JAVA - MULTITHREADING

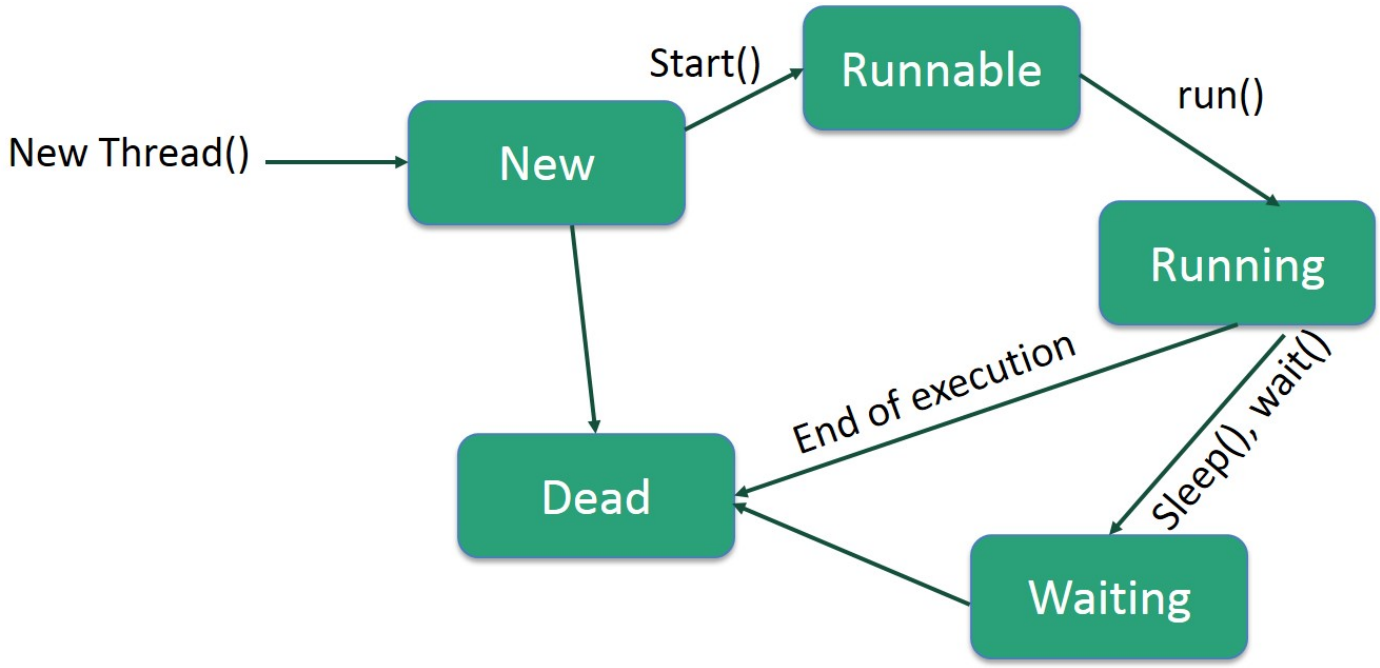
Java is a*multi threaded programming language* which means we can develop multi threaded program using Java. A multi threaded program contains two or more parts that can run concurrently and each part can handle different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

By definition multitasking is when multiple processes share common processing resources such as a CPU. Multi threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

Multi threading enables you to write in a way where multiple activities can proceed concurrently in the same program.

Life Cycle of a Thread:

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. Following diagram shows complete life cycle of a thread.



Above-mentioned stages are explained here:

- **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- **Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting:** Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task.A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated *Dead*:** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread Priorities:

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java thread priorities are in the range between MIN_PRIORITY *aconstantof*1 and MAX_PRIORITY *aconstantof*10. By default, every thread is given priority NORM_PRIORITY *aconstantof*5.

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependent.

Create Thread by Implementing Runnable Interface:

If your class is intended to be executed as a thread then you can achieve this by implementing **Runnable** interface. You will need to follow three basic steps:

Step 1:

As a first step you need to implement a run method provided by **Runnable** interface. This method provides entry point for the thread and you will put you complete business logic inside this method. Following is simple syntax of run method:

```
public void run( )
```

Step 2:

At second step you will instantiate a **Thread** object using the following constructor:

```
Thread(Runnable threadObj, String threadName);
```

Where, *threadObj* is an instance of a class that implements the **Runnable** interface and **threadName** is the name given to the new thread.

Step 3

Once Thread object is created, you can start it by calling **start** method, which executes a call to run method. Following is simple syntax of start method:

```
void start( );
```

Example:

Here is an example that creates a new thread and starts it running:

```
class RunnableDemo implements Runnable {
   private Thread t;
   private String threadName;

   RunnableDemo( String name){
       threadName = name;
       System.out.println("Creating " +  threadName );
   }
   public void run() {
      System.out.println("Running " +  threadName );
      try {
         for(int i = 4; i > 0; i--) {
            System.out.println("Thread: " + threadName + ", " + i);
            // Let the thread sleep for a while.
            Thread.sleep(50);
         }
     } catch (InterruptedException e) {
         System.out.println("Thread " +  threadName + " interrupted.");
     }
     System.out.println("Thread " +  threadName + " exiting.");
```

```
   }

   public void start ()
   {
      System.out.println("Starting " +  threadName );
      if (t == null)
      {
         t = new Thread (this, threadName);
         t.start ();
      }
   }

}

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

      RunnableDemo R1 = new RunnableDemo( "Thread-1");
      R1.start();

      RunnableDemo R2 = new RunnableDemo( "Thread-2");
      R2.start();
   }
}
```

