Java Programming

MCA 205

Unit - II

Learning Objectives

- **Exception Handling:** Fundamentals exception types, uncaught exceptions, throw, throw, final, built in exception, creating your own exceptions,
- **Multithreaded Programming:** Fundamentals, Java thread model: priorities, synchronization, messaging, thread classes, Runnable interface, inter thread Communication, suspending, resuming and stopping threads.
- **Input/Output Programming:** Basics, Streams, Byte and Character Stream, predefined streams, Reading and writing from console and files.
- **Using Standard Java Packages (lang, util, io, net).** Networking: Basics, networking classes and interfaces, using java.net package, doing TCP/IP and Data-gram Programming, RMI (Remote Method Invocation)

Introduction

- Users have high expectations for the code we produce. Users will use our programs in unexpected ways.
- Due to design errors or coding errors, our programs may fail in unexpected ways during execution
- It is our responsibility to produce quality code that does not fail unexpectedly.
- Consequently, we must design error handling into our programs.
- An Error is any unexpected result obtained from a program during execution.

Introduction

- Unhandled errors may manifest themselves as incorrect results or behavior, or as abnormal program termination.
- Errors should be handled by the programmer, to prevent them from reaching the user.
- Some typical causes of errors:
  - Memory errors (i.e. memory incorrectly allocated, memory leaks, "null pointer")
  - File system errors (i.e. disk is full, disk has been removed)
  - Network errors (i.e. network is down, URL does not exist)
  - Calculation errors (i.e. divide by 0)

Errors and Error Handling

- More typical causes of errors:
  - Array errors (i.e. accessing element –1)
  - Conversion errors (i.e. convert 'q' to a number)
  - Can you think of some others?
- Traditional Error Handling
  - 1. Every method returns a value (flag) indicating either success, failure, or some error condition. The calling method checks the return flag and takes appropriate action.
  - Downside: programmer must remember to always check the return value and take appropriate action. This requires much code (methods are harder to read) and something may get overlooked.

Errors and Error Handling

- Traditional Error Handling
  - Where used: traditional programming languages (i.e. C) use this method for almost all library functions (i.e. fopen() returns a valid file or else null)
  - Create a global error handling routine, and use some form of "jump" instruction to call this routine when an error occurs.
  - Downside: "jump" instruction (Goto) are considered "bad programming practice" and are discouraged. Once you jump to the error routine, you cannot return to the point of origin and so must (probably) exit the program.

Errors and Error Handling

- Exceptions – a better error handling
  - Exceptions are a mechanism that provides the best of both worlds.
  - Exceptions act similar to method return flags in that any method may raise an exception should it encounter an error.
  - Exceptions act like global error methods in that the exception mechanism is built into Java; exceptions are handled at many levels in a program, locally and/or globally.
- 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.

Errors and Error Handling

- What are they?
  - An exception is a representation of an error condition or a situation that is not the expected result of a method.
  - Exceptions are built into the Java language and are available to all program code.
  - Exceptions isolate the code that deals with the error condition from regular program logic.
- The object-oriented techniques to manage such errors comprise the group of methods known as exception handling.
- For the most part, exception handling has not changed since the original version of Java. However, Java 2, version 1.4 has added a new subsystem called the *chained exception facility.*
- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.

Exception Handling

- When an exceptional condition arises, an object representing that exception is created and *thrown in the method that caused the error.*
- That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught and processed.*
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.
- Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.
- Manually generated exceptions are typically used to report some error condition to the caller of a method.

Exception Handling

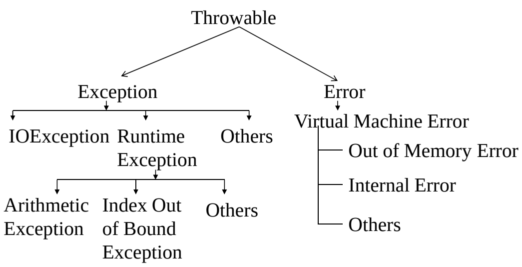
- Java exception handling is managed via five keywords: **try, catch, throw, throws,** and **finally.**
- Program statements that you want to monitor for exceptions are contained within a **try block.**
- **If an exception occurs within** the **try block, it is thrown. Your code can catch this exception (using catch) and handle** it in some rational manner.
- System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword **throw.**
- 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 before a method returns is put in a **finally block.**

Exception Handling

- 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 before try block ends }

Exception Types



Errors and Error Handling

- How are they used?
  - Exceptions fall into two categories:
    - ✓Checked Exceptions
    - ✓Unchecked Exceptions
  - Checked exceptions are inherited from the core Java class Exception. They represent exceptions that are frequently considered “non fatal” to program execution
  - Checked exceptions must be handled in your code, or passed to parent classes for handling.

Unchecked Exceptions

```
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 execution of* ***Exc0*** to stop, because once an exception has been thrown, it must be *caught by an exception* handler and dealt with immediately.
- Any exception that is not caught by your program will ultimately be processed by the default handler. 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 output generated when this example is executed.

