Improvement Test – Nov. 2017
Solution Scheme

Sub:	JAVA & J2EE					Sub Code:	10IS753	Branch:	ISE
Date:	18/11/2017	Duration:	90 min's	Max Marks:	50	Sem / Sec:	A&B		OBE

Answer any FIVE FULL Questions

MARKS

1 a. Describe the thread priority? How to assign and get the thread priority? [6M]

Solution:

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher-priority threads get more CPU time than lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower priority thread.
To set a thread's priority, use the **set Priority( )** method, which is a member of **Thread**.
final void setPriority(int *level*)
Here, *level* specifies the new priority setting for the calling thread. The value of *level* must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify **NORM_PRIORITY**, which is currently 5. These priorities are defined as **static final** variables within **Thread**. You can obtain the current priority setting by calling the **getPriority( )** method of **Thread**,
shown here:
final int getPriority( )

```
// Demonstrate thread priorities.
class clicker implements Runnable {
long click = 0;
Thread t;
private volatile boolean running = true;
public clicker(int p) {
t = new Thread(this);
t.setPriority(p);
}
public void run() {
while (running) {
click++;
}
}
public void stop() {
running = false;
}
public void start() {
t.start();
```

```
}
}
class HiLoPri {
public static void main(String args[]) {
Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
lo.start();
hi.start();
try {
Thread.sleep(10000);
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
lo.stop();
hi.stop();
// Wait for child threads to terminate.
try {
hi.t.join();
lo.t.join();
} catch (InterruptedException e) {
System.out.println("InterruptedException caught");
}
System.out.println("Low-priority thread: " + lo.click);
System.out.println("High-priority thread: " + hi.click);
}
}
```

The output of this program, shown as follows when run under Windows, indicates that the threads did context switch, even though neither voluntarily yielded the CPU nor blocked for I/O. The higher-priority thread got the majority of the CPU time.

Low-priority thread: 4408112
High-priority thread: 589626904

b. Write a java program to create two threads, one display “information” and other display “science” three times. [4M]

Solution:

```
class CS extends Thread
{
public void run()
{
for(int i=1; i<=5; i++)
System.out.println("Information " + i);
}
}
class IS extends Thread
{
public void run()
```

```
{
for(int i=1; i<=5; i++)
System.out.println("Science " + i);
}
}
public class CS_IS_ThreadProgram {
public static void main(String args[])
{
CS c1 = new CS();
c1.start();
IS i1 = new IS();
i1.start();
}
}
```

2. Explain the delegation event model. Briefly explain about any two event classes. [10M]

Solution:

The modern approach to handling events is based on the delegation event model, which defines standard and consistent mechanisms to generate and process events. Its concept is quite simple: a source generates an event and sends it to one or more listeners. In this scheme, the listener simply waits until it receives an event. Once an event is received, the listener processes the event and then returns. The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is able to "delegate" the processing of an event to a separate piece of code. In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them.

Event Classes: The classes that represent events are at the core of Java's event handling mechanism.
The following are the list of Event classes.

1) The Action Event Class: An Action Event is generated when a button is pressed, a list item is double clicked, or a menu item is selected.
2) The Adjustment Event Class: An Adjustment Event is generated by a scroll bar.
3) The Component Event Class: A Component Event is generated when the size, position, or visibility
. of a component is changed. There are four types of component events.
4) The Container Event Class A Container Event is generated when a component is added to or removed from a container x The Focus Event Class : A Focus Event is generated when a component gains or losses input focus

5) The Input Event Class : The abstract class Input Event is a subclass of Component Event and is the Super class for component input events. Its subclasses are Key Event and Mouse Event.
6) The Item Event Class : An Item Event is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected
7) The Key Event Class A Key Event is generated when keyboard input occurs.
8) The Mouse Event Class There are eight types of mouse events

9) The Mouse Wheel Event Class The Mouse Wheel Event class encapsulates a mouse wheel event.
10) The Text Event Class Instances of this class describe text events. These are generated by text fields and text areas when characters are entered by a user or program.
11) The Window Event Class There are ten types of window events. The Window Event class defines integer constants that can be used to identify them.

