ADVANCED JAVA PROGRAMMING

UNIT II: Exception handing: Types of Exceptions - Uncaught Exception - try, catch, throw, throws, finally - Built-in Exception, user defined exception. Multithreading: The Java thread model – Main thread - Creating a thread – Thread priorities - Synchronization. I/O basics – Stream Classes – Predefined streams – Reading/Writing console input/output-Applet class: Applet basics – Applet architecture – Applet display method – HTML Applet Tag.

Text Book:

Herbert Schildt, “The Complete Reference Java”, 7th Edition Tata McGraw-Hill Pub. Company Ltd.

Prepared by : B.Loganathan

Exception handing

- An *exception* is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error.
- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error.
- Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.

- ***Try and catch :***
- If an exception occurs within the **try** block, it is thrown. Our code can catch this exception (using **catch**) and handle it in some rational manner.
- Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.
- This is the general form of an exception-handling block:
- try {
- // block of code to monitor for errors
- }

- catch (*ExceptionType1 exOb*) {
- // exception handler for *ExceptionType1*
- }
- catch (*ExceptionType2 exOb*) {
- // exception handler for *ExceptionType2*
- }
- // ...
- finally {
- /* block of code to be executed after try block ends */
- }
- **Here, *ExceptionType* is the type of exception that has occurred.**

Types of Exceptions :

- All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy.
- Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches.
- There is an important subclass of **Exception**, called **RuntimeException**, such as *division by zero* and *invalid array indexing.*
- The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances. *Stack overflow* is an example of such an error.

Uncaught Exception

- **This small program includes an expression that intentionally causes a divide-by-zero error:**
- class Exc0 {
- public static void main(String args[]) {
- int d = 0;
- int a = 42 / d;
- }
- }
- **When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception.**
- **This causes the exception during Exc0 execution .**

- The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.
- Here is the exception generated when this example is executed:
- java.lang.ArithmeticException: / by zero at Exc0.main(Exc0.java:4)

- Notice here the class name, **Exc0**; the method name, **main**; the filename, **Exc0.java**; and the line number, **4**, are all included in the simple stack trace.

Using try, catch, throw, throws, finally

- To guard against and handle a run-time error, simply enclose the code that we want to monitor inside a **try** block. Immediately following the **try** block, include a **catch** clause that specifies the exception type that we wish to catch.
- The following program includes a **try** block and a **catch** clause that processes the **ArithmeticException** generated by the division-by-zero error:

```
class Exc2 {
public static void main(String args[]) {
int d, a;
try { // monitor a block of code.
d = 0;
a = 42 / d;
System.out.println("This will not be printed.");
}
catch (ArithmeticException e) {
// catch divide-by-zero error
System.out.println("Division by zero.");
}  /* end catch */
System.out.println("After catch statement.");
}  /* end try */
}
```

- **This program generates the following output:**
- Division by zero.
- After catch statement.
- **Notice that the call to println( ) inside the try block is never executed.**
- **Once an exception is thrown, program control transfers out of the try block into the catch block. Put differently, catch is not "called," so execution never "returns" to the try block from a catch.**
- **Thus, the line "This will not be printed." is not displayed.**

Throw:

- The general form of **throw** is shown here:
- throw *ThrowableInstance*;
- Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**.
- There are two ways we can obtain a **Throwable** object: using a parameter in a **catch** clause, or creating one with the **new** operator.
- Here is a 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);
}
}
}
```

- **The exception is then rethrown. Here is the resulting output:**
- Caught inside demoproc.
- Recaught: java.lang.NullPointerException: demo

- **Here, new is used to construct an instance of NullPointerException. Many of Java's built-in run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter.**
- **When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to print( ) or println( ).**

- ***Throws:***
- 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.
- 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.

```
// This is throws example progam
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
- ***Finally:***
- **When exceptions are thrown, execution in a method takes an abrupt, nonlinear path that alters the normal flow through the method.**
- **For example, if a method opens a file upon entry and closes it upon exit, then we will not want the code that closes the file to be bypassed by the exception-handling mechanism. The finally keyword is designed to address this contingency.**

- **finally** creates a block of code that will be executed after a **try/catch** block has completed and before the code following the **try/catch** block. The **finally** block will execute whether or not an exception is thrown.
- If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception.
- Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns.

