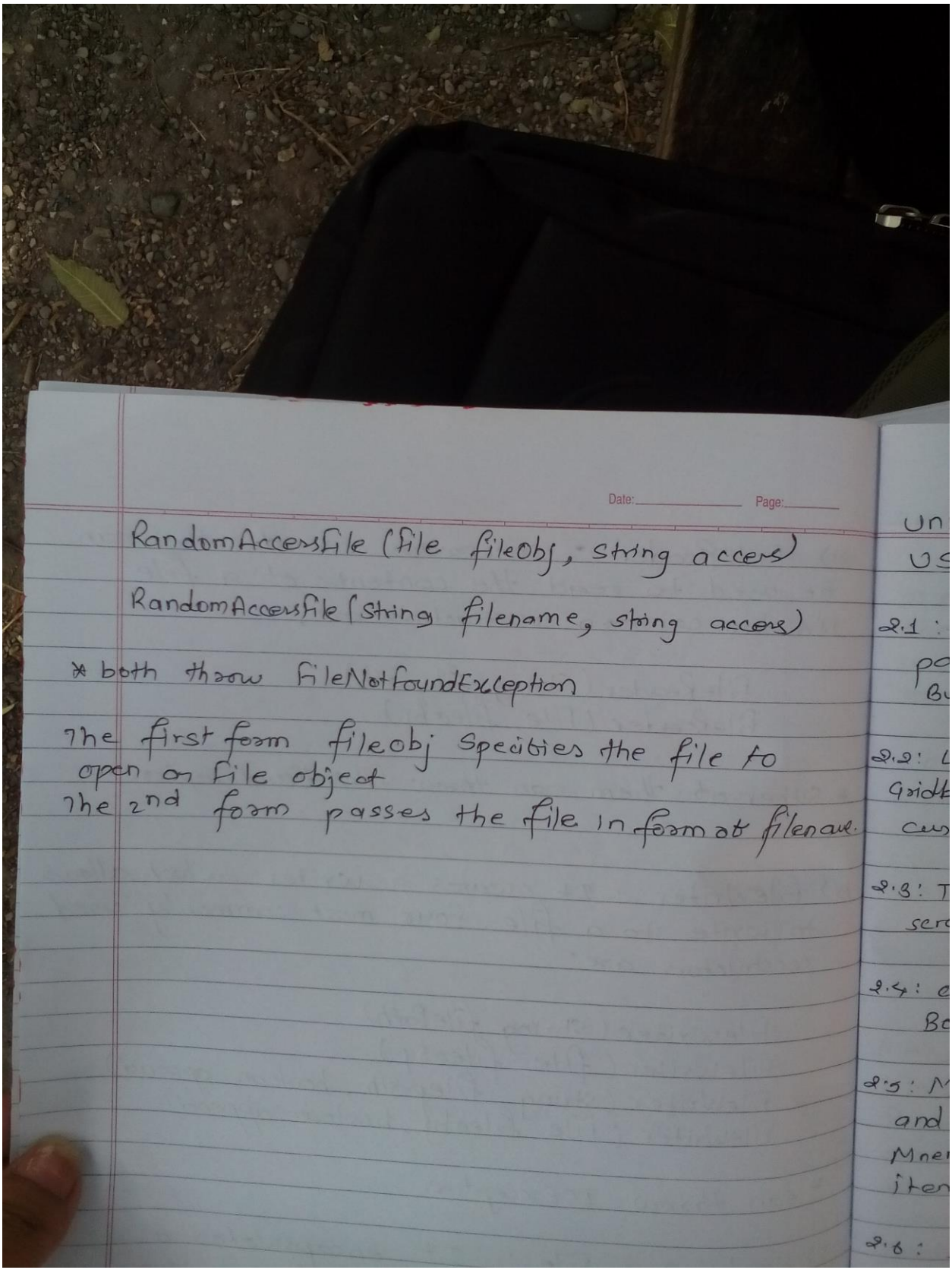
iv) FileWriter : It creates a writer which allows to write to a file. Four most commonly used constructors are:

FileWriter(String filePath)
FileWriter(File fileobj)
FileWriter(String filepath, boolean append)
FileWriter(File fileobj, boolean append)

* can throw IOException.

v) RandomAccessFile : It encapsulates a random access file. It implements the interfaces DataInput & DataOutput. RandomAccessFile is special because it supports positioning requests. It's basic two constructors are:

GURUKUL



Date: Page:

RandomAccessFile (file fileObj, string acces)

RandomAccessFile (String filename, string acces)

* both throw FileNotFoundException

The first form fileobj Specifies the file to open on file object

The 2nd form passes the file in form of filename.

Date: Page:

Unit : 2 :

User Interface components with swing

2.1 : Swing and MVC design patterns : design pattern, MVC pattern, MVC Analysis of swing Buttons.

2.2 : Layout Management : Border Layout, Grid layout, Gridbag Layout, Group Layout, using NO layout managers, custom layout managers.

2.3 : Text Input : Text fields, Password fields, Text areas, scroll pane, Label & labeling components.

2.4 : choice components : check Boxes, Radio Buttons, Borders, combo Boxes, sliders

2.5 : Menus : Menu Building, Icons in Menu items, check Box and Radio Buttons in Menu items, pop-up Menus, keyboard Mnemonics and Accelerators, Enabling and design menu items, Toolbars, ToolTips.

2.6 : Dialog Boxes : option Dialogs, creating dialogs, Data Exchange, file choosers, color choosers.

2.7 : component organizers : split panes, Tabbed pane, Desktop panes & Internal Frames, cascading & Tiling.

2.8 : Advance swing componenti : List, Trees, Tables Progress Bars.

2.2: Layout Management:

Layout managers arrange GUI components in a container for presentation purposes. You can use the layout managers for basic layout capabilities instead of determining every GUI component's exact position and size.

This feature enables an user to focus on the basic look-and-feel and lets the layout managers process most of the layout details. All layout managers implement the interface LayoutManager in package (java.awt)
class container's setLayout method takes an object that implements the LayoutManager interface as an argument.
The basic three ways to arrange GUI components:

1. Absolute positioning: It provides the greatest level of control over a GUI appearance. By setting a container's layout to null, you can specify the absolute position of each GUI component with respect to the upper-left corner of the container by using component methods setSize and setLocation or setBounds.

2. Layout managers: Using layout managers to position elements can be simpler, faster then creating a GUI with absolute positioning, but it lacks precise control over size & positioning of GUI components.

GURUKUL

3. Visual programming in an IDE: IDE's provide tools that make it easy to create GUI's. Each IDE typically provides a GUI design tool that allows you to drag and drop GUI components from a tool box onto a design area.

The user can then position, size & align the GUI components as you like. The IDE generates the Java code the creates the GUI.

Additionaly, you can typically add event-handling code for a particular component by double clicking the component.

FlowLayout: It is the sim

Basic swing program
(JLabel, JTextfield, JTextArea, JButton, JFrame JPanel, JOptionPane)

