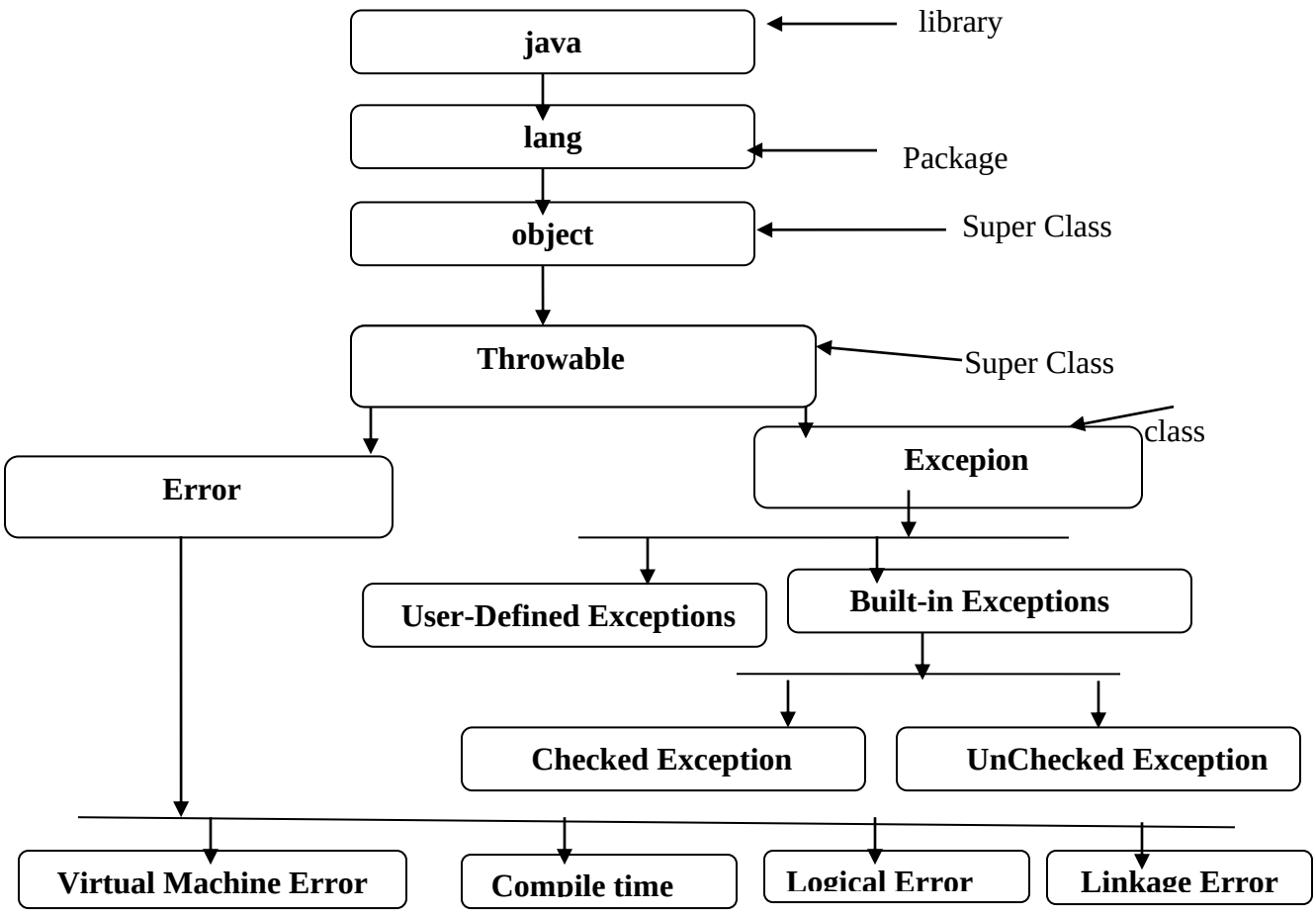
Unit-III

Exception Handling

Exception Hierarchy:



Java: JAVA API is a library contains the packages. These were developed by the JavaSoft people of Sun MicroSystems Inc. used to import the classes in developing the programs.