The Mouse Event Class:

There are eight types of mouse events. The **Mouse Event** class defines the following integer constants that can be used to identify them:

MOUSE_CLICKED The user clicked the mouse.
MOUSE_DRAGGED The user dragged the mouse.
MOUSE_ENTERED The mouse entered a component.
MOUSE_EXITED The mouse exited from a component.
MOUSE_MOVED The mouse moved.
MOUSE_PRESSED The mouse was pressed.
MOUSE_RELEASED The mouse was released.
MOUSE_WHEEL The mouse wheel was moved.

MouseEvent is a subclass of **InputEvent**. Here is one of its constructors:
MouseEvent(Component *src*, int *type*, long *when*, int *modifiers*,int *x*, int *y*, int *clicks*, boolean *triggersPopup*)
Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when a mouse event occurred. The coordinates of the mouse are passed in *x* and *y*. The click count is passed in *clicks*. The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform.
Two commonly used methods in this class are **getX( )** and **getY( )**.
This return the X and Y coordinates of the mouse within the component when the event occurred.
int getX( )
int getY( )
Alternatively, you can use the **getPoint( )** method to obtain the coordinates of the mouse.
Point getPoint( )

It returns a **Point** object that contains the X,Y coordinates in its integer members: **x** and **y**.
The **translatePoint( )** method changes the location of the event.

void translatePoint(int *x*, int *y*)

Here, the arguments *x* and *y* are added to the coordinates of the event.
The **getClickCount( )** method obtains the number of mouse clicks for this event.
Its signature is shown here:
int getClickCount( )

The **isPopupTrigger( )** method tests if this event causes a pop-up menu to appear on this platform. Its form is shown here:

boolean isPopupTrigger( )
Also available is the **getButton( )** method, shown here:
int getButton( )
It returns a value that represents the button that caused the event. The return value will be one of these constants defined by **Mouse Event**:
NOBUTTON BUTTON1 BUTTON2 BUTTON3
The **NOBUTTON** value indicates that no button was pressed or released.
Java SE 6 added three methods to **MouseEvent** that obtain the coordinates of the mouse relative to the screen rather than the component. They are shown here:
Point getLocationOnScreen( )
int getXOnScreen( )
int getYOnScreen( )

The **getLocationOnScreen( )** method returns a **Point** object that contains both the X and Y coordinate. The other two methods return the indicated coordinate.

The Key Event Class:

A **Key Event** is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants: **KEY_PRESSED**, **KEY_RELEASED**, and **KEY_TYPED**. The first two events are generated when any key is pressed or released. The last event occurs only when a character is generated. Remember, not all key presses result in characters. For example, pressing SHIFT does not generate a character.

There are many other integer constants that are defined by **KeyEvent**. For example, **VK_0** through **VK_9** and **VK_A** through **VK_Z** define the ASCII equivalents of the numbers and letters. Here are some others:

VK_ALT	VK_DOWN	VK_LEFT	VK_RIGHT
VK_CANCEL	VK_ENTER	VK_PAGE_DOWN	VK_SHIFT
VK_CONTROL	VK_ESCAPE	VK_PAGE_UP	VK_UP

The **VK** constants specify *virtual key codes* and are independent of any modifiers, such as control, shift, or alt.**Key Event** is a subclass of **Input Event**. Here is one of its constructors:

KeyEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *code*, char *ch*)

Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*. The system time at which the key was pressed is passed in *when*. The *modifiers*argument indicates which modifiers were pressed when this key event occurred. The virtual key code, such as **VK_UP**, **VK_A**, and so forth, is passed in *code*. The character equivalent (if one exists) is passed in *ch*. If no valid character exists, then *ch* contains **CHAR_UNDEFINED**.For **KEY_TYPED** events, *code* will contain **VK_UNDEFINED**.

The **Key Event** class defines several methods, but the most commonly used ones are **getKeyChar( )**, which returns the character that was entered, and **getKeyCode( )**, which returns the key code. Their general forms are shown here:

char getKeyChar( )
int getKeyCode( )

If no valid character is available, then **getKeyChar( )** returns **CHAR_UNDEFINED**. When a **KEY_TYPED** event occurs, **getKeyCode( )** returns **VK_UNDEFINED**.