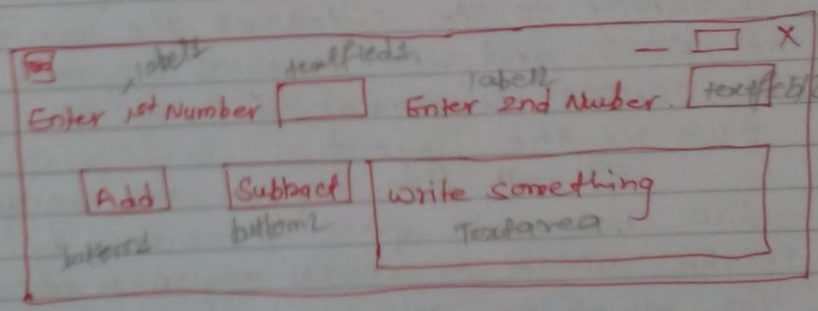
```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;

Public class swingtest extends JFrame{
    String num1, num2;
    int result;

    JFrame j = new JFrame();
    JPanel P = new JPanel();
    JLabel l1 = new JLabel("Enter 1st Number");
    JTextfield tf1 = new JTextfield(5);
    JLabel l2 = new JLabel("Enter 2nd Number");
    JTextFiled tf2 = new JTextfield(5);
    JButton b1 = new JButton("Add");
    JButton b2 = new JButton("Subtract");
    JTextArea ta1 = new JTextArea("Write Something", 5, 40);

    Public static void main(String[] args){
        new swingtest();
    }

    Public swingtest(){
        P.add(l1);
        p.add(tf1);
```

```
p.add(l2);
p.add(tf2);
p.add(b1);
p.add(b2);
p.add(ta1);
add(p);

b1.addActionListener(new Handler());
b2.addActionListener(new Handler());
setLocationRelativeTo(null);
setSize(300,400);
setVisible(true);
setDefaultCloseOperation(EXIT_ON_CLOSE);
}

Private class Handler implements ActionListener{
    public void actionPerformed(ActionEvent ae){
      num1 = tf1.getText();
      num2 = tf2.getText();

      if(ae.getSource == b1){
      result = Integer.parseInt(num1) +
               Integer.parseInt(num2);
      JOptionPane.showMessageDialog(null, result);
      }

      if(ae.getSource == b2){
        result = Integer.parseInt(num1) -
                 Integer.parseInt(num2);
```

```
p.add(l2);
p.add(tf2);
p.add(b1);
p.add(b2);
p.add(ta1);
add(p);

b1.addActionListener(new Handler());
b2.addActionListener(new Handler());
setLocationRelativeTo(null);
setSize(300,400);
setVisible(true);
setDefaultCloseOperation(EXIT_ON_CLOSE);
}

Private class Handler implements ActionListener{
    public void actionPerformed(ActionEvent ae){
    num1 = tf1.getText();
    num2 = tf2.getText();

    if(ae.getSource == b1){
    result = Integer.parseInt(num1) +
             Integer.parseInt(num2);
    JOptionPane.showMessageDialog(null, result);
    }

    if(ae.getSource == b2) {
      result = Integer.parseInt(num1) -
               Integer.parseInt(num2);
```

```
        JOptionPane.showMessageDialog(null, result);
    }
}
}
}
```

this program's output:



Date: Page:

BorderLayout: The BorderLayout layout manager is the default layout manager for a JFrame. It arranges components into five regions: NORTH, SOUTH, EAST, WEST and CENTER.

A BorderLayout limits a container to contain at most five components one in each region. The contents placed in NORTH & SOUTH extend horizontally to the sides of the container. The EAST and WEST regions expand vertically. The CENTER component fills all the remaining space in the layout.

If all five regions are occupied, the entire container's space is covered by GUI components. If the NORTH or SOUTH region is not occupied, the remaining components expand vertically to fill space & if EAST or WEST is not occupied, the GUI component in CENTER expands horizontally to fill remaining space. If the CENTER region is not occupied, the area is left empty. other GUI components do not expand.

GURUKUL

[A bit complex] Date: Page:

```
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;

    public class Borderlayoutdemo extends JFrame
                    implements ActionListener{

    private JButton [] buttons;
    private static final String [] names = {"Hide North",
      "Hide South", "Hide East", "Hide west", "Hide center"};
    private BorderLayout layout;


    public static void main(String args[])
    {  Borderlayoutdemo sn = new Borderlayoutdemo ();
       sn.setVisible (true);
       sn.setSize(400, 300);
       sn.setLocationRelativeTo (null);
       sn.setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    public Borderlayoutdemo (){
       super("Demo");
       layout = new BorderLayout (5,5);
       setLayout (layout);
```