```
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)
```


Errors and Error Handling

- How are they used?
  - Unchecked exceptions represent error conditions that are considered “fatal” to program execution.
  - You do not have to do anything with an unchecked exception. Your program will terminate with an appropriate error message.
- Examples:
  - Checked exceptions include errors such as “array index out of bounds”, “file not found” and “number format conversion”.
  - Unchecked exceptions include errors such as “null pointer”.

Exceptions

- How do you handle exceptions?
  - Exception handling is accomplished through the “try – catch” mechanism, or by a “throws” clause in the method declaration.
  - For any code that throws a checked exception, you can decide to handle the exception yourself, or pass the exception “up the chain” (to a parent class).

 U1.‹#›

Exceptions

- How do you handle exceptions?
  - To handle the exception, you write a “try-catch” block. To pass the exception “up the chain”, you declare a throws clause in your method or class declaration.
  - If the method contains code that may cause a checked exception, you MUST handle the exception OR pass the exception to the parent class (remember, every class has Object as the ultimate parent)

 U1.‹#›

Coding Exceptions

- Try-Catch Mechanism
  - Wherever your code may trigger an exception, the normal code logic is placed inside a block of code starting with the “try” keyword:
  - After the try block, the code to handle the exception should it arise is placed in a block of code starting with the “catch” keyword.
  - You may also write an optional “finally” block. This block contains code that is ALWAYS executed, either after the “try” block code, or after the “catch” block code.
  - Finally blocks can be used for operations that must happen no matter what (i.e. cleanup operations such as closing a file)

 U1.‹#›

Coding Exceptions

- Syntax
  - try { //… normal program code
    }
    catch(Exception e) {
    … exception handling code
    }
- Passing the exception
  - In any method that might throw an exception, you may declare the method as "throws" that exception, and thus avoid handling the exception yourself
  - Example
    - ✓public void myMethod throws IOException {
      … normal code with some I/O}

Coding Exceptions

- Types of Exceptions
  - All checked exceptions have class "Exception" as the parent class.
  - You can use the actual exception class or the parent class when referring to an exception
- Types of Exceptions
  - Examples:
    - ✓public void myMethod throws Exception {
    - ✓public void myMethod throws IOException {
    - ✓try { … }
      catch (Exception e) { … }
    - ✓try { … }
      catch (IOException ioe) { … }

Using try and catch

- Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself.
- Doing so provides two benefits. First, it allows you to fix the error. Second, it prevents the program from automatically terminating.
- Most users would be confused (to say the least) if your program stopped running and printed a stack trace whenever an error occurred! Fortunately, it is quite easy to prevent this.
- To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a **try block.**
- **Immediately following the try block, include a catch** clause that specifies the exception type that you wish to catch.

Example

```
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."); }
System.out.println("After catch statement."); } }
/* This program generates the following output:
Division by zero.
After catch statement.*/
```


Displaying Description of an Exception

- **Throwable** overrides the **toString( )** method (defined by **Object**) so that it returns a string containing a description of the exception. You can display this description in a **println( )** statement by simply passing the exception as an argument.
- For example, the **catch** block in the preceding program can be rewritten like this:

```
catch (ArithmeticException e) {
System.out.println("Exception: " + e);
a = 0; // set a to zero and continue }
```

- When this version is substituted in the program, and the program is run, each divide-by-zero error displays the following message:

  Exception: java.lang.ArithmeticException: / by zero
- While it is of no particular value in this context, the ability to display a description of an exception is valuable in other circumstances—particularly when you are experimenting with exceptions or when you are debugging.

Multiple catch clauses

- In some cases, more than one exception could be raised by a single piece of code. To handle this situation, you can specify two or more **catch** clauses, each catching a different type of exception.
- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try**/**catch** block.
- // Demonstrate multiple catch statements.

```
class MultiCatch {public static void main(String args[]) {
try { int a = args.length; System.out.println("a = " + a);
int b = 42 / a;
int c[] = { 1 }; c[42] = 99; } catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
 } catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index oob: " + e); }
System.out.println("After try/catch blocks.");} }
```

Multiple catch clauses

- When you use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses.
- This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass.

Nested try

- The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**.
- Each time a **try** statement is entered, the context of that exception is pushed on the stack. If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match. This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted.
- If no **catch** statement matches, then the Java run-time system will handle the exception


```
class NestTry { public static void main(String args[]) {
try { int a = args.length;
/* If no command-line args are present,the following statement will
generate a divide-by-zero exception. */
int b = 42 / a;   System.out.println("a = " + a);
try { // nested try block
/* If one command-line arg is used, then a divide-by-zero exception
will be generated by the following code. */
if(a==1) a = a/(a-a); // division by zero
/* If two command-line args are used, then generate an out-of-bounds
exception. */
if(a==2) { int c[] = { 1 };
c[42] = 99; // generate an out-of-bounds exception
} } catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index out-of-bounds: " + e);
} } catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);}}}
```

Nested try

- Nesting of **try** statements can occur in less obvious ways when method calls are involved.
- For example, you can enclose a call to a method within a **try** block. Inside that method is another **try** statement. In this case, the **try** within the method is still nested inside the outer **try** block, which calls the method.

throw

- Instead of Java Run-Time system, it is also possible for your 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**. Simple 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 into 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 the 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.


```
// 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);}}}
```

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.


```
// 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);
}
}
}
```


finally

- When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods.
- For example, if a method opens a file upon entry and closes it upon exit, then you 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 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.

```
// 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();}}
```

Note: If a finally block is associated with a try, the finally block will be executed upon conclusion of the try

Java's Built-in Exceptions

- Inside the standard package **java.lang**, Java defines several exception classes. A few have been used by the preceding examples. The most general of these exceptions are subclasses of the standard type **RuntimeException**.
- Since **java.lang** is implicitly imported into all Java programs, most exceptions derived from **RuntimeException** are automatically available. Furthermore, they need not be included in any method's **throws** list. In the language of Java, these are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in **java.lang** are listed in Table below.
- Next 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.* Java defines several other types of exceptions that relate to its various class libraries

Some Built-In Exception Classes

Exception	Meaning
ArithmeticException	Arithmetic-error such as divide by zero
ArrayIndexOutOfBoundsException	Array index is out of Bounds
ArrayStoreException	Assignment to an array element of an incompatible type
ClassCastException	Invalid Cast
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

Java's Checked Exceptions defined in java.lang

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
NoSuch FieldException	A requested field does not exist
NoSuchMethodException	A requested method does not exist

Creating your own Exception subclasses

- Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications.
- This is quite easy to do: just define a subclass of **Exception** (which is, of course, a subclass of **Throwable**). Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.
- The **Exception** class does not define any methods of its own. It does, of course, inherit those methods provided by **Throwable**. Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them.

Methods defined by Throwable

Method	Description
Throwable fillInStackTrace()	Returns a Throwable object that contains a completed stack trace. The object can be rethrown.
Throwable getCause()	Returns the exception that underlies the current exception. If there is no underlying exception, null is returned. Added by Java2, version 1.4.
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. The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array.
Throwable initCause(Throwable causeExc)	Associates causeExc with the invoking exception as a cause of the invoking exception. Returns a reference to the exception. Added by Java2 version 1.4.

Methods defined by Throwable

Method	Description
void printStackTrace()	Displays the stack trace
void printStackTrace (PrintStream stream)	Sends the stack trace to the specified stream
void PrintStackTrace (PrintWriter stream)	Sends the stack trace to the specified stream
void setStackTrace(StackTraceElement elements[])	Sets the stack trace to the elements passed in elements. This method is for specialized applications not normal use. Added by Java2 version 1.4
String toString()	Returns a string object containing a description of the exception. This method is called by println() when outputting a Throwable object

Chained Exceptions

- Java 2, version 1.4 added a new feature to the exception subsystem: *chained exceptions*.
- The chained exception feature allows you to associate another exception with an exception. This second exception describes the cause of the first exception. For example, imagine a situation in which a method throws an **ArithmeticException** because of an attempt to divide by zero. However, the actual cause of the problem was that an I/O error occurred, which caused the divisor to be set improperly.
- Although the method must certainly throw an **ArithmeticException**, since that is the error that occurred, you might also want to let the calling code know that the underlying cause was an I/O error.
- Chained exceptions let you handle this, and any other situation in which layers of exceptions exist. To allow chained exceptions, Java 2, version 1.4 added two constructors and two methods to **Throwable**.
- The constructors are shown here.

  Throwable(Throwable *causeExc*)

  Throwable(String *msg*, Throwable *causeExc*)