This would produce the following result:

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

Create Thread by Extending Thread Class:

The second way to create a thread is to create a new class that extends **Thread** class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

Step 1

You will need to override **run** method available in Thread class. This method provides entry point for the thread and you will put you complete business logic inside this method. Following is simple syntax of run method:

```
public void run( )
```

Step 2

Once Thread object is created, you can start it by calling **start** method, which executes a call to run method. Following is simple syntax of start method:

```
void start( );
```

Example:

Here is the preceding program rewritten to extend Thread:

```
class ThreadDemo extends Thread {
   private Thread t;
   private String threadName;

   ThreadDemo( String name){
       threadName = name;
       System.out.println("Creating " +  threadName );
   }
   public void run() {
      System.out.println("Running " +  threadName );
      try {
         for(int i = 4; i > 0; i--) {
            System.out.println("Thread: " + threadName + ", " + i);
            // Let the thread sleep for a while.
            Thread.sleep(50);
         }
     } catch (InterruptedException e) {
         System.out.println("Thread " +  threadName + " interrupted.");
     }
     System.out.println("Thread " +  threadName + " exiting.");
   }

   public void start ()
   {
      System.out.println("Starting " +  threadName );
      if (t == null)
      {
         t = new Thread (this, threadName);
         t.start ();
      }
   }

}

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

      ThreadDemo T1 = new ThreadDemo( "Thread-1");
      T1.start();

      ThreadDemo T2 = new ThreadDemo( "Thread-2");
      T2.start();
   }
}
```

This would produce the following result:

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

Thread Methods:

Following is the list of important methods available in the Thread class.

SN	Methods with Description
1	**public void start** Starts the thread in a separate path of execution, then invokes the run method on this Thread object.
2	**public void run** If this Thread object was instantiated using a separate Runnable target, the run method is invoked on that Runnable object.
3	**public final void setName***Stringname* Changes the name of the Thread object. There is also a getName method for retrieving the name.
4	**public final void setPriority***intpriority* Sets the priority of this Thread object. The possible values are between 1 and 10.
5	**public final void setDaemon***booleanon* A parameter of true denotes this Thread as a daemon thread.
6	**public final void join***longmillisec* The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.
7	**public void interrupt** Interrupts this thread, causing it to continue execution if it was blocked for any reason.
8	**public final boolean isAlive** Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

The previous methods are invoked on a particular Thread object. The following methods in the Thread class are static. Invoking one of the static methods performs the operation on the currently running thread.

SN	Methods with Description
1	**public static void yield** Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled.
2	**public static void sleep***longmillisec*

Causes the currently running thread to block for at least the specified number of milliseconds.

3 **public static boolean holdsLock*Objectx***

Returns true if the current thread holds the lock on the given Object.

4 **public static Thread currentThread**

Returns a reference to the currently running thread, which is the thread that invokes this method.

5 **public static void dumpStack**

Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application.

Example:

The following ThreadClassDemo program demonstrates some of these methods of the Thread class. Consider a class **DisplayMessage** which implements **Runnable**:

```
// File Name : DisplayMessage.java
// Create a thread to implement Runnable
public class DisplayMessage implements Runnable
{
   private String message;
   public DisplayMessage(String message)
   {
      this.message = message;
   }
   public void run()
   {
      while(true)
      {
         System.out.println(message);
      }
   }
}
```

Following is another class which extends Thread class:

```
// File Name : GuessANumber.java
// Create a thread to extentd Thread
public class GuessANumber extends Thread
{
   private int number;
   public GuessANumber(int number)
   {
      this.number = number;
   }
   public void run()
   {
      int counter = 0;
      int guess = 0;
      do
      {
          guess = (int) (Math.random() * 100 + 1);
          System.out.println(this.getName()
                      + " guesses " + guess);
          counter++;
      }while(guess != number);
```

```
      System.out.println("** Correct! " + this.getName()
                         + " in " + counter + " guesses.**");
   }
}
```

Following is the main program which makes use of above defined classes:

```
// File Name : ThreadClassDemo.java
public class ThreadClassDemo
{
   public static void main(String [] args)
   {
      Runnable hello = new DisplayMessage("Hello");
      Thread thread1 = new Thread(hello);
      thread1.setDaemon(true);
      thread1.setName("hello");
      System.out.println("Starting hello thread...");
      thread1.start();

      Runnable bye = new DisplayMessage("Goodbye");
      Thread thread2 = new Thread(bye);
      thread2.setPriority(Thread.MIN_PRIORITY);
      thread2.setDaemon(true);
      System.out.println("Starting goodbye thread...");
      thread2.start();

      System.out.println("Starting thread3...");
      Thread thread3 = new GuessANumber(27);
      thread3.start();
      try
      {
         thread3.join();
      }catch(InterruptedException e)
      {
         System.out.println("Thread interrupted.");
      }
      System.out.println("Starting thread4...");
      Thread thread4 = new GuessANumber(75);

   thread4.start();
      System.out.println("main() is ending...");
   }
}
```

This would produce the following result. You can try this example again and again and you would get different result every time.

```
Starting hello thread...
Starting goodbye thread...
Hello
Hello
Hello
Hello
Hello
Hello
Goodbye
Goodbye
Goodbye
Goodbye
Goodbye
.......
```

Major Java Multithreading Concepts:

While doing Multithreading programming in Java, you would need to have the following concepts very handy:

- What is thread synchronization?
- Handling threads inter communication
- Handling thread deadlock
- Major thread operations

JAVA - EXCEPTIONS

An exception *orexceptionalevent* is a problem that arises during the **execution** of a program. When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore these exceptions are to be handled.