- This finally can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning.
- The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause.
- Here is an example program that shows three methods that exit in various ways, none without executing their **finally** clauses:

```
// Demonstrate finally.
class FinallyDemo {
// Through an exception out of the method.
static void procA() {
try {
System.out.println("inside procA");
throw new RuntimeException("demo");
}
finally {
System.out.println("procA's finally");
}
}
```

- // Return from within a try block.
- static void procB() {
- try {
- System.out.println("inside procB");
- return;
- } finally {
- System.out.println("procB's finally");
- }
- }
- // Execute a try block normally.
- static void procC() {
- try {
- System.out.println("inside procC");
- } finally {

- System.out.println("procC's finally");
- }
- }
- public static void main(String args[]) {
- try {
- procA();
- } catch (Exception e) {
- System.out.println("Exception caught");
- }
- procB();
- procC();
- }

- In this example, **procA( )** prematurely breaks out of the **try** by throwing an exception. The **finally** clause is executed on the way out. **procB( )**’s **try** statement is exited via a **return** statement.
- The **finally** clause is executed before **procB( )** returns. In **procC( )**, the **try** statement executes normally, without error. However, the **finally** block is still executed.
- Here is the output generated by the preceding program:
- inside procA
- procA’s finally
- Exception caught
- inside procB
- procB’s finally
- inside procC
- procC’s finally

- ***Built-in Exception:***
- Inside the standard package **java.lang**, Java defines several exception classes.
- The *unchecked exceptions* defined in **java.lang** are listed in the first Table.
- The second Table lists those exceptions defined by **java.lang** that must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself. These are called *checked exceptions.*

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBounds Exception	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid Cast.
EnumConstantNotPrese ntException	An attempt is made to use an undefined enumeration value.
IllegalArgumentExceptio n	Illegal argument used to invoke a method.
IllegalMonitorStateExce ption	Illegal monitor operation, such as waiting on an unlocked thread.

IllegalStateException	**Environment or application is in incorrect state.**
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.

TABLE 2 :Java's Checked Exceptions

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

User Defined Exception

- Although Java’s built-in exceptions handle most common errors, we will probably want to create our own exception types to handle situations specific to our applications.
- This is quite easy to do: just define a subclass of **Exception** (which is, of course, a subclass of **Throwable**).
- The **Exception** class does not define any methods of its own. It does, of course, inherit those methods provided by **Throwable**.
- They are shown in the following Table as the Methods defined by Throwable.

Method	Description
Throwable fillInStackTrace( )	Returns a Throwable object that contains a completed stack trace. This object can be rethrown.
Throwable getCause( )	Returns the exception that underlies the current exception. If there is no underlying exception, null is returned.
String getLocalizedMessage( )	Returns a localized description of the exception.
String getMessage( )	Returns a description of the exception.
StackTraceElement[ ] getStackTrace( )	Returns an array that contains the stack trace, one element at a time, as an array of StackTraceElement.
void printStackTrace( )	Displays the stack trace.
String toString( )	Returns a String object containing a description of the exception. This method is called by println( ) when outputting a Throwable object.

- **Exception defines two are shown here:**
- Exception( )
- Exception(String *msg*)
- **The first form creates an exception that has no description. The second form lets specify a description of the exception.**
- /* This program creates a custom exception type. */
- class MyException extends Exception {
- private int detail;
- MyException(int a) {
- detail = a;
- }

```
public String toString() {
return "MyException[" + detail + "]";
}
}
class ExceptionDemo {
static void compute(int a) throws MyException {
System.out.println("Called compute(" + a + ")");
if(a > 10)
throw new MyException(a);
System.out.println("Normal exit");
}
public static void main(String args[]) {
try {
compute(1);
compute(20);
} catch (MyException e) {
System.out.println("Caught " + e);
}
}
}
```

- This example defines a subclass of **Exception** called **MyException.** This subclass is quite simple: it has only a constructor plus an overloaded **toString( )** method that displays the value of the exception.
- The **ExceptionDemo** class defines a method named **compute( )** that throws a **MyException** object. The exception is thrown when **compute( )**'s integer parameter is greater than 10.
- The **main( )** method sets up an exception handler for **MyException**, then calls **compute()** with a legal value (less than 10) and an illegal one to show both paths through the code.