```
class ChainExcDemo {
static void demoproc() {
// create an exception
NullPointerException e =
new NullPointerException("top layer");
// add a cause
e.initCause(new ArithmeticException("cause"));
throw e; }
public static void main(String args[]) {
try {demoproc();}
 catch(NullPointerException e) {
// display top level exception
System.out.println("Caught: " + e);
// display cause exception
System.out.println("Original cause: " +e.getCause());}}}
```

```
// 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);}}}
```


Multithreading

- A multithreaded program contains two or more parts that can run concurrently.
- Each part of the program is called a *thread*, that defines a separate path of execution. Multithreading is a specialized form of multitasking.
- Multitasking, supported by virtually all modern operating systems is of two distinct types of multitasking: process-based and thread-based.
- Process-based multitasking is more familiar. A *process is, in essence, a program that is executing. Thus, process-based* multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

Introduction

- In *thread-based multitasking ,the thread is the smallest unit of* dispatchable code. This means that a single program can perform two or more tasks simultaneously. For ex, a text editor can format text at the same time that it is printing, as long as these actions are being performed by two separate threads.
- Thus, process-based multitasking deals with the "big picture," and thread-based multitasking handles the details.
- 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.

Introduction

- Threads, on the other hand, are lightweight, share the same space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is low cost. Java programs make use of process-based multitasking environments, process-based multitasking is not under the control of Java. However, multithreaded multitasking is.
- Multithreading enables writing very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.
- This is especially important for the interactive, networked environment in which Java operates, because idle time is common.

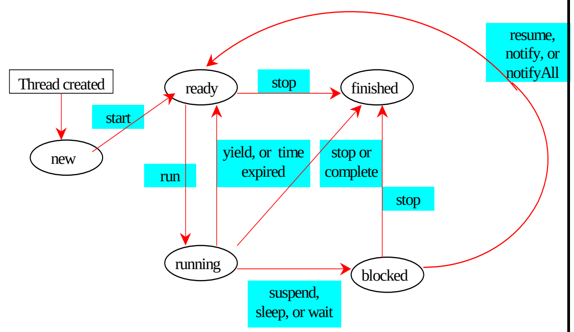
Java Thread Model

- The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind. Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.
- 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. Once this polling mechanism returns with, say, a signal that a network file is ready to be read, then the event loop dispatches control
- to the appropriate event handler. Until this event handler returns, nothing else can happen in the system.

Java Thread Model

- The benefit of Java's multithreading is that one thread can pause without stopping other parts of your program. For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere. Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause.
- When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

Thread States



 U1.‹#›

Thread Priorities

- Java assigns to each thread a priority that determines how that thread should be treated with respect to the others.
- Thread priorities are integers that specify the relative priority of one thread to another. As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running.
- Instead, 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 are :

i) A thread can voluntarily relinquish control.

ii) A thread can be preempted by a higher-priority thread.

- In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated.

 U1.‹#›

Synchronization

- Multithreading introduces an asynchronous behavior to programs, there must be a way to enforce synchronicity when needed.
- For example, if two threads want to communicate and share a complicated data structure, such as a linked list, there needs to be some way to ensure that they don't conflict with each other. That is, one must prevent one thread from writing data while another thread is in the middle of reading it.
- For this purpose, Java implements an elegant twist on an age-old model of interprocess synchronization: the *monitor.*
- *The monitor is a control mechanism* first defined by C.A.R. Hoare. It can be considered a very small box that can hold only one thread. Once a thread enters monitor, all other threads must wait until that thread exits the monitor. A monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

 U1.‹#›

Synchronization

- Most multithreaded systems expose monitors as objects that your program must explicitly acquire and manipulate.
- Java provides a cleaner solution. There is no class “Monitor”; instead, 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. This enables you to write very clear and concise multithreaded code, because synchronization support is built in to the language.

 U1.

Messaging

- Once a program is divided into separate threads, one needs to define how they will communicate with each other.
- When programming with most other languages, one must depend on the operating system to establish communication between threads.
- This, of course, adds overhead. By contrast, 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.

 U1.

The Thread Class and the Runnable Interface

- Java’s multithreading system is built upon the **Thread class, its methods, and its** companion interface, **Runnable**
- **Thread encapsulates a thread of execution. Since** you can’t directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the **Thread instance that spawned it.**
- **To create a new thread, your** program will either extend **Thread or implement the Runnable interface.**
- The **Thread class defines several methods that help manage threads.**

 U1.

Notable Methods

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

 U1.

The Main Thread

- When a Java program begins, one thread begins running immediately. This is usually called the *main thread of your program, because it is the one that is executed* when your 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 your program begins, it can be controlled through a **Thread object. To do so, you must obtain a reference to it** by calling the method **currentThread( ), which is a public static member of Thread class. Its** general form is shown here:

  static Thread currentThread( )
- This method returns a reference to the thread in which it is called. A reference to the main thread, allows controlling it just like any other thread.

 U1.