An exception can occur for many different reasons, below given are some scenarios where exception occurs.

- A user has entered invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Based on these we have three categories of Exceptions you need to understand them to know how exception handling works in Java,

- **Checked exceptions:** A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the Programmer should take care of *handle* these exceptions.

  For example, if you use **FileReader** class in your program to read data from a file, if the file specified in its constructor doesn't exist, then an *FileNotFoundException* occurs, and compiler prompts the programmer to handle the exception.

```
import java.io.File;
import java.io.FileReader;

public class FilenotFound_Demo {

   public static void main(String args[]){
      File file=new File("E://file.txt");
      FileReader fr = new FileReader(file);
   }

}
```

If you try to compile the above program you will get exceptions as shown below.

```
C:\>javac FilenotFound_Demo.java
FilenotFound_Demo.java:8: error: unreported exception FileNotFoundException; must be
caught or declared to be thrown
      FileReader fr = new FileReader(file);
                      ^
1 error
```

Note: Since the methods **read** and **close** of FileReader class throws IOException, you can observe that compiler notifies to handle IOException, along with FileNotFoundException.

- **Unchecked exceptions:** An Unchecked exception is an exception that occurs at the time of execution, these are also called as Runtime Exceptions, these include programming bugs, such as logic errors or improper use of an API. runtime exceptions are ignored at the time of compilation.

  For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an *ArrayIndexOutOfBoundsExceptionexception* occurs.

```
public class Unchecked_Demo {

   public static void main(String args[]){
      int num[]={1,2,3,4};
      System.out.println(num[5]);
   }

}
```

If you compile and execute the above program you will get exception as shown below.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
 at Exceptions.Unchecked_Demo.main(Unchecked_Demo.java:8)
```

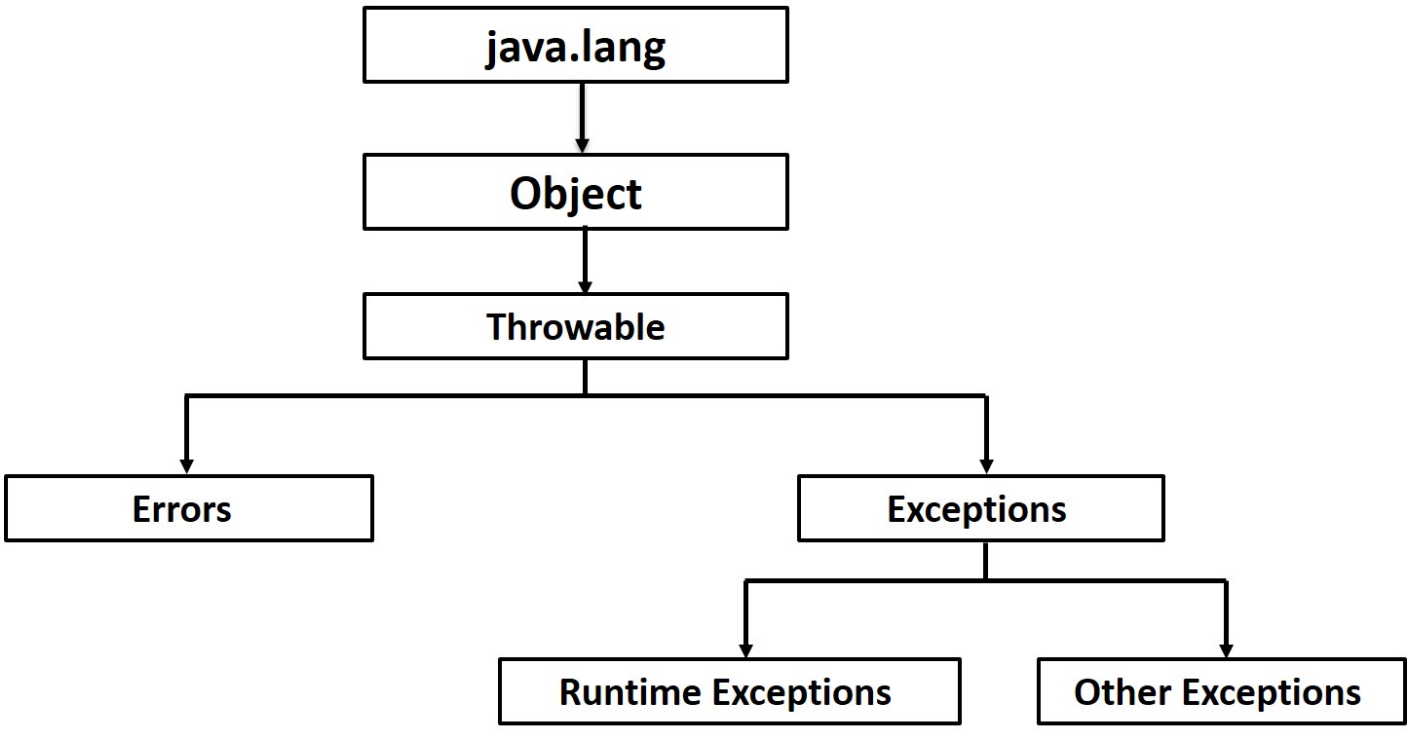
- **Errors:** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Exception Hierarchy:

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Errors are not normally trapped form the Java programs. These conditions normally happen in case of severe failures, which are not handled by the java programs. Errors are generated to indicate errors generated by the runtime environment. Example : JVM is out of Memory. Normally programs cannot recover from errors.

The Exception class has two main subclasses: IOException class and RuntimeException Class.



Here is a list of most common checked and unchecked Java's Built-in Exceptions.

Exceptions Methods:

Following is the list of important medthods available in the Throwable class.

SN	Methods with Description

1 **public String getMessage**

Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.

2 **public Throwable getCause**

Returns the cause of the exception as represented by a Throwable object.

3 **public String toString**

Returns the name of the class concatenated with the result of getMessage

4 **public void printStackTrace**

Prints the result of toString along with the stack trace to System.err, the error output stream.

5 **public StackTraceElement [] getStackTrace**

Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.

6 **public Throwable fillInStackTrace**

Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

Catching Exceptions:

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
   //Protected code
}catch(ExceptionName e1)
{
   //Catch block
}
```