- **Here is the program result:**
- Called compute(1)
- Normal exit
- Called compute(20)
- Caught MyException[20]

- ***Multithreading:***
- **Java provides built-in support for *multithreaded programming.* A multithreaded program contains two or more parts that can run concurrently.**

- Each part of a program is called a *thread,* and each thread defines a separate path of execution.
- There are two distinct types of multitasking: *process based* and *thread-based*.
- A *process* is, in essence, a program that is executing. Thus, *process-based* multitasking is the feature that allows our computer to run two or more programs concurrently.
- For example, process-based multitasking enables us to run the Java compiler at the same time that we are using a text editor.

- 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.
- For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.
- Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is also costly.

- Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is low cost.
- ***<u>The Java Thread Model:</u>***
- Single-threaded systems use an approach called an *event loop* with *polling.* In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next.

- This wastes CPU time. It can also result in one part of a program dominating the system and preventing any other events from being processed.
- In general, in a singled-threaded environment, when a thread *blocks* (that is, suspends execution) because it is waiting for some resource, the entire program stops running.
- The benefit of Java’s *multithreading* is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of our program.
- When a thread blocks in a Java program, only the single thread that is blocked pauses. All the threads continue to run.

- Threads exist in several states. A thread can be *running.* It can be *ready to run* as soon as it gets CPU time. A running thread can be *suspended,* which temporarily suspends its activity.
- A suspended thread can then be *resumed,* allowing it to pick up where it left off. A thread can be *blocked* when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately.
- <u>Thread Priorities:</u>
- Thread priorities are integers that specify the relative priority of one thread to another.

- A thread's priority is used to decide when to switch from one running thread to the next. This is called a *context switch.* The rules that determine when a context switch takes place are simple:
- *A thread can voluntarily relinquish control:*
- This is done by explicitly yielding, sleeping, or blocking on pending I/O. All other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
- *A thread can be preempted by a higher-priority thread:*
- In this case, a lower-priority thread that does not yield the processor is simply preempted by a higher-priority thread. This is called *preemptive multitasking.*

- ***Synchronization:***
- **Because multithreading introduces an asynchronous behavior to our programs, there must be a way to enforce synchronicity when we need it.**
- **We must prevent one thread from writing data while another thread is in the middle of reading it.**
- **Java implements an elegant twist on an age-old model of interposes synchronization: the *monitor*. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.**

- Each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object.
- ***Messaging :***
- Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have. Java's messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.

- ***Thread Class and the Runnable Interface*** :

- Java’s multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**.
- To create a new thread, our program will either extend **Thread** or implement the **Runnable** interface.
- The **Thread** class defines several methods that help manage threads. The ones that will be used are shown here:

Method	Meaning
getName	Obtain a thread’s name.
getPriority	Obtain a thread’s priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

The Main Thread

- When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of our program, because it is the one that is executed when our program begins.
- The main thread is important for two reasons:

• It is the thread from which other “child” threads will be spawned.

• Often, it must be the last thread to finish execution because it performs various shutdown actions.

- Although the main thread is created automatically when our program is started, it can be controlled through a **Thread** object.
- Its general form is shown here:
- static Thread currentThread( )
- This method returns a reference to the thread in which it is called. The following example program shows the main thread.
- In this program, a reference to the current thread (the main thread) is obtained by calling **currentThread( )**, and this reference is stored in the local variable **t**.
- Next, the program displays information about the thread.

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

- **The program then calls setName( ) to change the internal name of the thread.**
- **Here is the output generated by this program:**

- Current thread: Thread[main,5,main]
- After name change: Thread[My Thread,5,main]
- 5
- 4
- 3
- 2
- 1
- **Notice the output produced when t is used as an argument to println( ).**

I/O basics

- Java does provide strong, flexible support for I/O as it relates to files and networks.
- Java programs perform I/O through streams. A *stream* is an abstraction that either produces or consumes information.
- An input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk file, or a network connection.
- Java implements streams within class hierarchies defined in the **java.io** package.

- ***Stream Classes :***
- Java defines two types of streams: byte and character.
- *Byte streams* provide a convenient means for handling input and output of bytes. Byte streams are used, when reading or writing binary data.
- *Character streams* provide a convenient means for handling input and output of characters.
- At the lowest level, all I/O is still byte-oriented. The character-based streams simply provide a convenient and efficient means for handling characters.