```
// Controlling the main Thread.
class CurrentThreadDemo {
public static void main(String args[]) {
Thread t = Thread.currentThread();
System.out.print("Current thread: " + t);
// change the name of the thread
t.setName("My Thread");
System.out.print("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");}}}
```

 U1.

Creating a Thread

- You create a thread by instantiating an object of type **Thread.**
- Java defines two ways in which this can be accomplished:
- ■ You can implement the **Runnable interface.**
- ■ You can extend the **Thread class, itself.**

Implementing Runnable

- The easiest way to create a thread is to create a class that implements the **Runnable** interface.
- **You can construct a thread on** any object that implements **Runnable. To implement Runnable, a class need only** implement a single method called **run( ), which is declared like this:**

  public void run( )

- Inside **run( ), you will define the code that constitutes the new thread. It is** important to understand that **run( ) can call other methods, use other classes, and** declare variables, just like the main thread can. The only difference is that **run( )** establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run( ) returns.**

Implementing Runnable

- After you create a class that implements **Runnable, you will instantiate an object of** type **Thread from within that class. Thread defines several constructors. The one that** we will use is :

  Thread(Runnable *threadOb, String threadName)*

- In this constructor, *threadOb is an instance of a class that implements the* ***Runnable*** interface. This defines where execution of the thread will begin. The name of the new thread is specified by *threadName.*
- After the new thread is created, it will not start running until you call its **start( )** method, which is declared within **Thread. In essence, start( ) executes a call to run( ).**
- The **start( ) method is shown here:**

  void start( )

```
class NewThread implements Runnable { Thread t;
NewThread() { // Create a new, second thread t
t.start(); }// Start the thread
// This is the entry point for the second thread.
public void run() { try { for(int i = 5; i > 0; i--) {
System.out.println("Child Thread: " + i);
Thread.sleep(500); }
} catch (InterruptedException e) { System.out.println("Child
interrupted."); }
System.out.println("Exiting child thread.");}}
class ThreadDemo {
public static void main(String args[]) {// create a new thread
try {for(int i = 5; i > 0; i--) { System.out.println("Main Thread: " + i);
Thread.sleep(1000);} } catch (InterruptedException e) {
System.out.println("Main thread interrupted.");}
System.out.println("Main thread exiting.");}}
```


Extending Thread

- The second way to create a thread is to create a new class that extends **Thread,** and then to create an instance of that class. The extending class must override the **run( ) method, which is the entry point for the new thread. It must also call start( )** to begin execution of the new thread. Here is the preceding program rewritten to extend **Thread:**


```
class NewThread extends Thread {// Create a new, second thread
start(); } // Start the thread
public void run() { try { for(int i = 5; i > 0; i--) {
System.out.println("Child Thread: " + i);  Thread.sleep(500);}
} catch (InterruptedException e) {
System.out.println("Child interrupted.");}
System.out.println("Exiting child thread.");}}
class ExtendThread {
public static void main(String args[]) { // create a new thread
try { for(int i = 5; i > 0; i--) { System.out.println("Main Thread: " + i);
Thread.sleep(1000);}} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");}
System.out.println("Main thread exiting.");}}
```


Choosing an Approach

- Java has two ways to create child threads, which approach is better.? The answers to these questions turn on the same point.
- The **Thread class defines several methods that can be overridden by a derived class.**
- Of these methods, the only one that *must be overridden is* ***run( ). This is, of course, the*** same method required when you implement **Runnable.**
- **Many Java programmers feel** that classes should be extended only when they are being enhanced or modified in some way. So, if you will not be overriding any of **Thread's other methods, it is** probably best simply to implement **Runnable.**

Creating Multiple Threads

- So far, we have been using only two threads: the main thread and one child thread.
- However, your program can spawn as many threads as it needs. For example, the following program creates three child threads:
- // Create multiple threads.

```
class NewThread implements Runnable {String name;//threadname
Thread t; NewThread(String threadname) { name = threadname;
t = new Thread(this, name); System.out.println("New thread: " + t);
    t.start(); }// Start thread, This is the entry point for thread.
public void run() {try { for(int i = 5; i > 0; i--) {
System.out.println(name + ": " + i); Thread.sleep(1000);}}
catch (InterruptedException e){System.out.println(name+"Interrupted");}
System.out.println(name + " exiting.");}}
```



```
class MultiThreadDemo {
public static void main(String args[]) {
new NewThread("One"); // start threads
new NewThread("Two");
new NewThread("Three");
try {// wait for other threads to end
Thread.sleep(10000);} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");}
System.out.println("Main thread exiting.");}}
```

Using isAlive( ) and join( )

- As mentioned, the main thread should finish last. In the preceding examples, this is accomplished by calling **sleep( ) within main( ), with a long enough** delay to ensure that all child threads terminate prior to the main thread.
- However, How can one thread know when another thread has ended? Fortunately, **Thread provides a means** to answer this question.
- Two ways exist to determine whether a thread has finished. First, you can call **isAlive( ) on the thread. This method is defined by Thread, and its general form is** shown here:
- final boolean isAlive( )
- While **isAlive( ) is occasionally useful, the method that you will more commonly** use to wait for a thread to finish is called **join( ), shown here:**
- final void join( ) throws InterruptedException


```
class NewThread implements Runnable { String name; // threadname
Thread t; NewThread(String threadname) {name = threadname;
t = new Thread(this, name); System.out.println("New thread: " + t);
t.start(); }// Start the thread ,This is the entry point for thread.
public void run() { try {for(int i = 5; i > 0; i--)
{System.out.println(name + ": " + i); Thread.sleep(1000);}
} catch (InterruptedException e) {
System.out.println(name + " interrupted.");}
System.out.println(name + " exiting.");}}
class DemoJoin {public static void main(String args[]) {
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
System.out.println("Thread One is alive: "+ ob1.t.isAlive());
System.out.println("Thread Two is alive: "+ ob2.t.isAlive());
try {System.out.println("Waiting for threads to finish.");  ob1.t.join();
```