The code which is prone to exceptions is placed in the try block, when an exception occurs, that exception occurred is handled by catch block associated with it. Every try block should be immediately followed either by a class block or finally block.

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block *orblocks* that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Example:

The following is an array is declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
// File Name : ExcepTest.java
import java.io.*;
public class ExcepTest{
```

```
   public static void main(String args[]){
      try{
         int a[] = new int[2];
         System.out.println("Access element three :" + a[3]);
      }catch(ArrayIndexOutOfBoundsException e){
         System.out.println("Exception thrown  :" + e);
      }
      System.out.println("Out of the block");
   }
}
```

This would produce the following result:

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
Out of the block
```

Multiple catch Blocks:

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:

```
try
{
   //Protected code
}catch(ExceptionType1 e1)
{
   //Catch block
}catch(ExceptionType2 e2)
{
   //Catch block
}catch(ExceptionType3 e3)
{
   //Catch block
}
```

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

Example:

Here is code segment showing how to use multiple try/catch statements.

```
try
{
   file = new FileInputStream(fileName);
   x = (byte) file.read();
}catch(IOException i)
{
   i.printStackTrace();
   return -1;
}catch(FileNotFoundException f) //Not valid!
{
   f.printStackTrace();
   return -1;
}
```

Catching multiple type of exceptions

Since Java 7 you can handle more than one exceptions using a single catch block, this feature simplifies the code. Below given is the syntax of writing

```
catch (IOException|FileNotFoundException ex) {
   logger.log(ex);
   throw ex;
```

The throws/throw Keywords:

If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword.

Try to understand the difference between throws and throw keywords, *throws* is used to postpone the handling of a checked exception and *throw* is used to invoke an exception explicitly.

The following method declares that it throws a RemoteException:

```
import java.io.*;
public class className
{
   public void deposit(double amount) throws RemoteException
   {
      // Method implementation
      throw new RemoteException();
   }
   //Remainder of class definition
}
```

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException:

```
import java.io.*;
public class className
{
   public void withdraw(double amount) throws RemoteException,
                               InsufficientFundsException
   {
       // Method implementation
   }
   //Remainder of class definition
}
```

The finally block

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax:

```
try
{
   //Protected code
}catch(ExceptionType1 e1)
{
   //Catch block
}catch(ExceptionType2 e2)
{
   //Catch block
}catch(ExceptionType3 e3)
{
   //Catch block
}finally
```

```
{
   //The finally block always executes.
}
```

Example:

```
public class ExcepTest{

   public static void main(String args[]){
      int a[] = new int[2];
      try{
         System.out.println("Access element three :" + a[3]);
      }catch(ArrayIndexOutOfBoundsException e){
         System.out.println("Exception thrown  :" + e);
      }
      finally{
         a[0] = 6;
         System.out.println("First element value: " +a[0]);
         System.out.println("The finally statement is executed");
      }
   }
}
```

This would produce the following result:

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
The finally statement is executed
```

Note the following:

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses when ever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

The try-with-resources

Generally when we use any resources like streams, connections etc.. we have to close them explicitly using finally block. In the program given below we are reading data from a file using **FileReader** and we are closing it using finally block.

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class ReadData_Demo {

   public static void main(String args[]){
      FileReader fr=null;
      try{
         File file=new File("file.txt");
         fr = new FileReader(file);  char [] a = new char[50];
         fr.read(a); // reads the content to the array
         for(char c : a)
         System.out.print(c); //prints the characters one by one
      }catch(IOException e){
          e.printStackTrace();
       }
       finally{
          try{
              fr.close();
          }catch(IOException ex){
               ex.printStackTrace();
```

```
            }
        }
    }

}
```

try-with-resources, also referred as **automatic resource management**. is a new exception handling mechanism that was introduced in Java7, which automatically closes the resources used within the try catch block.

To use this statement you simply need to declare the required resources within the parenthesis, the created resource will be closed automatically at the end of the block, below given is the syntax of try-with-resources statement.

```
try(FileReader fr=new FileReader("file path"))
   {
   //use the resource
   }catch(){
      //body of catch
    }
   }
```

Below given is the program that reads the data in a file using try-with-resources statement.

```
import java.io.FileReader;
import java.io.IOException;

public class Try_withDemo {

   public static void main(String args[]){

      try(FileReader fr=new FileReader("E://file.txt")){
         char [] a = new char[50];
         fr.read(a); // reads the contentto the array
         for(char c : a)
         System.out.print(c); //prints the characters one by one
      }catch(IOException e){
          e.printStackTrace();
       }
   }
}
```

Following points are to be kept in mind while working with try-with resources statement.

- To use a class with try-with-resources statement it should implement **AutoCloseable** interface and the **close** method of it gets invoked automatically at runtime.
- You can declare more than one class in try-with-resources statement.
- while you declare multiple classes in the try block of try-with-resources statement these classes are closed in reverse order.
- Except the deceleration of resources within the parenthesis every thing is same as normal try/catch block of a try block.
- The resource declared in try gets instantiated just before the start of the try-block.
- The resource declared at the try block is implicitly declared as final.