Lang: lang is a package included in java library. And it is considered as a default package named as language. Implicitly it is imported into every java programs.

Object: Object is a super class of all classes(user defined, pre-defined classes) directly or indirectly. Because it is included in the lang package.

Throwable: Throwable is super class of errors and exceptions in java. Throwable is deriving from the object class.

Error: Error is a class. This is not handled. We known the error in program after the compilation denoted by the java compiler. Always these were detected at compile time.

An error in a program is called bug. Removing errors from program is called debugging. There are basically three types of errors in the Java program:

- Compile time errors: Errors which occur due to syntax or format is called compile time errors. These errors are detected by java compiler at compilation time. Desk checking is solution for compile-time errors.

Example:

```
    import java.io.*;
class Compile
{
static public void main(String args[])
{
    System.out.println("hello")
}
}
```

Output:

```
Compile.java:16 ';' expected
System.out.println("hello")^
1 error
```

- Logical errors: These are the errors that occur due to bad logic in the program. These errors are rectified by comparing the outputs of the program manually.

Example:

```
class Salary
{
public static void main(String args[])
{
double sal=5000.00;
sal=sal*15/100; //use:sal+=sal*15/100;
System.out.println("incremented salary:"+sal);
}
}
```

Output: java Salary

Incremented salary:750.0

Exception: An abnormal event in a program is called Exception.

- Exception may occur at compile time or at runtime.
- Exceptions which occur at compile time are called Checked exceptions.

Checked Exceptions:

- A checked exception is any subclass of Exception (or Exception itself), excluding class RuntimeException and its subclasses.
- You should compulsorily handle the checked exceptions in your code, otherwise your code will not be compiled. i.e you should put the code which may cause checked exception in try block. "checked" means they will be checked at compiletime itself.
- There are two ways to handle checked exceptions. You may declare the exception using a throws clause or you may use the try..catch block.
- The most perfect example of Checked Exceptions is IOException which should be handled in your code Compulsorily or else your Code will throw a Compilation Error.

e.g.: ClassNotFoundException, NoSuchMethodException, NoSuchFieldException etc

```
import java.io.*;

class Sample

{

void accept( ) throws IOException

  {

   BufferedReader br=new BufferedReader (new InputStreamReader(System.in));

    System.out.print ("enter ur name: ");

    String name=br.readLine ( );

    System.out.println ("Hai "+name);

  }

}

class ExceptionNotHandle

{

 public static void main (String args[])

  {

    Sample s=new Sample ( );

    s.accept ( );

  }

}
```

Output: javac ExceptionNotHandle.java

ExceptionNotHandle.java:16: unreported exception java.io.IOException must be caught or declared to be thrown

s.accept();^

1 error

- Exceptions which occur at run time are called Unchecked exceptions.

Unchecked Exceptions :

- Unchecked exceptions are RuntimeException and any of its subclasses. Class Error and its subclasses also are unchecked.
- Unchecked runtime exceptions represent conditions that, generally speaking, reflect errors in your program's logic and cannot be reasonably recovered from at run time.
- With an unchecked exception, however, compiler doesn't force client programmers either to catch the exception or declare it in a throws clause.
- The most Common examples are ArrayIndexOutOfBoundException, NUllPointerException ,ClassCastException

eg: ArrayIndexOutOfBoundsException, ArithmeticException, NumberFormatException etc.

Example:

```
public class V
{
static public void main(String args[])
{
int d[]={1,2};
d[3]=99;
int a=5,b=0,c;
c=a/b;
System.out.println("c is:"+c);
System.out.println("okay");
}
}
```

Output:

Exception in thread “main” java.lang.ArrayInde xOutOfBoundsException: 3

At V.main (V.java:6)

Concepts of Exception Handling:

exception is an abnormal condition that arises during the execution of a program that disrupts the normal flow of execution.