```
ob2.t.join();
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");}
System.out.println("Thread One is alive: "+ ob1.t.isAlive());
System.out.println("Thread Two is alive: "+ ob2.t.isAlive());
System.out.println("Main thread exiting.");
}
}
```

Thread Priorities

- 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 lowerpriority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.)
- 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.

Thread Priorities

- In theory, threads of equal priority should get equal access to the CPU. But you need to be careful.
- Java is designed to work in a wide range of environments. Some of those environments implement multitasking fundamentally differently than others.
- For safety, threads that share the same priority should yield control once in a while. This ensures that all threads have a chance to run under a nonpreemptive operating system.
- In practice, even in nonpreemptive environments, most threads still get a chance to run, because most threads inevitably encounter some blocking situation, such as waiting for I/O. Also, some types of tasks are CPU-intensive. Such threads dominate the CPU. For these types of threads, you want to yield control occasionally, so that other threads can run.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason

Thread Priorities

- To set a thread's priority, use the **setPriority( ) method, which is a member of Thread. This is its general form:**

  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 final** variables within **Thread.**
- You can obtain the current priority setting by calling the **getPriority( ) method of Thread, shown here:**

  final int getPriority( )

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason

Synchronization

- When two or more threads need to access a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization.* Java provides unique, language-level support for it.
- Key to synchronization is the concept of the monitor (also called a *semaphore*). A *monitor* is an object that is used as a mutually exclusive lock, or *mutex.* Only one thread can *own* a monitor at a given time.
- When a thread acquires a lock, it is said to have *entered* the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* the monitor. These other threads are said to be *waiting* for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

Synchronization

- Synchronization in other languages, such as C or C++, can be a bit tricky to use. This is because most languages do not, themselves, support synchronization. Instead, to synchronize threads, your programs need to utilize operating system primitives.
- Fortunately, as Java implements synchronization through language elements, most of the complexity associated with synchronization has been eliminated.
- One can synchronize code in either of two ways. Both involve the use of the **synchronized** keyword, and both are examined here:

i) Using Synchronized Methods

ii) The synchronized Statement

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason

Using Synchronized Methods

- Synchronization is easy in Java, because all objects have their own implicit monitor associated with them.
- To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait.
- To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Ms. Ritika Wason


```
class Callme {synchronized void call(String msg) {
System.out.print("[" + msg);
try {Thread.sleep(1000);} catch(InterruptedException e) {
System.out.println("Interrupted");} System.out.println("]");}}
class Caller implements Runnable {
String msg; Callme target; Thread t;
public Caller(Callme targ, String s) {target = targ; msg = s;
t = new Thread(this); t.start();}
public void run() { target.call(msg);}}
class Synch { public static void main(String args[]) {
Callme target = new Callme();
Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target, "Synchronized");
Caller ob3 = new Caller(target, "World");
try { ob1.t.join(); ob2.t.join(); ob3.t.join();}
catch(InterruptedException e) {System.out.println("Interrupted");}}}
```


The synchronized Statement

- Creating **synchronized** methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases.
- To understand why, consider the following. Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods. Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add **synchronized** to the appropriate methods within the class. How can access to an object of this class be synchronized? Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a **synchronized** block.
- This is the general form of the **synchronized** statement:

  synchronized(*object*) { // statements to be synchronized }


```
class Callme { void call(String msg) {
System.out.print("[" + msg); try{Thread.sleep(1000);}
catch (InterruptedException e) {System.out.println("Interrupted");}
System.out.println("]");}}
class Caller implements Runnable {String msg;Callme target;Thread t;
public Caller(Callme targ, String s) {
target = targ; msg = s; t = new Thread(this); t.start(); }
public void run() {synchronized(target) { // synchronizedblock
target.call(msg);}}}
class Synch1 { public static void main(String args[]) {
Callme target = new Callme(); Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target,"Synchronized");
Caller ob3 = new Caller(target, "World");
try {ob1.t.join(); ob2.t.join(); ob3.t.join();
} catch(InterruptedException e) {System.out.println( “Interrupted”);}}}
```


Interthread Communication

- One can unconditionally block other threads from asynchronous access to certain methods. This use of the implicit monitors in Java objects is powerful, but you can achieve a more subtle level of control through interprocess communication.
- As discussed earlier, multithreading replaces event loop programming by dividing your tasks into discrete and logical units.
- Threads also provide a secondary benefit: they do away with polling. Polling is usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken. This wastes CPU time. To avoid polling, Java includes an elegant interprocess communication mechanism via the **wait( )**, **notify( )**, and **notifyAll( )** methods.
- These methods are implemented as **final** methods in **Object**, so all classes have them. All three methods can be called only from within a **synchronized** context.

wait(), notify() and notifyAll()

- **wait( )** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )**.
- ■ **notify( )** wakes up the first thread that called **wait( )** on the same object.
- ■ **notifyAll( )** wakes up all the threads that called **wait( )** on the same object.
- The highest priority thread will run first.
- These methods are declared within **Object**, as shown here:

```
final void wait( ) throws InterruptedException
final void notify( )
final void notifyAll( )
```

- Additional forms of **wait( )** exist that allow you to specify a period of time to wait.


```
public class WaitMethod extends Thread{ String st="Hello";
public static void main(String args[]) {
WaitMethod wait=new WaitMethod();
wait.start(); new Example(wait); }
public void run(){
try { synchronized(this){
wait();
System.out.println("value is :"+st);}
}catch(Exception e){}}
public void valchange(String st){ this.st=st;
try { synchronized(this) {
notifyAll();}
}catch(Exception e){}}}
```