User-defined Exceptions:

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes:

- All exceptions must be a child of Throwable.
- If you want to write a checked exception that is automatically enforced by the Handle or

Declare Rule, you need to extend the Exception class.

- If you want to write a runtime exception, you need to extend the RuntimeException class.

We can define our own Exception class as below:

```
class MyException extends Exception{
}
```

You just need to extend the predefined **Exception** class to create your own Exception. These are considered to be checked exceptions. The following **InsufficientFundsException** class is a user-defined exception that extends the Exception class, making it a checked exception. An exception class is like any other class, containing useful fields and methods.

Example:

```
// File Name InsufficientFundsException.java
import java.io.*;

public class InsufficientFundsException extends Exception
{
   private double amount;
   public InsufficientFundsException(double amount)
   {
      this.amount = amount;
   }
   public double getAmount()
   {
      return amount;
   }
}
```

To demonstrate using our user-defined exception, the following CheckingAccount class contains a withdraw method that throws an InsufficientFundsException.

```
// File Name CheckingAccount.java
import java.io.*;

public class CheckingAccount
{
   private double balance;
   private int number;

   public CheckingAccount(int number)
   {
      this.number = number;
   }

   public void deposit(double amount)
   {
      balance += amount;
   }

   public void withdraw(double amount) throws InsufficientFundsException
   {
      if(amount <= balance)
      {
         balance -= amount;
      }
      else
      {
         double needs = amount - balance;
         throw new InsufficientFundsException(needs);
      }
   }

   public double getBalance()
```

```
   {
      return balance;
   }

   public int getNumber()
   {
      return number;
   }
}
```

The following BankDemo program demonstrates invoking the deposit and withdraw methods of CheckingAccount.

```
// File Name BankDemo.java
public class BankDemo
{
   public static void main(String [] args)
   {
      CheckingAccount c = new CheckingAccount(101);
      System.out.println("Depositing $500...");
      c.deposit(500.00);

      try
      {
         System.out.println("\nWithdrawing $100...");
         c.withdraw(100.00);
         System.out.println("\nWithdrawing $600...");
         c.withdraw(600.00);
      }catch(InsufficientFundsException e)
      {
         System.out.println("Sorry, but you are short $" + e.getAmount());
         e.printStackTrace();
      }
    }
}
```

Compile all the above three files and run BankDemo, this would produce the following result:

```
Depositing $500...

Withdrawing $100...

Withdrawing $600...
Sorry, but you are short $200.0
InsufficientFundsException
        at CheckingAccount.withdraw(CheckingAccount.java:25)
        at BankDemo.main(BankDemo.java:13)
```

Common Exceptions:

In Java, it is possible to define two catergories of Exceptions and Errors.

- **JVM Exceptions:** - These are exceptions/errors that are exclusively or logically thrown by the JVM. Examples : NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException,
- **Programmatic exceptions:** - These exceptions are thrown explicitly by the application or the API programmers Examples: IllegalArgumentException, IllegalStateException.

Java Applets

Applets

- An applet is a Panel that allows interaction with a Java program
- Typically embedded in a Web page and can be run from a browser
- You need special HTML in the Web page to tell the browser about the applet
- For security reasons applets run in a sandbox
  - Sandbox is a
    - Byte-code verifier
    - Class loader
    - Security manager
  - Only the correct classes are loaded
  - The classes are in the correct format
  - Un-trusted classes
    - Will not execute dangerous instructions
    - Are not allowed to access protected system resources

Applet Support

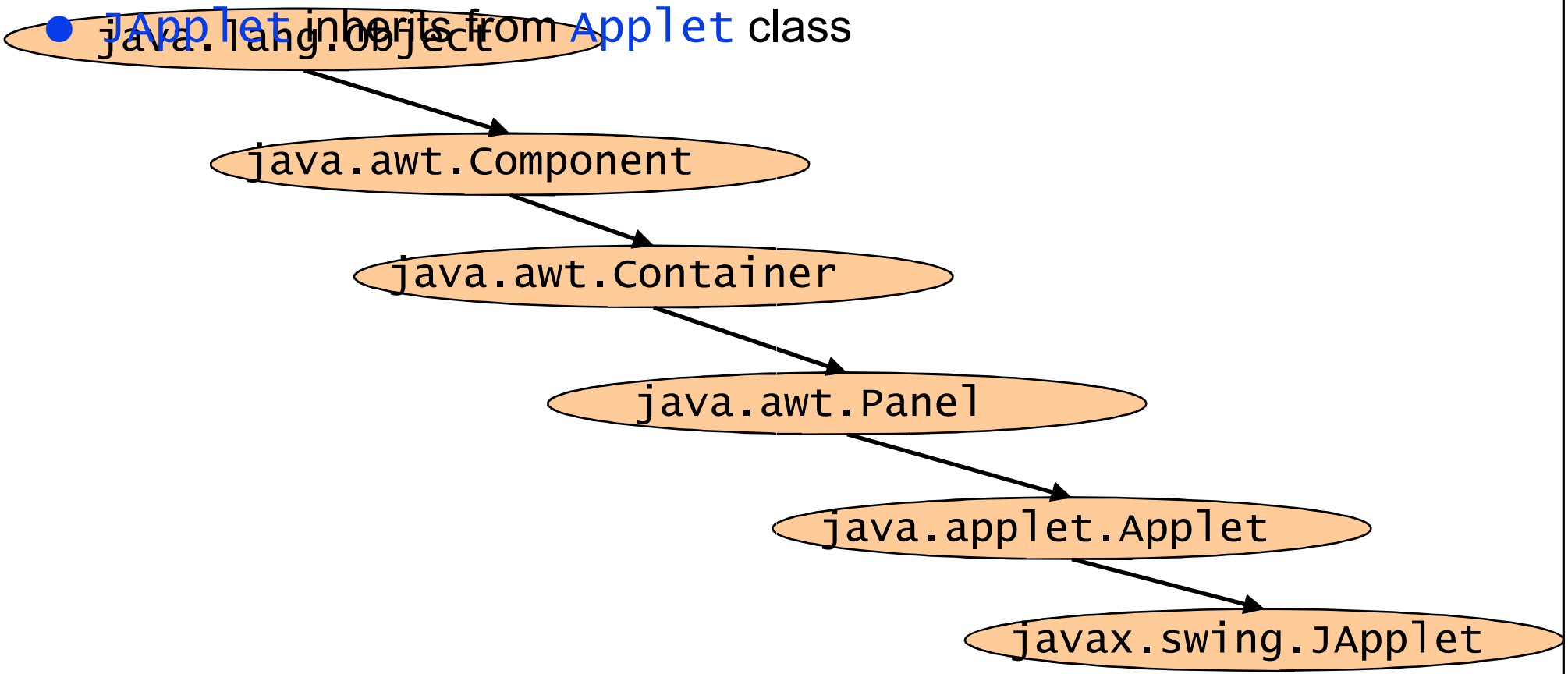
- Java 1.4 and above are supported from the most modern browsers if these browsers have the appropriate plug-in
- Basic browsers that support applets
  - Internet Explorer
  - Netscape Navigator (sometimes)
- However, the best support isn't a browser, but the standalone program `appletviewer`
- In general you should try to write applets that can be run with any browser