3. What is an exception? Explain the usage of throw and throws keyword. [10M]

Solution:

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.
The Three Kinds of Exceptions x Checked exceptions are subject to the Catch or Specify Requirement. All exceptions are checked exceptions, except for those indicated by Error, Runtime Exception, and their subclasses.
Errors are not subject to the Catch or Specify Requirement. Errors are those exceptions indicated by Error and its subclasses. x Runtime exceptions are not subject to the Catch or Specify Requirement. Runtime exceptions are those indicated by Runtime Except ion and its subclasses. Valid Java programming language code must honor the Catch or Specify Requirement. This means that code that might throw certain exceptions must be enclosed by either of the following:

• A try statement that catches the exception. The try must provide a handler for the exception, as described in Catching and Handling Exceptions. A method that specifies that it can throw the exception. The method must provide a throws clause that lists the exception, as described in Specifying the Exceptions Thrown by a Method. Code that fails to honor the Catch or Specify Requirement will not compile. This example describes how to use the three exception handler components — the try, cat ch, and finally blocks The first step in constructing an exception handler is to enclose the code that might throw an exception within a try block. In general, a try block looks like the following. try { code } catch and finally blocks . . . Example : private Vector vector; private static final int SIZE = 10; PrintWriter out = null; try { System.out.println("Entered try statement"); out = new PrintWriter(new FileWriter("OutFile.txt")); for (int i = 0; i < SIZE; i++) { out.println("Value at: " + i + " = " + vector.elementAt(i)); } } **The catch Blocks**
You associate exception handlers with a try block by providing one or more catch blocks directly after the try block. No code can be between the end of the try block and the beginning of the first catch block.

Throw:
Program to throw an exception explicitly, using the **throw** statement. The general form of **throw** is shown here:
throw ThrowableInstance;
Here, ThrowableInstance must be an object of type **Throwable** or a subclass of **Throwable**. Primitive types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions. There are two ways you can obtain a **Throwable** object: using a parameter in a **catch** clause, or creating one with the **new** operator. The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed. The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of exception. If it does find a match, control is transferred to that statement. If

not, then the next enclosing **try** statement is inspected, and so on. If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace.

sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

```
// Demonstrate throw.
class ThrowDemo {
static void demoproc() {
try {
throw new NullPointerException("demo");
} catch(NullPointerException e) {
System.out.println("Caught inside demoproc.");
throw e; // rethrow the exception
}
}
public static void main(String args[]) {
try {
demoproc();
} catch(NullPointerException e) {
System.out.println("Recaught: " + e);
}
}
}
```

This program gets two chances to deal with the same error. First, **main( )** sets up an exception context and then calls **demoproc( )**. The **demoproc( )** method then sets up another exception handling context and immediately throws a new instance of **NullPointerException**, which is caught on the next line. The exception is then rethrown. Here is the resulting output:

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

Throws

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a **throws** clause in the method's declaration. A **throws** clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses. All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result. This is the general form of a method declaration that includes a **throws** clause:

```
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.Following is an example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a **throws** clause to declare this fact, the program will not compile.

```
// This program contains an error and will not compile.
class ThrowsDemo {
static void throwOne() {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
throwOne();
```

```
}
}
```

To make this example compile, you need to make two changes. First, you need to declare that **throwOne( )** throws **IllegalAccessException**. Second, **main( )** must define a **try**/**catch** statement that catches this exception. The corrected example is shown here:

```
// This is now correct.
class ThrowsDemo {
static void throwOne() throws IllegalAccessException {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
try {
throwOne();
} catch (IllegalAccessException e) {
System.out.println("Caught " + e);
}
}
}
```

Here is the output generated by running this example program:

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

4. Explain the lifecycle of an applet and write an applet program to display the message "Hello World" [10M]

Solution:

Life Cycle of an Applet:

All but the most trivial applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution. Four of these methods, **init( )**, **start( )**, **stop( )**, and **destroy( )**, apply to all applets and are defined by **Applet**. Default implementations for all of these methods are provided. Applets do not need to override those methods they do not use. However, only very simple applets will not need to define all of them. AWT-based applets will also override the **paint( )** method, which is defined by the AWT **Component** class. This method is called when the applet's output must be redisplayed. (Swing-based applets use a different mechanism to accomplish this task.) These five methods can be assembled into the skeleton shown here:

An Applet Skeleton

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

/* <applet code="AppletSkel" width=300 height=100>
 </applet> */
```

```
public class AppletSkel extends JApplet
{

    // Called first.
    public void init()
    {
       // initialization
    }

   /* Called second, after init(). Also called whenever the applet is restarted.*/

   public void start()
      {
            // start or resume execution
       }


  // Called when the applet is stopped.

    public void stop ()
   {
         // suspends execution
   }

  /* Called when applet is terminated. This is the last method executed. */
   public void destroy()
   {
         // perform shutdown activities
   }

   // Called when an applet's window must be restored.

  public void paint(Graphics g)
   {
      // redisplay contents of window
   }

  }
```

Applet Initialization and Termination

It is important to understand the order in which the various methods shown in the skeleton are called. When an applet begins, the following methods are called, in this sequence:

1. init( )
2. start( )
3. paint( )

When an applet is terminated, the following sequence of method calls takes place:
1. stop( )

2. destroy( )

init( ):

The init( ) method is the first method to be called. This is where you should initialize variables.This method is called only once during the run time of your applet.

start( ):

The start( ) method is called after init( ). It is also called to restart an applet after it has been stopped. Whereas init( ) is called once—the first time an applet is loaded—start( ) is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at start( ).

paint( ):

The paint( ) method is called each time your applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored. paint( ) is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, paint( ) is called. The paint( ) method has one parameter of type Graphics. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

stop( ):

The stop( ) method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When stop( ) is called, the applet is probably running. You should use stop( ) to suspend threads that don't need to run when the applet is not visible. You can restart them when start( ) is called if the user returns to the page.

destroy( ):

The destroy( ) method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The stop( ) method is always called before destroy( ).

Overriding update( ):

In some situations, your applet may need to override another method defined by the AWT, called update( ). This method is called when your applet has requested that a portion of its window be redrawn. The default version of update( ) simply calls paint( ). However, you can override the update( ) method so that it performs more subtle repainting. In general, overriding update( ) is a specialized technique that is not applicable to all applets.

Program -4M

Program:

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

/* <applet code="AppletSkel" width=300 height=100>
 </applet> */
```

```
public class AppletSkel extends JApplet
{

    public void init()
    {
       setBackground(Color.cyan);
       setForeground(Color.red);

    }
    public void paint(Graphics g)
   {
     g.drawString("HelloWorld");
   }
}
```

5. Write a servlet program to add a cookie. [6M]

Solution:

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ServletDemo extends HttpServlet{

        public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException{

                response.setContentType("text/html");
                PrintWriter pw = response.getWriter();

                Cookie cookie = new Cookie("url","mkyong dot com");
                cookie.setMaxAge(60*60); //1 hour
                response.addCookie(cookie);

                pw.println("Cookies created");
        }
}
```

b. Write a simple html file to pass the parameter to servlet and display the parameter values accepted by servlet. [4M]

Solution:

```
<html>

<head>
<title>Login Form</title>
</head>
<body>
        <h2>Login Page</h2>
        <p>Please enter your username and password</p>
        <form method="GET" action="loginServlet">
                <p>
                        Username <input type="text" name="userName" size="50">
                </p>
                <p>
                        Password <input type="text" name="password" size="20">
                </p>
                <p>
                        <input type="submit" value="Submit" name="B1">
                </p>
        </form>
        <p> </p>
</body>
</html>
```

6. Write short notes on i)session tracking ii)cookie [10M]

Solution:

i)session tracking

Why id it needed : In many internet applications it is important to keep track of the session when the user moves from one page to another or when there are different users logging on to the same website at the same time x There are three typical solutions to this problem.