- ***The Byte Stream Classes :***
- Byte streams are defined by using two class hierarchies. At the top are two abstract classes: **InputStream** and **OutputStream**.
- The abstract classes **InputStream** and **OutputStream** define several key methods that the other stream classes implement.
- Two of the most important are **read( )** and **write( )**, which, respectively, read and write bytes of data.
- The byte stream classes are shown in the following Table:

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered Output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that write to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An Output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that write to a file
FilterInputStream	Implements InputStream
FilterOutputStream	Implements OutputStream
InputStream	Abstract class that describes stream input

Stream Class	Meaning
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains print() and println( )
PushbackInputStream	Input stream that supports one-byte
RandomAccessFile	Supports random access file I/O
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

Character Stream I/O Classes

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer

Stream Class	Meaning
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates Characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains print( ) and println( )
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
Writer	Abstract class that describes character stream output
StringReader	Input stream that reads from a string
StringWriter	Output stream that write to a string

The Predefined Streams

- All Java programs automatically import the **java.lang** package. This package defines a class called **System**, which encapsulates several aspects of the run-time environment.
- **System** also contains three predefined stream variables: **in**, **out**, and **err**. These fields are declared as **public**, **static**, and **final** within **System**. This means that they can be used by any other part of our program and without reference to a specific **System** object.
- **System.out** refers to the standard output stream. By default, this is the console.

- **System.in** refers to standard input, which is the keyboard by default. **System.err** refers to the standard error stream, which also is the console by default.
- **System.in** is an object of type **InputStream**; **System.out** and **System.err** are objects of type **PrintStream**.
- ***<u>Reading Console Input:</u>***
- The preferred method of reading console input is to use a character-oriented stream, which makes our program easier to internationalize and maintain.

- In Java, console input is accomplished by reading from **System.in**. To obtain a character based stream that is attached to the console, in a **BufferedReader** object.
- **BufferedReader** supports a buffered input stream. Its most commonly used constructor is shown here:
- BufferedReader(Reader *inputReader*)
- Here, *inputReader* is the stream that is linked to the instance of **BufferedReader** that is being created.
- **Reader** is an abstract class. One of its concrete subclasses is **InputStreamReader**, which converts bytes to characters.

- An **InputStreamReader** object that is linked to **System.in**, use the following constructor:
- InputStreamReader(InputStream *inputStream*)
- Because **System.in** refers to an object of type **InputStream**, it can be used for *inputStream.*
- The following line of code creates a **BufferedReader** that is connected to the keyboard:
- BufferedReader br = new BufferedReader(new
- InputStreamReader(System.in));
- After this statement executes, **br** is a character-based stream that is linked to the console through **System.in**.

- ***Reading Characters:***
- To read a character from a **BufferedReader**, use **read( )**. The version of **read( )** that we will be using is :
- int read( ) throws IOException
- Each time that **read( )** is called, it reads a character from the input stream and returns it as an integer value.
- The following program demonstrates **read( )** by reading characters from the console until the user types a "q."

- /* Use a BufferedReader to read characters from the console. */
- import java.io.*;
- class BRRead {
- public static void main(String args[])
- throws IOException
- {
- char c;
- BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
- System.out.println("Enter characters, 'q' to quit.");

- // read characters
- do {
- c = (char) br.read();
- System.out.println(c);
- } while(c != 'q');
- }
- }
- **Here is a Output of the program run:**
- Enter characters, 'q' to quit.
- 123abcq
- 1
- 2
- 3
- a
- b
- c
- q

- ***Reading Strings :***
- **To read a string from the keyboard, use the version of readLine( ) that is a member of the BufferedReader class. Its general form is shown here:**
- String readLine( ) throws IOException
- **It returns a String object.**
- **The following part of program reads and displays lines of text until we enter the word "stop":**
- do {
- str = br.readLine();
- System.out.println(str);
- } while(!str.equals("stop"));

Writing console output

- Console output is most easily accomplished with **print( )** and **println( )**. These methods are defined by the class **PrintStream** (which is the type of object referenced by **System.out**).
- **write( )** can be used to write to the console. The simplest form of **write( )** defined by **PrintStream** is shown here:
- void write(int *byteval*)
- This method writes to the stream the byte specified by *byteval.*

- **Here is a short example that uses write( ) to output the character "A" followed by a newline to the screen:**
- // Demonstrate System.out.write().
- class WriteDemo {
- public static void main(String args[]) {
- int b;
- b = 'A';
- System.out.write(b);
- System.out.write('\n');
- }
- }

The PrintWriter Class :

- **PrintWriter** is one of the character-based classes. Using a character-based class for console output makes it easier to internationalize our program.
- **PrintWriter** defines several constructors. The one we will use is shown here:
- PrintWriter(OutputStream *outputstream*, boolean *flushOnNewline*)
- Here, *outputstream* is an object of type **OutputStream**. If *flushOnNewline* is **true**, flushing automatically takes place. If **false**, flushing is not automatic.