GURUKUL

```
buttons = new JButton[names.length];

for(int i=0; i<names.length; i++)
  {
      buttons[i] = new JButton(names[i]);
      buttons[i].addActionListener(this);
  }

add(buttons[0], BorderLayout.NORTH)
add(buttons[1], BorderLayout.SOUTH)
add(buttons[2], BorderLayout.EAST)
add(buttons[3], BorderLayout.WEST)
add(buttons[4], BorderLayout.CENTER);
}

public void actionperformed(ActionEvent ae){
    for(JButton button : buttons)
      {
          if (ae.getSource() == button)
              button.setVisible(false);
          else button.setVisible(true);
      }

}
}
```

[simple]

```
import java.awt.*;
import java.swing.*;

public class bordertest extends JFrame implements.
        ActionListener{

JButton [] buttons;
String [] names={"East", "west", "North", "South",
                 "center"};
BorderLayout layout;

Public static void main(String args[])
{
   bordertest.bt = new bordertest();
    bt. bor();
    bt. setLocationRelativeTo(null);
    bt. setSize(600,500);
    bt. setVisible(true);
   bt. setDefaultCloseOperation(EXIT_ON_CLOSE);
   }

Public void bor(){
    layout = new BorderLayout(5,5);
    setLayout(layout);
    buttons = new JButton[names.length];
```

GURUKUL

```
for(int i = 0; i<names.length; i++)
{
    buttons[i] = new JButton(names[i]);
    button[i] = addActionListener(this);
}

add(buttons[0], BorderLayout.EAST);
add(buttons[1], BorderLayout.WEST);
add(buttons[2], BorderLayout.NORTH);
add(buttons[3], BorderLayout.SOUTH);
add(buttons[4], BorderLayout.CENTER);

setDefaultCloseOperation(EXIT_ON_CLOSE);
}

Public void actionPerformed(ActionEvent ae)
{
    for(JButton Button: Buttons)
    {
        if(ae.getSource() == button)
            button.setVisible(false);
        else button.setVisible(true);
    }
}
```

GURUKUL

52594

Date: Page:

GridLayout

The GridLayout layout manager divides the container into a grid so that components can be placed in rows and columns

class GridLayout inherits directly from class object & implements interface LayoutManager

Every component in a GridLayout has the same width and height.

The components are added to a GridLayout starting at the top-left cell of the grid and proceeding left to right until the row is full. Then the process continues left to right on the next row of the grid & so on.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
Import java.awt.event.ActionListener;

Public class gridtest extends JFrame implements ActionListener{
   JButton[] buttons;
String[] names={"one", "Two", "Three", "Four", "Five",
                "Six"};
   GridLayout layouts;

Public static void main(String args[]){
    gridtest gt = new gridtest();
    gt.sgt();
```

GURUKUL

```
gt.setSize(600,500);
gt.setVisible(true);
gt.setLocationRelativeTo(null);
gt.setDefaultCloseOperation(EXIT_ON_CLOSE);
}

Public void sgt(){
    layout1 = new GridLayout(3,2,5,5)
    // 3 rows, 2 cols 5/5 vertical-horizontal spacing.
    setLayout(layout1);
    buttons = new JButton[names.length];
    for(int i=0; i< names.length; i++)
    {
        buttons[i] = new JButton(names[i]);
        buttons[i].addActionListener(this);
        add(buttons[i]);
    }
}

public void actionPerformed(ActionEvent ae){
    for(JButton button: buttons)
    {
        if(ae.getSource == button)
        button.setVisible(false);
        else button.setVisible(true);
    }
}
```

JcheckBox

Date: Page:

The JCheckBox and JRadioButton are subclasses of JToggleButton. A JRadioButton is different from a JcheckBox is that normally several JRadioButtons are grouped togehter and are mutually exclusive i.e. only one group can be seleded at any time whereas more than one checkbox can be selected at a time.