Cookies.
x HTTP cookies can be used to store information about a shopping session, and x each subsequent connection can look up the current session and then extract information about that session from some location on the server machine. x This is an excellent alternative, and is the most widely used approach. **URL Rewriting.** x You can append some extra data on the end of each URL that identifies the session, and the server can associate that session identifier with data it has stored about that session. x This is also an excellent solution, and even has the advantage that it works with browsers that don't support cookies or where the user has disabled cookies. x However, it has most of the same problems as cookies, namely that the server-side program has a lot of straightforward but tedious processing to do. In addition, you have to be very careful that every URL returned to the user (even via indirect means like x Hidden form fields. x HTML forms have an entry that looks like the following: <INPUT TYPE="HIDDEN" NAME="session" VALUE="...">. x This means that, when the form is submitted, the specified name and value are included in the GET or POST data. x This can be used to store information about the session. x However, it has the major disadvantage that it only works if every page is dynamically generated, since the whole point is that each session has a unique identifier.

• Servlets solution : x The HttpSession API. is a high-level interface built on top of cookies or URL-rewriting. In fact, on many servers, they use cookies if the browser supports them, but automatically revert to URL-rewriting when cookies are unsupported or explicitly disabled. x The servlet author doesn't need to bother with many of the details, doesn't have to explicitly manipulate cookies or information appended to the URL, and is automatically given a convenient place to store data that is associated with each session.

example,

```
HttpSession session = request.getSession(true);
ShoppingCart previousItems = (ShoppingCart)session.getValue("previousItems");
if (previousItems != null)
{
doSomethingWith(previousItems);
}
else
{
previousItems = new ShoppingCart(...);
doSomethingElseWith(previousItems);
 }
```

ii)cookie

Cookies are small bits of textual information that a Web server sends to a browser and that the browser returns unchanged when visiting the same Web site or domain later. By having the server read information it sent the client previously, the site can provide visitors with a number of conveniences like:

x Identifying a user during an e-commerce session.. x Avoiding username and password.

x Customizing a site. x Focusing advertising. To send cookies to the client, a servlet would create one or more cookies with the appropriate names and values via **new Cookie (name, value)**

Placing Cookies in the Response Headers

The cookie is added to the Set-Cookie response header by means of the addCookie method of HttpServletResponse.

Example:

```
 Cookie userCookie = new Cookie("user", "uid 1234");
response.addCookie(userCookie);
```

To send cookies to the client, you created a Cookie then used addCookie to send a Set-Cookie HTTP response header.

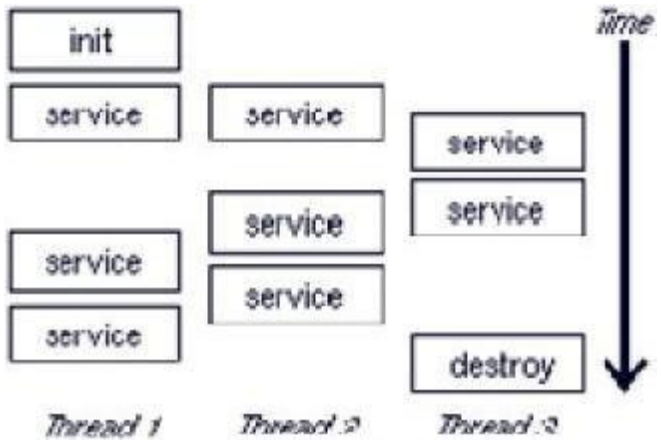
To read the cookies that come back from the client, call getCookies on the HttpServletRequest.