Notion of Applets in Java

- You can write an applet by extending the class `Applet`
- `Applet` class
  - Contains code that works with a browser to create a display window
  - Is just a class like any other
    - You can even use it in applications if you want
- When you write an applet you are only writing part of a program
- The browser supplies the `main` method
- NOTE: If you use Swing components in your applet you must use the `JApplet` class
  - `JApplet` extends the class `Applet`

The genealogy of the `Applet` class

- `Applet` inherits `awt Component` class and `awt Container` class
- `JApplet` inherits from `Applet` class



The Simplest Possible Applet

TrivialApplet.java

```
import java.applet.Applet;
public class TrivialApplet extends Applet { }
```

TrivialApplet.html

```
<applet
    code="TrivialApplet.class"
    width=150  height=100>
</applet>
```

The Simplest Reasonable Applet

```
import java.awt.*;
import java.applet.Applet;

public class HelloWorld extends Applet {
    public void paint( Graphics g ) {
        g.drawString( "Hello World!", 30, 30 );
    }
}
```

Applet methods

- Basic methods
  - `public void init ()`
  - `public void start ()`
  - `public void stop ()`
  - `public void destroy ()`
- Other Supplementary methods
  - `public void showStatus(String)`
  - `public String getParameter(String)`

How a Java Applet works?

- You write an applet by extending the class `Applet`
- `Applet` class defines methods as
  - `init( )`
  - `start( )`
  - `stop( )`
  - `destroy( )`
  - and some others…
- These methods do not do anything
  - They are stubs
- You make the applet do something by overriding these methods
- You don't need to override all these methods
  - Just the ones you care about

Method `init( )`

- This is the first of your methods to be executed
- It is automatically called by the system when the JVM launches the applet for the first time
- It is only executed once
- It is the best place to
  - Initialize variables
  - Define the GUI Components
    - E.g. buttons, text fields, scrollbars, etc.
  - Lay the components out
  - Add listeners to these components
  - Pick up any HTML parameters
- Almost every applet you ever write will have an `init( )` method

Method `start( )`

- Not usually needed
- It is automatically called after the JVM calls the `init( )` method
- Also called whenever a user returns to the HTML page containing the applet after having gone to other pages
  - i.e. each time the page is loaded and restarted
- Can be called repeatedly
  - Common place to restart a thread
    - E.g. resuming an animation
- Used mostly in conjunction with `stop( )`