Error: When a dynamic linking failure or some other “hard” failure in the virtual machine occurs, the virtual machine throws an Error.

Java exception handling is managed via by five keywords: try, catch, throw, throws, finally.

Try: The try block is said to govern the statements enclosed within it and defines the scope of any exception associated with it. It detects the exceptions.

Catch: The catch block contains a series of legal Java statements. These statements are executed if and when the exception handler is invoked. It holds an exception.

Throw: To manually throw an exception ,use the keyword throw.

Throws: Any exception that is thrown out of a method must be specified as such by a throws clause.

Finally: Any code that absolutely must be executed after a try block completes is put in a finally block. After the exception handler has run, the runtime system passes control to the finally block.

General form of an exception handling:

```
try
{
  //block of code to monitor for errors
}
catch(ExceptionType exOb)
{
  //exception handler for ExceptionType
}
//...
finally
{
  //block of code to be executed after try block ends
}
```

Example:

```
public class ExceptionDemo
{
   public static void main(String args[])throws IOException
   {
      int subject[]={12,23,34,21};
      try
```

```
    {
      System.out.println(subject[2]);
      System.out.println("not okay");
    }
    catch(ArrayIndexOutOfBoundException e)
   {
     System.out.println("i caught the exception:"+e);
      throw e;
    }
  finally
  {
    System.out.println("okay");
   }
 }
}
```

Output:

34

Not Okay

okay

Benefits of Exception Handling:

- First, it allows you to fix the error.
- Second, it prevents the program from automatically terminating.
- Third, it adopts the robustness to program.

Termination or Resumptive Models:

The first question that arises, is how or, in particular, where to indicate resumption. Basically, there are only two possibilities:

Firstly, the decision whether to resume or notcan be made at the raising point, i.e. by the raise statement itself. This implies that a language would have to offer two different raise statements: one for the termination model and another one for resumption, i.e. where the handler always “returns” and resumes execution at the raising point.

The main advantage of this possibility is, that there is no doubt about the continuation of the control flow. In particular, it is already known in the raising context, whether a handler will resume or not.

But is this feasible?

Usually only after having tried to cure the cause of an exception, we can say, whether the attempt was successful or not. Therefore, only the handler of an exception can decide, whether it could cure the cause for an exception or not. this knowledge is essential, because resumption only makes sense with the motivation to cure the cause for the exception before resuming normal execution.

Therefore, we suggest, that the respective handler should indicate, whether to terminate or to resume.

```
public void a() {

try { b(); }

catch (Exception1 e1) { ..... }

catch (Exception2 e2) {

/* Try to cure the cause. */

if (error_is_curable)

resume new Solution("the solution");

else { /*Clean up and proceed*

*as with termination.*/ } }

public void b () throws Exception2 {

.....

throw new Exception2("Caused by error")

accept (Solution s1) { ..... }

accept (AnotherSolution s2) { ..... }

..... }
```

Fig. 1. A simple resumption scenario demonstrating the new syntax.