This returns an array of Cookie objects corresponding to the values that came in on the Cookie HTTP request header x Once this array is obtained, loop down it, calling getName on each Cookie until find one matching the name you have in mind. You then call getValue on the matching Cookie, doing some processing specific to the resultant value.

```
public static String getCookieValue(Cookie[] cookies, String cookieName, String defaultValue)
{
for(int i=0; i<cookies.length; i++)
{
 Cookie cookie = cookies[i];
if (cookieName.equals(cookie.getName()))
 return(cookie.getValue());
}
return(defaultValue);
 }
```

7. What are servlets? Explain the servlet lifecycle with an example. [10M]

Solution:



The life cycle of a servlet is controlled by the container in which the servlet has been deployed. When a request is mapped to a servlet, the container performs the following steps: 1. If an instance of the servlet does not exist, the web container: a. Loads the servlet class b. Creates an instance of the servlet class c. Initializes the servlet instance by calling the init method. Initialization is covered in Initializing a Servlet 2. Invokes the service method, passing a request and response object. 3. If the container needs to remove the servlet, it finalizes the servlet by calling the servlet's destroy method.

Example

```
 import java.io.*;
import javax.servlet.*;
import javax.servlet.http. *;
public class HelloClientServlet extends HttpServlet
{ protected void doGet(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException
res.setContentType("text/html");
PrintWriter out = res.getWriter();
out.println("<HTML><HEAD><TITLE>Hello Client !</TITLE>"+ "</HEAD><BODY>Hello Client !</BODY></HTML>"); out.close();
}
 public String getServletInfo()
 {
 return "HelloClientServlet 1.0 by Stefan Zeiger";
}
 }
```

8 Explain about method overloading and overriding in java with examples. [10M]

Solution:

Two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading. Method overloading is one of the ways that Java supports polymorphism. If you have never used a language that allows the overloading of methods, then the concept may seem strange at first. But as you will see, method overloading is one of Java's most exciting and useful features.When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters.While overloaded methods may have different return types, the return type alone isinsufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

example:

```
// Demonstrate method overloading.
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
// Overload test for one integer parameter.
void test(int a) {
System.out.println("a: " + a);
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// overload test for a double parameter
double test(double a) {
System.out.println("double a: " + a);
return a*a;
}
}
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
double result;
// call all versions of test()
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of ob.test(123.25): " + result);
}
}
```

This program generates the following output:

```
No parameters
a: 10
a and b: 10 20
```

double a: 123.25
Result of ob.test(123.25): 15190.5625

Overriding Methods

xWhen a method is called on an object, Java looks for that method definition in the class of that object, and if it doesn't find one, it passes the method call up the class hierarchy until a method definition is found. xMethod inheritance enables you to define and use methods repeatedly in subclasses without having to duplicate the code itself.
xHowever, there may be times when you want an object to respond to the same methods but have different behavior when that method is called. In this case, you can override that method. Overriding a method involves defining a method in a subclass that has the same signature as a method in a superclass. Then, when that method is called, the method in the subclass is found and executed instead of the one in the superclass.

Creating Methods that Override Existing Methods To override a method, all you have to do is create a method in your subclass that has the same signature (name, return type, and parameter list) as a method defined by one of your class's superclasses. Because Java executes the first method definition it finds that matches the signature, this effectively "hides" the original method definition.

The PrintClass class

```
class PrintClass
{
int x = 0; int y = 1; void printMe() { System.out.println("X is " + x + ", Y is " + y);
System.out.println("I am an instance of the class " + this.getClass().getName());
```

Create a class called PrintSubClass that is a subclass of (extends)
PrintClass. class PrintSubClass extends PrintClass

```
{
int z = 3; public static void main(String args[])
{
PrintSubClass obj = new PrintSubClass(); obj .printMe(); }
}
```

Here's the output from PrintSubClass: X is 0, Y is 1 I am an instance of the class PrintSubClass In the main() method of PrintSubClass, you create a PrintSubClass object and call the printMe() method. Note that PrintSubClass doesn't define this method, so Java looks for it in each of PrintSubClass's superclasses—and finds it, in this case, in PrintClass. because printMe() is still defined in PrintClass, it doesn't print the z instance variable. To call the original method from inside a method definition, use the super keyword to pass the method call up the hierarchy: void myMethod (String a, String b) { // do stuff here super.myMethod(a, b); // maybe do more stuff here }
The super keyword, somewhat like the this keyword, is a placeholder for this class's superclass. You can use it anywhere you can use this, but to refer to the superclass rather than to the current class.