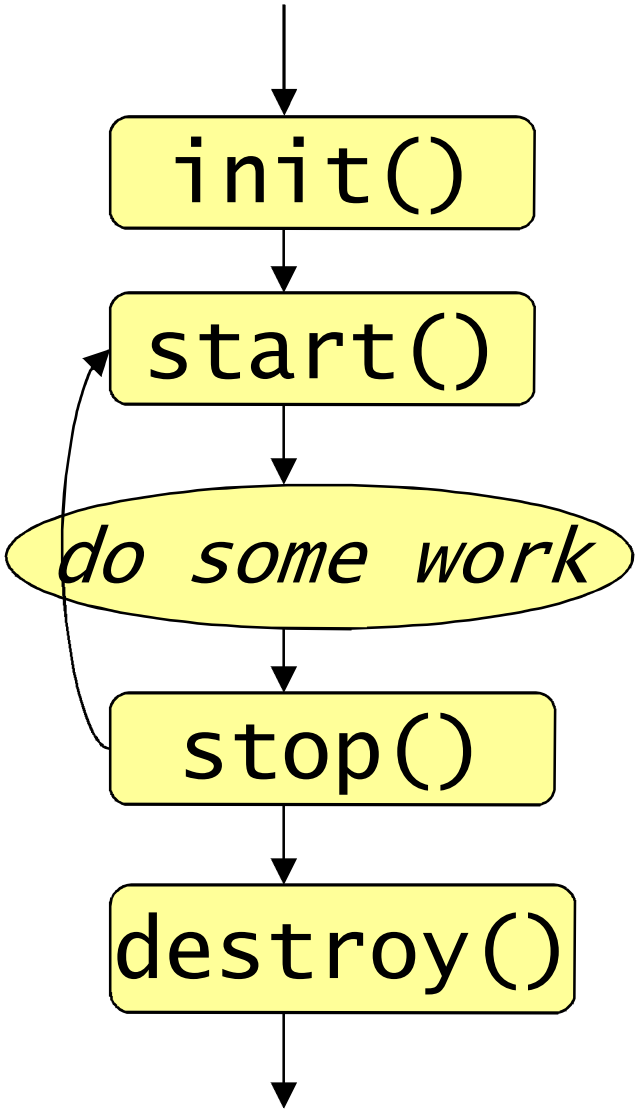
Method `stop( )`

- Not usually needed
- It is automatically called when the user moves off the page on which the applet sits
- Can be called repeatedly in the same applet
- Called just before `destroy( )`
- Gives a chance to stop time-consuming activity from slowing down the system when the user is not paying attention to the applet
- Should not be called directly
- Used mostly in conjunction with `start( )`

Method `destroy( )`

- Almost never needed
- Called after `stop( )`
- The JVM guarantees to call this method when the browser shuts down normally
- Use to explicitly release system resources
  - E.g. threads
- System resources are usually released automatically
- Commonly used for reclaiming non-memory-dependent resources

Order of Methods' Calls

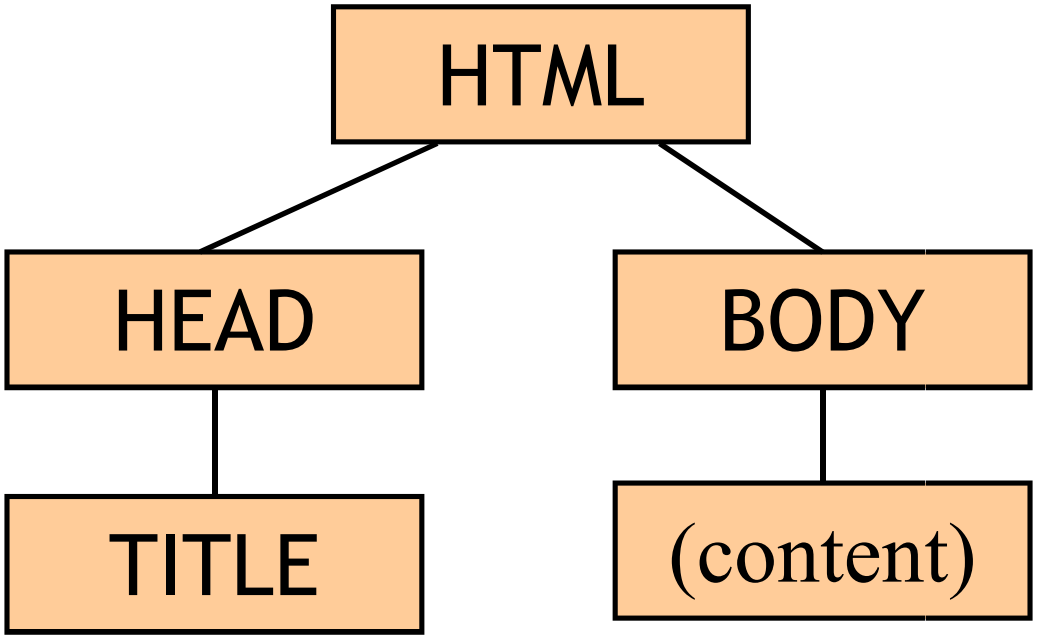


- `init( )` and `destroy( )` are only called once each
- `start( )` and `stop( )` are called whenever the browser enters and leaves the page
- do some work is code called by the listeners that may exist in the applet

Other Useful Applet Methods

- `System.out.println(String)`
  - Works from `appletviewer`, not from browsers
  - Automatically opens an output window
- `showStatus(String)`
  - Displays the String in the applet's status line
  - Each call overwrites the previous call
  - You have to allow time to read the line!

Structure of an HTML page



- Most HTML tags are containers
  - Not Java `Containers` !!!!
- A container is `<tag>` to `</tag>`

Invocation of Applets in HTML Code

```
<html>
  <head>
    <title> Hi World Applet </title>
  </head>

  <body>
    <applet code="HiWorld.class"
     width=300 height=200>
        <param name="arraysize" value="10">
    </applet>
  </body>
</html>
```

Not a container

Method getParameter(String)

- This method is called for the retrieval of the value of a parameter with specific name which is set inside the HTML code of the applet
  - This name is the only argument of the method

- E.g. let the HTML code for the applet

```
<applet code="HiWorld.class" width=300 height=200>
      <param name="arraysize" value="10">
</applet>
```

- A possible method call could be

```
String s = this.getParameter("arraysize");

try { size = Integer.parseInt (s) }
catch (NumberFormatException e) {…}
```

An Applet that adds two floating-point numbers

- Class and attributes' declarations

```
import java.awt.Graphics;   // import Graphics class
import javax.swing.*;  // import swing package

public class AdditionApplet extends JApplet {

// sum of the values entered by the user
double sum;
```