```
import javaxx.swing.*;
import java.awt.event.ItemListener;
import java.awt.ItemEvent;

Public class checktest extends JFrame{
    JPanel P = new JPanel();
    JTextArea ta = new JTextArea(10,5)
    JCheckBox bold = new JcheckBox("Bold");
    JChakBox italics = new JCheckBox("Italics");

    Public static void main(String args[])
    {
        checktest ct = new checktest();
        ct.chk;
        ct.setSize(400,300);
        ct.setLocationRelativeTo(null);
        ct.setVisible(true);
        ct.setDefaultcloseOperation(EXIT_ON_CLOSE);
    }
```

GURUKUL

```
Public void chk();
{
    p.add(ta);
    p.add(bold);
    p.add(italics);
    add(p);
    italics.addItemListener(new handler());
    bold.addItemListener(new handler());
}

Public class handler implements ItemListener{
    public void itemStateChanged(ItemEvent e){
     if (bold.isSelected())
      JOptionPane.showMessageDialog(null,
                "You checked Bold");

     else if (italics.isSelected())
      JOptionPane.showMessageDialog(null,
                "You checked italics");
    }
}
}
```

Radio
are sim
states -
buttons
one b

all othe
used to
only

```
import jav
import jav
import jav

Public cl
    JF
    JR
    JR
    JR
    Bu

pub
{
}
```

JRadioButton

Date: Page:

Radio buttons declared with class JRadioButton are similar to checkboxes in that they have two states - selected & not selected. However, radio buttons normally appear as a group in which only one button can be selected at a time selecting a different radio button forces all others to be deselected. Radio buttons are used to represent mutually exclusive options. i.e. only one option at a time.

```
import javax.swing.*;
Import java.awt.event.ActionEvent;
Import java.awt.event.ActionListener;

Public class radiotest extends JFrame implements ActionListener{
    JPanel p = new JPanel();
    JRadioButton rb1 = new JRadioButton("Bold", true);
    JRadioButton rb2 = new JRadioButton("Italics", false);
    JRadioButton rbg = new JRadioButton("plain", false);
    ButtonGroup rbg = new ButtonGroup();

    public static void main(String args[])
    {
        radiotest rt = new radiotest();
        rt.rad();
        rt.setVisible(true);
        rt.setSize(400, 300);
        rt.setDefaultCloseOperation(EXIT_ON_CLOSE);
        rt.setLocationRelativeTo(null);
    }
```

```
public void rad(){
    p.add(rb1);
    p.add(rb2);
    p.add(rb3);
    add(p);
    rbg.add(rb1);
    rbg.add(rb2);
    rbg.add(rb3);

    rb1.addActionListener(this);
    rb2.addActionListener(this);
    rb3.addActionListener(this);
}

public void actionPerformed(ActionEvent ae){
    if(ae.getSource()==rb1)
        JOptionPane.showMessageDialog(null,"click Bold");
    else if(ae.getSource()==rb2)
        JOptionPane.showMessageDialog(null,"click Italics");
    else if(ae.getSource()==rb3)
        JOptionPane.showMessageDialog(null,"click plain");
}

}
```

JComboBox

A comboBox also referred to as a drop-down list enables the user to select one item from a list. comboxes are implemented with class JComboBox, which extends class JComponent. JComboBoxes generate ItemEvents just as JCheckBoxes and JRadioButtons do.

```
import java.awt.BorderLayout
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import javax.swing.*;

public class combotest extends JFrame implements ItemListener{
    JPasswordField jp = new JPasswordField(10);
    JTextfied tf = new JTextfield(15);
    string array = {"Admin", "User", "system", "Guest"};
    JComboBox cb = new JComboBox(array);

    Public static void main(String args[])
    {
        combotest ct = new combotest();
        ct.com();
        ct.setVisible(true);
        ct.setSize(400, 300);
        ct.setDebaultcloseOperation("EXIT-ON-CLOSE");
        ct.setLocationRelativeTo(null);
    }
```

JComboBox

Date: Page:

A comboBox also referred to as a drop-down list enables the user to select one item from a list. comboxes are implemented with class JComboBox, which extends class JComponent. JComboBoxes generate ItemEvents just as JCheckBoxes and JRadioButtons do.