```
class Example extends Thread{
WaitMethod wait;
Example(WaitMethod wait) {
this.wait=wait;
start();
}
public void run(){
try{
System.out.println("Value is changed to: "+wait.st);
wait.valchange("Hello World");
}catch(Exception e){}
}
}
```


Deadlock

- A special type of error that one needs to avoid that relates specifically to multitasking is *deadlock,* which occurs when two threads have a circular dependency on a pair of synchronized objects.
- For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete.
- Deadlock is a difficult error to debug for two reasons:
  - In general, it occurs only rarely, when the two threads time-slice in just the right way.
  - It may involve more than two threads and two synchronized objects. (That is, deadlock can occur through a more convoluted sequence of events than just described.)

Suspending, Resuming, & Stopping Threads

- Sometimes, suspending execution of a thread is useful. For example, a separate thread can be used to display the time of day. If the user doesn't want a clock, then its thread can be suspended. Whatever the case, suspending a thread is a simple matter. Once suspended, restarting the thread is also a simple matter.
- The mechanisms to suspend, stop, and resume threads differ between Java 2 and earlier versions. Although you should use the Java 2 approach for all new code, you still need to understand how these operations were accomplished for earlier Java environments.
- For example, you may need to update or maintain older, legacy code. You also need to understand why a change was made for Java 2.

Suspending, Resuming, and Stopping Threads Using Java 2

- While the **suspend( )**, **resume( )**, and **stop( )** methods defined by **Thread** seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs.
- Here's why: The **suspend( )** method of the **Thread** class is deprecated in Java 2. This was done because **suspend( )** can sometimes cause serious system failures. Assume that a thread has obtained locks on critical data structures. If that thread is suspended at that point, those locks are not relinquished. Other threads that may be waiting for those resources can be deadlocked.
- The **resume( )** method is also deprecated. It does not cause problems, but cannot be used without the **suspend( )** method as its counterpart.
- The **stop( )** method of the **Thread** class, too, is deprecated in Java 2. This was done because this method can sometimes cause serious system failures. Assume that a thread is writing to a critically important data structure and has completed only part of its changes. If that thread is stopped at that point, that data structure might be left in a corrupted state.

Suspending, Resuming, and Stopping Threads Using Java 2

- Because you can't use the **suspend( )**, **resume( )**, or **stop( )** methods in Java 2 to control a thread, you might be thinking that no way exists to pause, restart, or terminate a thread. But, fortunately, this is not true. Instead, a thread must be designed so that the **run( )** method periodically checks to determine whether that thread should suspend, resume, or stop its own execution.
- Typically, this is accomplished by establishing a flag variable that indicates the execution state of the thread. As long as this flag is set to "running," the **run( )** method must continue to let the thread execute. If this variable is set to "suspend," the thread must pause. If it is set to "stop," the thread must terminate.
- Of course, a variety of ways exist in which to write such code, but the central theme will be the same for all programs.


```
// Suspending and resuming a thread for Java 2
class NewThread implements Runnable {String name; Thread t;
 boolean suspendFlag;NewThread(String threadname) { name =
threadname;
t = new Thread(this, name);
System.out.println("New thread: " + t); suspendFlag = false; t.start();
}
public void run() { try { for(int i = 15; i > 0; i--) {
System.out.println(name + ": " + i); Thread.sleep(200);
synchronized(this) { while(suspendFlag) { wait(); } }}
} catch (InterruptedException e) {System.out.println(name + "
interrupted."); }
System.out.println(name + " exiting."); }
void mysuspend() { suspendFlag = true;}
synchronized void myresume() {
suspendFlag = false; notify();}}
```