- **The following application illustrates using a PrintWriter to handle console output:**
- // Demonstrate PrintWriter
- import java.io.*;
- public class PrintWriterDemo {
- public static void main(String args[]) {
- PrintWriter pw = new PrintWriter(System.out, true);
- pw.println("This is a string");
- int i = -7;
- pw.println(i);
- double d = 4.5e-7;
- pw.println(d);
- }
- }
- **The output from this program is shown here**:
- This is a string
- -7
- 4.5E-7

Applet Fundamentals

- *Applets* are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a web document.
- Applets differ from console-based applications in several key areas. The simple applet shown here:
- import java.awt.*;
- import java.applet.*;
- public class SimpleApplet extends Applet {
- public void paint(Graphics g) {
- g.drawString("A Simple Applet", 20, 20);
- }
- }

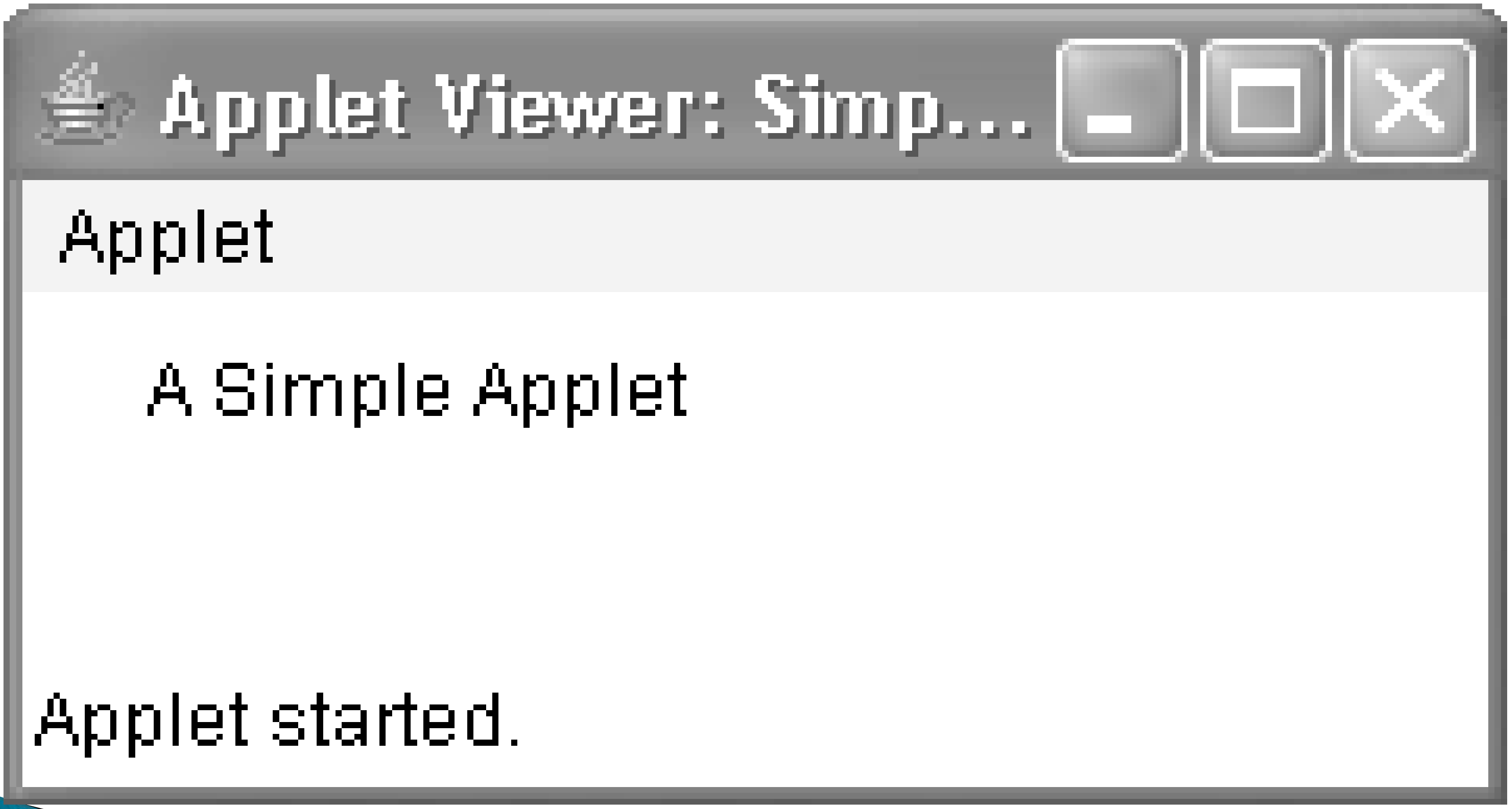
- This applet begins with two **import** statements. The first imports the Abstract Window Toolkit (AWT) classes. Applets interact with the user (either directly or indirectly) through the AWT, not through the console-based I/O classes.
- The second **import** statement imports the **applet** package, which contains the class **Applet**. Every applet that we create using a subclass of **Applet.**
- The next line in the program declares the class **SimpleApplet.** This class must be declared as **public**, because it will be accessed by code that is outside the program.