```
import java.awt.BorderLayout
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import javax.swing.*;

public class combotest extends JFrame implements ItemListener{
    JPasswordField jp = new JPasswordField(10);
    JTextfied tf = new JTextfield(15);
    string array = {"Admin", "User", "system", "Guest"};
    JComboBox cb = new JComboBox(array);

    Public static void main(string args[])
    {
        combotest ct = new combotest();
        ct.com();
        ct.setVisible(true);
        ct.setSize(400, 300);
        ct.setDebaultcloseOperation("EXIT_ON_CLOSE");
        ct.setLocationRelativeTo(null);
    }
```

GURUKUL

```
Publi
public void com()
{
add(cb, BorderLayout.NORTH);
add(jp, BorderLayout.SOUTH);
add(tf, BorderLayout.CENTER);
cb.addItemListener(this);
}

Public void itemStateChanged(ItemEvent e)
{
  if(e.getStateChange == = ItemEvent.SELECTED)
  {
    String str = e.getItem().toString();
    tf.setText("You have selected" +str);
  }
}
}
```

JProgressBar

Date: Page:

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;

public class prgabar extends JFrame{
    JProgressBar prg;
    Timer t;
    JButton btn;
    int interval = 1000;
    JPanel jp;
    int i;

    public static void main(String args[])
    {
        prgabar bar = new prgabar();
        bar.setSize(300,300);
        bar.setComponent();
        bar.setVisible(true);
        bar.setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

void setComponent(){
    jp = new JPanel();
    prg = new JProgressBar(0,20);
    prg.setValue(0);

    btn = new JButton("Start");
    jp.add(prg);
    jp.add(btn);
    add(jp);
```

```
btn.addActionListener(~~new~~ this){
    public void actionPerformed(ActionEvent ea){
        i=0;
        t.start();
        btn.setEnabled(false);
    }

t=new Timer(interval, new ActionListener(){
    public void actionPerformed(ActionEvent ~~arg~~){
        if(i==20){
            t.stop();
            btn.setEnabled(true);
        }
        else
        {i++;
        prg.setValue(i);
        }
    }
});
    }
}
}
```

JTabbedPane

```
import javax.swing.*;

public class usingtab extends JFrame{
    JPanel jp1;
    JPanel jp2;
    JLabel l1;
    JLabel l2;
    JTabbedPane tp;

    public static void main(String args[]){
        usingtab ut = new usingtab();
        ut.test();
        ut.setSize(300,300)
        ut.setVisible(true);
        ut.setDefaultCloseOperation(EXIT_ON_CLOSE)
    }

    Public void test(){
        l1 = new JLabel("viewing 1st Tab");
        l2 = new JLabel("viewing 2nd Tab");
        jp1 = new JPanel();
        jp2 = new JPanel();
        tp = new JTabbedPane();
        jp1.add(l1);
        jp2.add(l2);
        tp.add("1st Tab", jp1);
        tp.add("2nd Tab", jp2);
        add(tp);
    }
}
```

JTabbedPane

```
import javax.swing.*;

public class usingtab extends JFrame {
    JPanel jp1;
    JPanel jp2;
    JLabel l1;
    JLabel l2;
    JTabbedPane tp;

    public static void main(String args[]){
        usingtab ut = new usingtab();
        ut.test();
        ut.setSize(300,300)
        ut.setVisible(true);
        ut.setDefaultCloseOperation(EXIT_ON_CLOSE)
    }

    Public void test(){
        l1 = new JLabel("viewing 1st Tab");
        l2 = new JLabel("viewing 2nd Tab");
        jp1 = new JPanel();
        jp2 = new JPanel();
        tp = new JTabbedPane();
        jp1.add(l1);
        jp2.add(l2);
        tp.add("1st Tab", jp1);
        tp.add("2nd Tab", jp2);
        add(tp);
    }
}
```

Design Pattern : MVC pattern

Swing uses the Model-view-controller architecture as the fundamental design behind each of its components.
In general, MVC breaks GUI components into three elements. Each of these element defines & determines how the component behaves.

i) Model : The model maintains the state data for each component. There are different models for different types of components.
For eg: the model of the scrollbar component might contain information about the current position of it's adjustable "thumb", it's minimum & maximum values and the thumb's width.
A menu on the other hand, may simply contain a list of the menu items.