```
class SuspendResume { public static void main(String args[]) {
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two"); try { Thread.sleep(1000);
ob1.mysuspend(); System.out.println("Suspending thread One");
Thread.sleep(1000); ob1.myresume();
System.out.println("Resuming thread One"); ob2.mysuspend();
System.out.println("Suspending thread Two"); Thread.sleep(1000);
ob2.myresume(); System.out.println("Resuming thread Two");
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted"); } try {
System.out.println("Waiting for threads to finish."); ob1.t.join();
ob2.t.join();
} catch (InterruptedException e) { System.out.println("Main thread
Interrupted"); }
System.out.println("Main thread exiting.");}}
```


I/O Programming Basics

- **java.io**, provides support for I/O operations.
- Most programs cannot accomplish their goals without accessing external data. Data is retrieved from an *input* source. The results of a program are sent to an *output* destination.
- In Java, these sources or destinations are defined very broadly. For example, a network connection, memory buffer, or disk file can be manipulated by the Java I/O classes.
- Although physically different, these devices are all handled by the same abstraction: the *stream.* A stream, is a logical entity that either produces or consumes information.
- A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices they are linked to differ.
- *Java 2, version 1.4 includes some additional I/O capabilities which are contained in the **java.nio** package.*

Java I/O Classes

BufferedInputStream	File	PipedReader
BufferedOutputStream	FileDescriptor	PipedWriter
BufferedReader	FileInputStream	PrintStream
BufferedWriter	FileOutputStream	PrintWriter
BufferedArrayInputStream	FileReader	Reader
BufferedArrayOutputStream	FileWriter	StreamTokenizer
CharArrayReader	InputStream	StringReader
CharArrayWriter	InputStreamReader	StringWriter
DataInputStream	LineNumberReader	Writer
DataOutputStream	ObjectInputStream	

The ObjectInputStream.GetField and ObjectOutputStream.PutField inner classes were added by Java 2. The java.io package also contains two classes that were deprecated by Java 2 : LineNumberInputStream and StringBufferInputStream. These classes should not be used for new code.

Java I/O Interfaces

- The following interfaces are defined by **java.io:**

DataInput	FilenameFilter	ObjectOutput
DataOutput	ObjectInput	ObjectStreamConstants
Externalizable	ObjectInoutValidation	Serializable
FileFilter		

The FileFilter interface was added by Java 2.

Stream classes

- **Reader**, and **Writer classes** are used to create several concrete stream subclasses.
- Although your programs perform their I/O operations through concrete subclasses, the top-level classes define the basic functionality common to all stream classes.
- **InputStream** and **OutputStream** are designed for byte streams. **Reader** and **Writer** are designed for character streams.
- The byte stream classes and the character stream classes form separate hierarchies. In general, you should use the character stream classes when working with characters or strings, and use the byte stream classes when working with bytes or other binary objects.

The Byte Streams

- The byte stream classes provide a rich environment for handling byte-oriented I/O.
- A byte stream can be used with any type of object, including binary data. This versatility makes byte streams important to many types of programs. Since the byte stream classes are topped by **InputStream** and **OutputStream**, our discussion will begin with them.

InputStream

- **InputStream** is an abstract class that defines Java's model of streaming byte input. All of the methods in this class will throw an **IOException** on error conditions.

int available()	Returns an estimate of the number of bytes that can be read from this input stream without blocking
void close()	Closes this input stream and releases any system resources associated with the stream
void mark(int readlimit)	Marks the current position in this input stream
abstract int read()	Reads the next byte of data from the stream
void reset()	Repositions this stream to the position at the time the mark method was last called on this stream

OutputStream

- **OutputStream** is an abstract class that defines streaming byte output. All of the methods in this class return a **void** value and throw an **IOException** in case of error.

void close()	Closes this output stream and releases any system resources associated with this stream
void flush()	Flushes this output stream and forces any buffered output bytes to be written out.
void write (byte [] b)	Writes b.length bytes from the specified byte array to this output stream.
abstract void write (int b)	Writes the specified byte to this output stream

The Character Streams

- While the byte stream classes provide sufficient functionality to handle any type of I/O operation, they cannot work directly with Unicode characters. Since one of the main purposes of Java is to support the "write once, run anywhere" philosophy, it was necessary to include direct I/O support for characters. At the top of the character stream hierarchies are the **Reader** and **Writer** abstract classes.
- Reader :**Reader** is an abstract class that defines Java's model of streaming character input. All of the methods in this class will throw an **IOException** on error conditions.
- Writer: **Writer** is an abstract class that defines streaming character output. All of the methods in this class return a **void** value and throw an **IOException** in the case of errors.

Methods defined by Reader

abstract void close()	Closes the stream and releases any system resources associated with it.
void mark(int readAheadLimit)	Marks the present position in the stream
boolean markSupported()	Tells whether this stream supports the mark operation
int read()	Reads a single character
int read(char [] cbuf)	Reads characters into an array
boolean ready()	Tells whether this stream is ready to be read
void reset()	Resets the stream
long skip(long n)	Skips characters

Methods defined by Writer

Writer append (char c)	Appends the specified character to this Writer
abstract void close()	Closes this stream, flushing it first
abstract void flush()	Flushes the stream
void write(char[] cbuf)	Writes an array of characters
void write(int c)	Writes a single character
void write(String str)	Writes a string
void write(String str, int off, int len)	Writes a portion of a string

Predefined Stream

- 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.
- For example, using some of its methods, you can obtain the current time associated with the system. **System** also contains three predefined stream variables, **in**, **out**, and **err**. These fields are declared as **public** and **static** within **System**. This implies they can be used by any other part of your program, without reference to a **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. However, these streams may be redirected to any compatible I/O device.
- **System.in** is an object of type **InputStream**; **System.out** and **System.err** are objects of type **PrintStream**. These are byte streams, even though they typically are used to read and write characters from and to the console.

Reading Console Input

- In Java 1.0, the only way to perform console input was to use a byte stream, and older code that uses this approach persists.
- Today, using a byte stream to read console input is still technically possible, but may require the use of a deprecated method, and is not recommended.
- The preferred method of reading console input for Java 2 is to use a character-oriented stream, which makes your 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, you wrap **System.in** in a **BufferedReader** object, to create a character stream.
- **BuffereredReader** supports a buffered input stream. Its constructor is

BufferedReader(Reader *inputReader*)

Reading Console Input

- 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.
- To obtain 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*. Putting it all together, 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**.

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 the object referenced by **System.out**).
- Even though **System.out** is a byte stream, using it for simple program output is still acceptable. However, a character-based alternative is also available.
- Because **PrintStream** is an output stream derived from **OutputStream**, it also implements the low-level method **write( )**. Thus, **write( )** can be used to write to the console.

  void write(int *byteval*)
- This method writes to the stream the byte specified by *byteval*. Although *byteval* is declared as an integer, only the low-order eight bits are written. Here is a short example that uses **write( )** to output the character "A" followed by a newline to the screen:

```
class WriteDemo { public static void main(String args[]) { int b;b = 'A';
System.out.write(b); System.out.write('\n'); } }
```


The PrintWriter Class

- Using **System.out to write to the console is still permissible under Java,** its use is recommended for debugging purposes only.
- For real-world programs, the recommended method of writing to the console when using Java is through a **PrintWriter stream. PrintWriter** is one of the character-based classes.
- Using a character-based class for console output makes it easier to internationalize your 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, and flushOnNewline controls*** whether Java flushes the output stream every time a **println( ) method is called.**
- **If** *flushOnNewline is **true, flushing automatically takes place. If false, flushing is not*** automatic.

The PrintWriter Class

- **PrintWriter supports the print( ) and println( ) methods for all types including Object.**
- **Thus, you can use these methods in the same way as they have been used with System.out. If an argument is not a simple type, the PrintWriter methods call the** object's **toString( ) method and then print the result.**
- To write to the console by using a **PrintWriter, specify System.out for the output** stream and flush the stream after each newline.
- For example, this line of code creates a **PrintWriter that is connected to console output:**

```
PrintWriter pw = new PrintWriter(System.out, true);
```



```
// 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);  }}
```

Reading and Writing Files

- Java provides a number of classes and methods that allow you to read and write files.
- In Java, all files are byte-oriented, and Java provides methods to read and write bytes from and to a file. However, Java allows you to wrap a byte-oriented file stream within a character-based object.
- Two of the most often-used stream classes are **FileInputStream and FileOutputStream, which create byte streams linked to files.**
- **To open a file, you simply** create an object of one of these classes, specifying the name of the file as an argument to the constructor. While both classes support additional, overridden constructors, the most common forms are:

FileInputStream(String *fileName) throws FileNotFoundException*

FileOutputStream(String *fileName) throws FileNotFoundException*

Reading and Writing Files

- *In earlier versions of Java,* ***FileOutputStream( ) threw an IOException when an*** *output file could not be created. This was changed by Java 2.*
- When you are done with a file, we close it by calling **close( ). It is defined** by both **FileInputStream and FileOutputStream, as shown here:**

  void close( ) throws IOException
- To read from a file, you can use a version of **read( ) that is defined within FileInputStream.**
- int read( ) throws IOException
- Each time that it is called, it reads a single byte from the file and returns the byte as an integer value. **read( ) returns –1 when the end of the file is encountered. It can throw an IOException.**