An Applet that adds two floating-point numbers

- Method `init( )`

```
public void init() {
      String firstNumber, secondNumber;
      double number1, number2;

      // read in first number from user
      firstNumber = JOptionPane.showInputDialog(
                  "Enter first floating-point value" );

      // read in second number from user
      secondNumber = JOptionPane.showInputDialog(
                  "Enter second floating-point value" );
```

An Applet that adds two floating-point numbers

- Method `init( )` cont.(1)

```
    // convert numbers from type String to type double
    number1 = Double.parseDouble( firstNumber );
    number2 = Double.parseDouble( secondNumber );

    // add the numbers
    sum = number1 + number2;
} //end of init
```

An Applet that adds two floating-point numbers

- Method `paint(Graphics)`

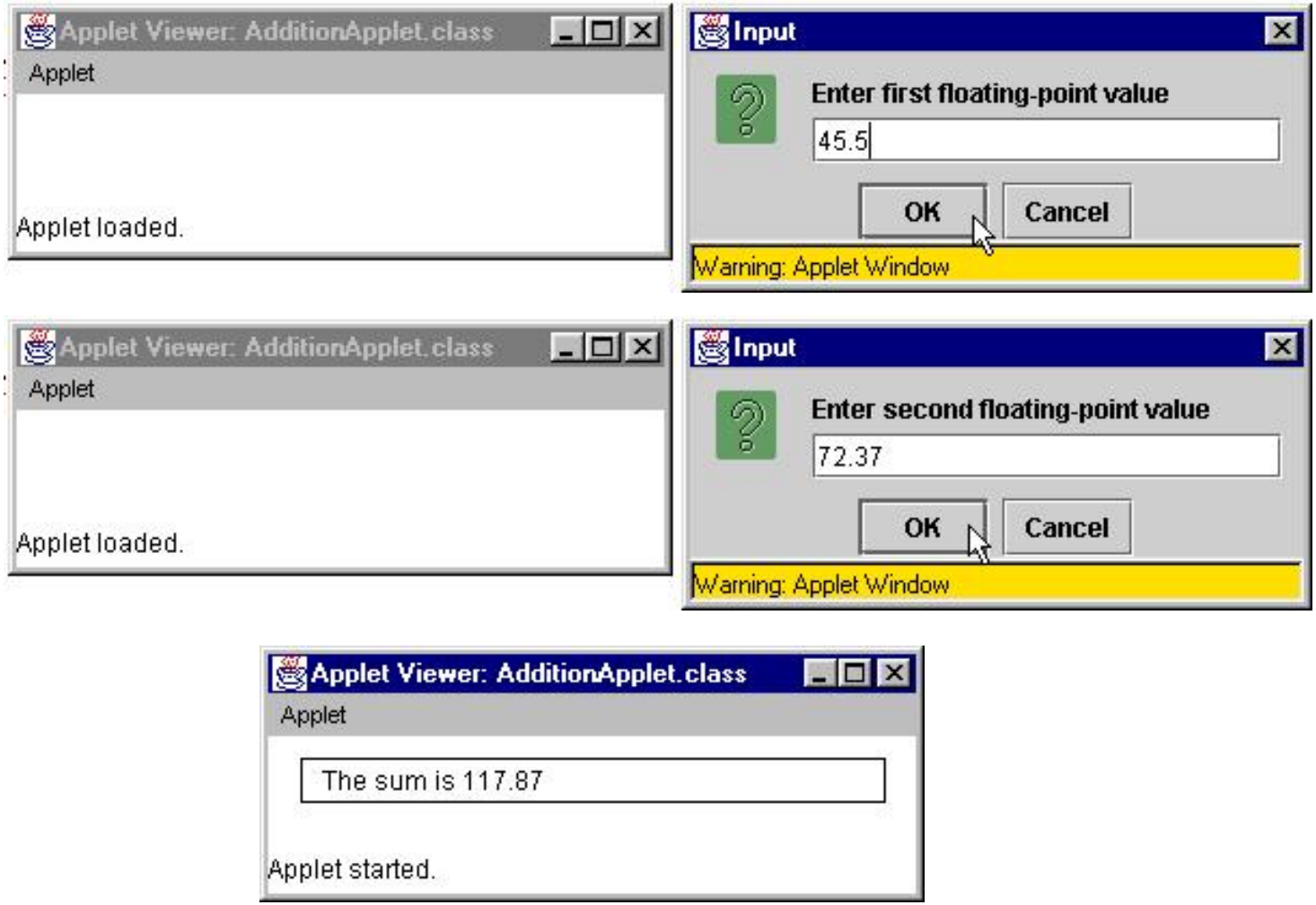
```
public void paint( Graphics g ){

      // draw the results with g.drawString
      g.drawRect( 15, 10, 270, 20 );
      g.drawString( "The sum is " + sum, 25, 25 );
} //end of paint
} //end of AdditionApplet class
```

- HTML source for the applet

```
<html>
<applet code=AdditionApplet.class width=300 height=50>
</applet>
</html>
```

An Applet that adds two floating-point numbers

- Output



A Digital Clock Applet

- Class and attributes' declarations

```
import java.awt.*;
import java.util.Calendar;
import java.applet.Applet;

public class DigitalClock extends Applet
                          implements Runnable {

protected Thread clockThread;
protected Font font;
protected Color color;
```

A Digital Clock Applet

- Initialization of fields in method `init( )`

```
public void init() {
        clockThread = null;
        font = new Font("Monospaced", Font.BOLD, 48);
        color = Color.green;
} //end of init
```

- Method `start( )`

```
public void start() {
     if (clockThread == null) {
          clockThread = new Thread(this);
          clockThread.start();
     }
} //end of start
```

calls the `run()` method