The data of each ~~model~~ component remains the same no matter how the component is painted on the screen i.e. model data exists independent of the component's visual representation.

ii) View : The view refers how you see the component on the screen. The user interacts with the view when any action is performed on the view or view state is changed the view notifies the controller about the action

Design pattern : MVC pattern

Swing uses the Model-view-controller architecture as the fundamental design behind each of its components.

In general, MVC breaks GUI components into three elements. Each of these element defines & determines how the component behaves.

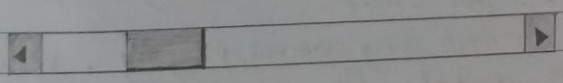
i) Model : The model maintains the state data for each component. There are different models for different types of components.
For eg: the model of the scrollbar component might contain information about the current position of it's adjustable "thumb", it's minimum & maximum values and the thumb's width.
A menu on the other hand, may simply contain a list of the menu items.

The data of each ~~model~~ component remains the same no matter how the component is painted on the screen i.e. model data exists independent of the component's visual representation.

ii) View : The view refers how you see the component on the screen. The user interacts with the view. when any action is performed on the view or view state is changed the view notifies the controller about the action.

or change in state of view. Now it is the job of controller to handle that.

iii) Controller: The controller is the portion of the user interface that dictates how the component interacts with events.
Events may come in many forms a mouse click, gaining or losing focus, a keyboard event etc.
The controller decides how each component will react to the event.



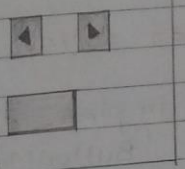
Model	view.	controller.
Minimum = 0 Maximum = 100 value = 15 width = 5		Accept mouse clicks on end buttons. Accept mouse drag on thumb.

Fig: Scrollbar & MVC elements of scrollbar.

GURUKUL

Date: Page:

Summary:

i) The user interacts with the view. any changes in view are notified to the controller.

ii) The controller takes the action & determines the necessary means to manipulate the model. The controller may ask the view to change in case of events. eg: enabling & disabling buttons.

iii) The model notifies the view when it's state has changed and the view may also ask the model for state.
eg: if a new song starts playing, view may ask the model for song name, duration, artist which will be displayed in view.

MVC analysis of swing Buttons:

The interface implemented for swing buttons is called 'ButtonModel'. All the classes implementing that interface can define the state of the various kinds of buttons.
The swing library contains a single class called 'DefaultButtonModel', that implements 'ButtonModel' interface.
The data maintained by a button model by

looking at the properties of the 'ButtonModel' interface.

Properties of the ButtonModel Interface

property value	value
actionCommand	The action command string associated with this button
mnemonic	The keyboard mnemonic for this button.
armed	true if the button was pressed and the mouse is still over the button
enabled	true if the button is selectable.
pressed	true if the button was pressed but the mouse hasn't yet been released.
rollover	true if the mouse is over the button
selected	true if the button has been toggled on.

Each JButton object stores a button model which can be retrieved.