```
• import java.io.*;
class ShowFile { public static void main(String args[]) throws
   IOException
{int i;  FileInputStream fin;
try { fin = new FileInputStream(args[0]);
} catch(FileNotFoundException e) {
System.out.println("File Not Found"); return;
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Usage: ShowFile File"); return; }
// read characters until EOF is encountered
do {i = fin.read();
if(i != -1) System.out.print((char) i);} while(i != -1);
fin.close();}}
```

Standard Java Packages (Core API)

- When Java 1.0 was released, it included a set of eight packages, called the *core API*. These are the packages you will use most often in your day-to-day programming.
- Each subsequent release added to the core API. Today, the Java API contains a large number of packages.
- We shall now have a brief look at the four most commonly used packages of Java:

java.lang

- **java.lang** is one of Java's most widely used package. It is automatically imported into all programs. It contains classes and interfaces that are fundamental to virtually all of Java programming.
- **java.lang** includes the following classes:

Boolean	Float	Process	StrictMath
Byte	Integer	ProcessBuilder	String
Character	Long	Runtime	StringBuffer
Class<T>	Math	RuntimePermission	StringBuilder
ClassLoader	Number	SecurityManager	System
Compiler	Object	Short	Thread
Double	Package	StackTraceElement	Throwable

In addition, there are two classes defined by Character: Character.Subset and Character.UnicodeBlock. These were added by Java 2.

java.lang

- **java.lang** also defines the following interfaces:
  - **Cloneable**
  - **Comparable**
  - **Runnable**
  - **CharSequence**
- The **Comparable** interface was added by Java 2. **CharSequence** was added by Java 2,version 1.4.
- Several of the classes contained in **java.lang** contain deprecated methods, most dating back to Java 1.0. These deprecated methods are still provided by Java 2, to support an ever-shrinking pool of legacy code, and are not recommended for new code.
- Most of the deprecations took place prior to Java 2.

java.util

- The **java.util package contains one of Java's most powerful subsystems: collections.**
- Collections were added by the initial release of Java 2, and enhanced by Java 2, version 1.4. A *collection is a group of objects.*
- *The addition of collections caused* fundamental alterations in the structure and architecture of many elements in **java.util.**
- In addition to collections, **java.util contains a wide assortment of classes and** interfaces that support a broad range of functionality.
- These classes and interfaces are used throughout the core Java packages and, of course, are also available for use in programs that you write. Their applications include generating pseudorandom numbers, manipulating date and time, observing events, manipulating sets of bits, and tokenizing strings. Because of its many features, **java.util is one of Java's most widely used packages.**

The java.util classes are listed here

AbstractCollection	Currency	Locale
AbstractList	Date	Observable
AbstractMap	Dictionary	PriorityQueue
AbstractQueue	EnumMap	Properties
AbstractSet	EnumSet	Random
ArrayDeque	EventObject	Scanner
ArrayList	Formatter	Stack
Arrays	HashMap	StringTokenizer
BitSet	HashSet	Timer
Calender	Hashtable	TreeMap
Collections	LinkedList	TreeSet

java.util interfaces

Collection	Iterator	Observer
Comparator	List	Queue
Deque	ListIterator	RandomAccess
Enumeration	Map	Set
EventListener	NavigableMap	SortedMap
Formattable	NavigableSet	SortedSet

java.io

- **java.io, provides support for I/O operations.**
- As all programmers learn early on, most programs cannot accomplish their goals without accessing external data.
- Data is retrieved from an *input source. The results of* a program are sent to an *output destination. In Java, these sources or destinations are* defined very broadly. For example, a network connection, memory buffer, or disk file can be manipulated by the Java I/O classes.
- Although physically different, these
- devices are all handled by the same abstraction: the *stream. A stream,* is a logical entity that either produces or consumes information.
- A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices they are linked to differ.

 U1.

Java I/O classes

BufferedInputStream	CharArrayWriter	FileOutputStream
BufferedOutputStream	Console	FileReader
BufferedReader	DataInputStream	FileWriter
BufferedWriter	DataOutputStream	InputStream
ByteArrayInputStream	File	OutputStream
ByteArrayOutputStream	FileDescriptor	PrintStream
CharArrayReader	FileInputStream	PrintWriter

The ObjectInputStream.GetField and ObjectOutputStream.PutField inner classes were added by Java 2. The java.io package also contains two classes that were deprecated by Java 2 and are not shown in the preceding table: LineNumberInputStream and StringBufferInputStream. These classes should not be used for new code.

 U1.

Interfaces of **java.io**

Closeable	Flushable
DataInput	ObjectInput
DataOutput	ObjectInputValidator
Externalizable	ObjectOutput
FileFilter	ObjectStreamConstant
FileNameFilter	Serializable

 U1.

java.net

- The **java.net package, provides support for** networking. Its creators called Java "programming for the Internet."
- What makes Java a good language for networking are the classes defined in the **java.net package.**
- These networking classes encapsulate the "socket" paradigm pioneered in the Berkeley Software Distribution (BSD) from the University of California at Berkeley.
- Java supports TCP/IP both by extending the already established stream I/O interface and by adding the features required to build I/O objects across the network. Java supports both the TCP and UDP protocol families. TCP is used for reliable stream-based I/O across the network. UDP supports a simpler, hence faster, point-to-point datagram-oriented model.

Classes in the **java.net package**

Authenticator	HttpCookie	MulticastSocket
CacheRequest	HttpURLConnection	NetPermission
CacheResponse	Inet4Address	NetworkInterface
ContentHandler	Inet6Address	Proxy
DatagramPacket	InetAddress	ServerSocket
DatagramSocket	InetSocketAddress	Socket

As you can see, several new classes were added by Java 2, version 1.4. Some of these are to support the new IPv6 addressing scheme. Others provide some added flexibility to the original java.net package. Java 2, version 1.4 also added functionality, such as support for the new I/O classes, to several of the preexisting networking classes. Some of the additions made by Java 2, version 1.4 are Inet4Address, Inet6Address, and URI.

java.net Interfaces

ContentHandlerFactory	FileNameMap
CookiePolicy	SocketImplFactory
CookieStore	SocketOptions
DatagramSocketImplFactory	URLStreamHandlerFactory