- Inside **SimpleApplet**, **paint( )** is declared. This method is defined by the AWT and must be overridden by the applet.
- Inside **paint( )** is a call to **drawString( )**, which is a member of the **Graphics** class.
- This method outputs a string beginning at the specified X,Y location. It has the following general form:
- void drawString(String *message*, int *x*, int *y*)
- Here, *message* is the string to be output beginning at *x,y*. In a Java window, the upper-left corner is location 0,0. The call to **drawString( )** in the applet causes the message “A Simple Applet” to be displayed beginning at location 20,20.

- **There are two ways in which we can run an applet:**
- • Executing the applet within a Java-compatible web browser.
- • Using an applet viewer, such as the standard tool, **appletviewer**. An applet viewer executes our applet in a window.
- **To execute an applet in a web browser, we need to write a short HTML text file that contains a tag that loads the applet.**
- **Here is the HTML file that executes SimpleApplet:**
- <applet code="SimpleApplet" width=200 height=60>
- </applet>

- The **width** and **height** statements specify the dimensions of the display area used by the applet.
- To execute **SimpleApplet** with an applet viewer, the following command line will run **SimpleApplet**:
- C:\>appletviewer RunApp.html
- Applets do not need a **main( )** method.
- User I/O is not accomplished with Java’s stream I/O classes. Instead, applets use the interface provided by the AWT or Swing.
- The window produced by **SimpleApplet**, as displayed by the applet viewer, is shown in the following illustration:

SimpleApplet Output:



```
A Simple Applet
```

Applet started.

The Applet Class :

- The **Applet** class defines the methods shown in the following table:

Method	Description
void destroy( )	Called by the browser just before an applet is terminated.
AccessibleContext getAccessibleContext( )	Returns the accessibility context for the invoking object.
AppletContext getAppletContext( )	Returns the context associated with the applet.
String getAppletInfo( )	Returns a string that describes the applet.
AudioClip getAudioClip(URL url)	Returns an AudioClip object that encapsulates the audio clip found at the location specified by url.

Method	Description
AudioClip getAudioClip(URL url, String clipName)	Returns an AudioClip object that encapsulates the audio clip found at the location specified by url.
URL getCodeBase( )	Returns the URL associated with the invoking applet.
URL getDocumentBase( )	Returns the URL of the HTML document that invokes the applet.
Image getImage(URL url)	Returns an Image object that encapsulates the image found at the location specified by url.
Image getImage(URL url, String imageName)	Returns an Image object that encapsulates the image found at the location specified by url.
Locale getLocale( )	Returns a Locale object that is used by various locale sensitive classes and methods.
String getParameter(String paramName)	Returns the parameter associated with paramName. null is returned if the specified parameter is not found.

Method	Description
String[ ] [ ] getParameterInfo( )	Returns a String table that describes the parameters recognized by the applet.
void init( )	Called when an applet begins execution. It is the first method called for any applet.
boolean isActive( )	Returns true if the applet has been started. It returns false if the applet has been stopped.
static final AudioClip newAudioClip(URL url)	Returns an AudioClip object that encapsulates the audio clip found at the location specified by url.
void play(URL url)	If an audio clip is found at the location specified by url, the clip is played.
void play(URL url, String clipName)	If an audio clip is found at the location specified by url with the name specified by clipName, the clip is played.
void start( )	Called by the browser when an applet should start (or resume) execution.
void stop( )	Called by the browser to suspend execution of the applet.