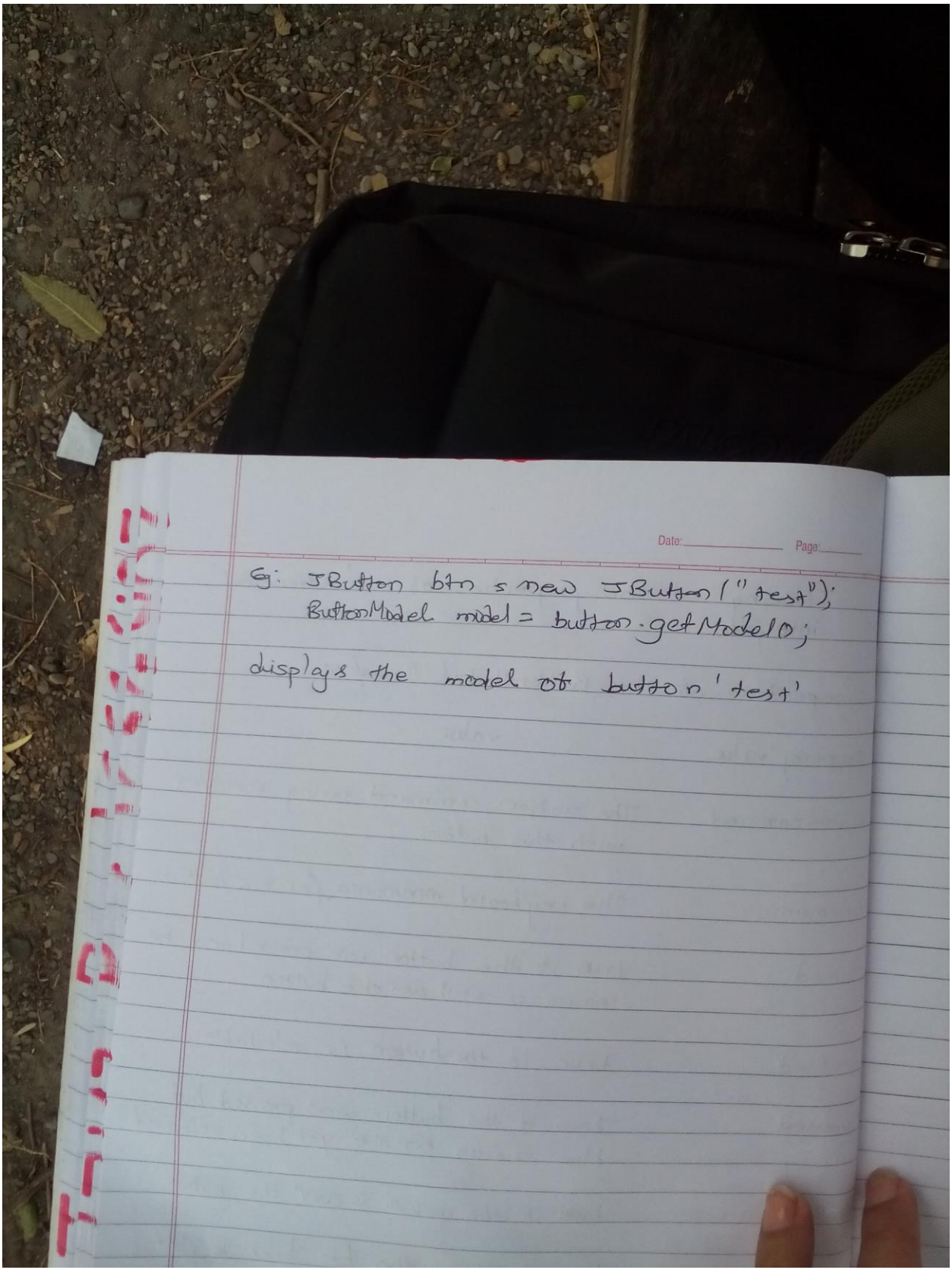
looking at the properties of the 'ButtonModel' interface.

Properties of the ButtonModel Interface.

property value	value
actionCommand	The action command string associated with this button.
mnemonic	The keyboard mnemonic for this button.
armed	true if the button was pressed and the mouse is still over the button.
enabled	true if the button is selectable.
pressed	true if the button was pressed but the mouse hasn't yet been released.
rollover	true if the mouse is over the button
selected	true if the button has been toggled on.

Each JButton object stores a button model which can be retrieved.

GURUKUL


Date: Page:
Eg: JButton btn = new JButton("test");
ButtonModel model = button.getModel();
displays the model of button 'test'