Applet Architecture

- An applet is a window-based program. First, applets are event driven. Second, the user initiates interaction with an applet.
- *Applet Skeleton :*
- 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**.

- *Applet Initialization and Termination :*
- **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 we should initialize variables. This method is called only once during the run time of our applet.**

- ***start( ):***
- The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped. if a user leaves a web page and comes back, the applet resumes execution at **start( )**.
- ***paint( ):***
- The **paint( )** method is called each time our applet's output must be redrawn. **paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called.

- *stop( ) :*
- The **stop( )** method is called when a web browser leaves the HTML document containing the applet.
- We should use **stop( )** to suspend threads that don't need to run when the applet is not visible. We 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 our applet needs to be removed completely from memory. We should free up any resources the applet may be using.

Applet Display Methods

- Applets are displayed in a window, and AWT-based applets use the AWT to perform input and output.
- To output a string to an applet, use **drawString( )**, which is a member of the **Graphics** class.
- Typically, it is called from within either **update( )** or **paint( )**. It has the following general form:
- void drawString(String *message*, int *x*, int *y*)

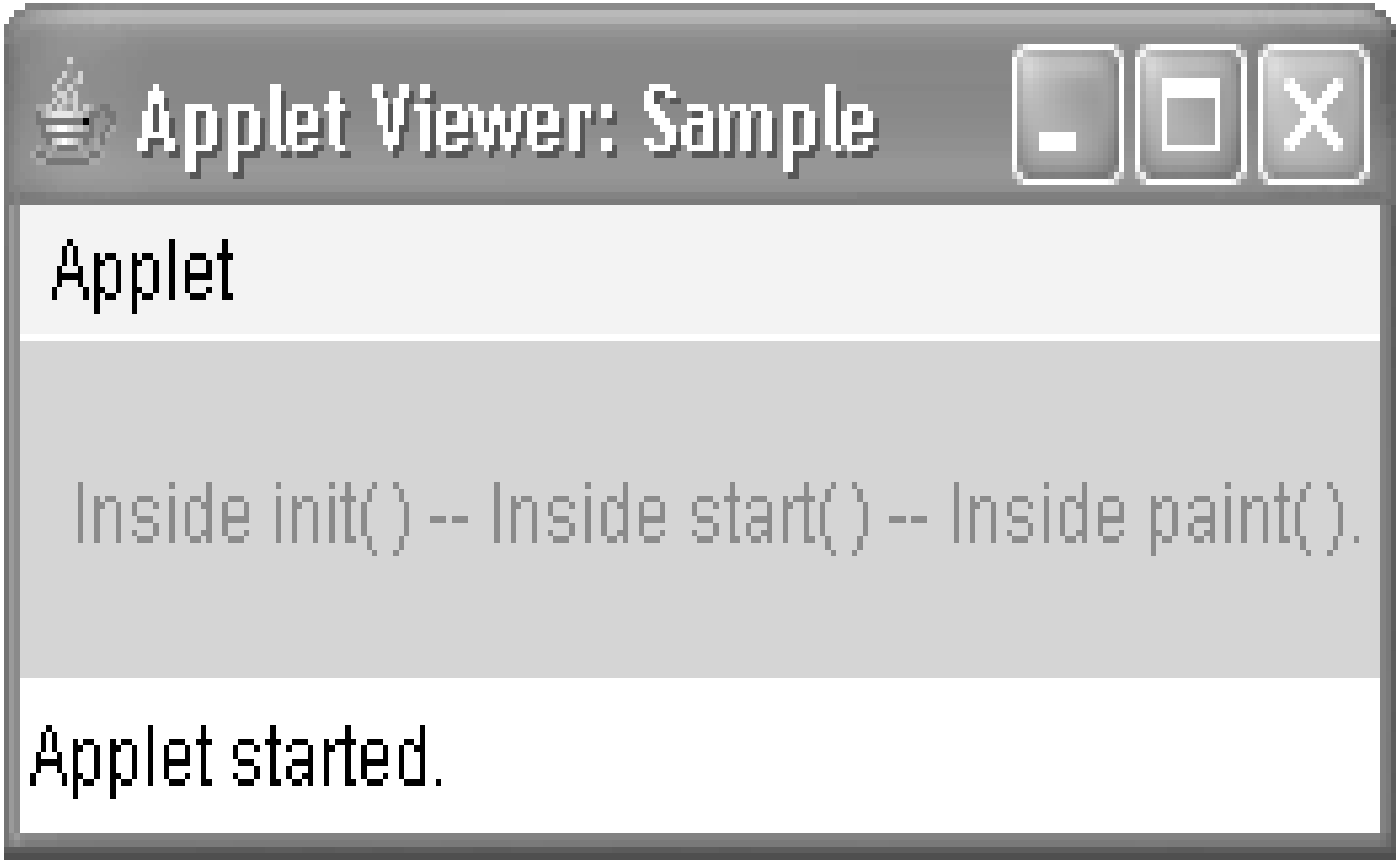
- Here, *message* is the string to be output beginning at *x,y.* In a Java window, the upper-left corner is location 0,0.
- To set the background color of an applet's window, use **setBackground( )**. To set the foreground color (the color in which text is shown), use **setForeground( )**.
- They have the following general forms:
- void setBackground(Color *newColor*)
- void setForeground(Color *newColor*)
- Here, *newColor* specifies the new color. The class **Color** defines the constants.

- The following example sets the background color to green and the text color to red:
- setBackground(Color.green);
- setForeground(Color.red);
- We can obtain the current settings for the background and foreground colors by calling **getBackground( )** and **getForeground( )**, respectively.
- The following simple example applet that sets the background color to cyan, the foreground color to red, and displays a message that illustrates the order in which the **init( )**, **start()**, and **paint( )** methods are called when an applet starts up:

- /* A simple applet that sets the foreground and
- background colors and outputs a string. */
- import java.awt.*;
- import java.applet.*;
- /*<applet code="Sample" width=300 height=50>
- </applet> */
- public class Sample extends Applet{
- String msg;
- // set the foreground and background colors.
- public void init() {
- setBackground(Color.cyan);
- setForeground(Color.red);
- msg = "Inside init( ) --";
- }

```
// Initialize the string to be displayed.
public void start() {
msg += " Inside start( ) --";
}
// Display msg in applet window.
public void paint(Graphics g) {
msg += " Inside paint( ).";
g.drawString(msg, 10, 30);
}
}
```

- **This applet generates the window shown in the following diagram:**



```
Inside init( ) -- Inside start( ) -- Inside paint( ).
```

```
Applet started.
```

The HTML APPLET Tag

- The APPLET tag be used to start an applet from both an HTML document and from an applet viewer.
- An applet viewer will execute each APPLET tag that it finds in a separate window, while web browsers will allow many applets on a single page.
- The syntax for a fuller form of the APPLET tag is shown in following diagram. Bracketed items are optional.

- < APPLET
- [CODEBASE = *codebaseURL*]
- CODE = *appletFile*
- [ALT = *alternateText*]
- [NAME = *appletInstanceName*]
- WIDTH = *pixels* HEIGHT = *pixels*
- [ALIGN = *alignment*]
- [VSPACE = *pixels*] [HSPACE = *pixels*]
- >
- [< PARAM NAME = *AttributeName* VALUE = *AttributeValue*>]
- [< PARAM NAME = *AttributeName2* VALUE = *AttributeValue*>]
- . . .
- [*HTML Displayed in the absence of Java*]
- </APPLET>

- ***CODEBASE :***
- **CODEBASE is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file (specified by the CODE tag).**
- ***CODE :***
- **CODE is a required attribute that gives the name of the file containing our applet's compiled .class file. This file is relative to the code base URL of the applet.**
- ***ALT :***
- **The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser recognizes the APPLET tag but can't currently run Java applets.**

- ***NAME :***
- **NAME is an optional attribute used to specify a name for the applet instance. Applets must be named in order for other applets on the same page to find them by name and communicate with them.**
- ***WIDTH and HEIGHT :***
- **WIDTH and HEIGHT are required attributes that give the size (in pixels) of the applet display area.**
- ***ALIGN :***
- **ALIGN is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.**

- *VSPACE and HSPACE :*
- These attributes are optional. VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet.
- They're treated the same as the IMG tag's VSPACE and HSPACE attributes.
- *PARAM NAME and VALUE :*
- The PARAM tag allows us to specify applet-specific arguments in an HTML page. Applets access their attributes with the **getParameter()** method.
- Other valid APPLET attributes include ARCHIVE, which lets specify one or more archive files, and OBJECT, which specifies a saved version of the applet.