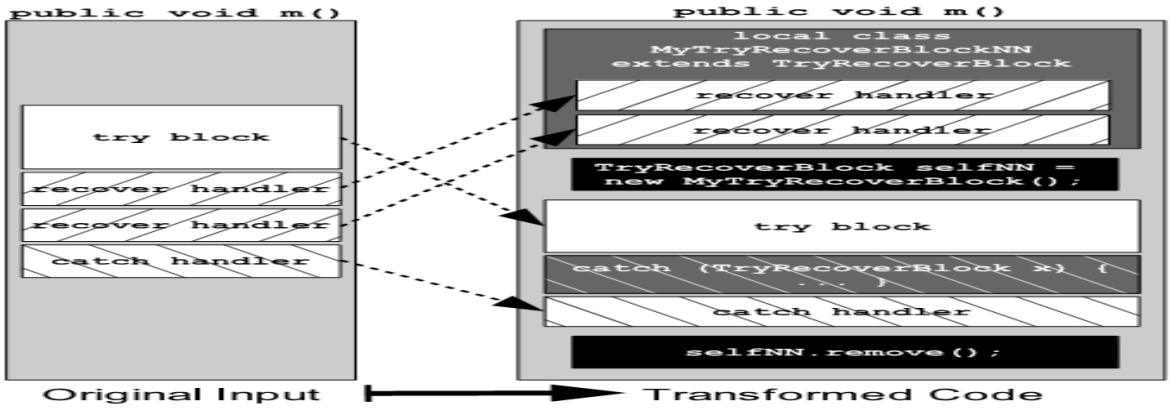


Fig. 2. Sketch of the basic transformation of a try–recover construct by the precompiler.

Usage of try, catch, throw, throws, finally:

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. To illustrate how easily this can be done, 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.");

}

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

}

}
```

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 exceptionis 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. Once the catch statement has executed, program control continues with the next line in the program following the entire try/catch mechanism.

A try and its catch statement form a unit. The scope of the catch clause is restricted to those statements specified by the immediately preceding try statement. A catch statement cannot catch an exception thrown by another try statement (except in the case of nested try statements, described shortly).

Note: The statements that are protected by try must be surrounded by curly braces. (That is, they must be within a block.) You cannot use try on a single statement.

The goal of most well-constructed catch clauses should be to resolve the exceptional condition and then continue on as if the error had never happened. For example, in the next program each iteration of the for loop obtains two random integers. Those two integers are divided by each other, and the result is used to divide the value 12345. The final result is put into a. If either division operation causes a divide-by-zero error, it is caught, the value of a is set to zero, and the program continues.

```
// Handle an exception and move on.

import java.util.Random;

class HandleError {
```

```
public static void main(String args[]) {

int a=0, b=0, c=0;

Random r = new Random();

for(int i=0; i<32000; i++) {

try {

b = r.nextInt();

c = r.nextInt();

a = 12345 / (b/c);

} catch (ArithmeticException e) {

System.out.println("Division by zero.");

a = 0; // set a to zero and continue

}

System.out.println("a: " + a);

}

}

}
```

Displaying a 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 type of 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. The following example traps two different exception types:

```
// 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.");

}

}
```

This program will cause a division-by-zero exception if it is started with no command-line arguments, since a will equal zero. It will survive the division if you provide a command-line argument, setting a to something larger than zero. But it will cause an ArrayIndexOutOfBoundsException, since the int array c has a length of 1, yet the program attempts to assign a value to c[42].

Here is the output generated by running it both ways:

```
C:\>java MultiCatch

a = 0

Divide by 0: java.lang.ArithmeticException: / by zero

After try/catch blocks.

C:\>java MultiCatch TestArg

a = 1

Array index oob: java.lang.ArrayIndexOutOfBoundsException:42

After try/catch blocks.
```

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. Further, in Java, unreachable code is an error. For example, consider the following program:

```
/* This program contains an error. A subclass must come before its superclass in a series of catch statements. If not, unreachable code will be created and a compile-time error will result.*/

class SuperSubCatch {

public static void main(String args[]) {

try {

int a = 0;

int b = 42 / a;

} catch(Exception e) {

System.out.println("Generic Exception catch.");

}

/* This catch is never reached because

ArithmeticException is a subclass of Exception. */

catch(ArithmeticException e) { // ERROR - unreachable

System.out.println("This is never reached.");

}

}

}
```

If you try to compile this program, you will receive an error message stating that the second catch statement is unreachable because the exception has already been caught. Since ArithmeticException is a subclass of Exception, the first catch statement will handle all Exception-based errors, including ArithmeticException. This means that the second catch statement will never execute. To fix the problem, reverse the order of the catch statements.

Nested try Statements

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. Here is an example that uses nested try statements:

```
// An example of nested try statements.

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);
}
}
}
```

As you can see, this program nests one try block within another. The program works as follows.When you execute the program with no command-line arguments, a divide-by-zero exception is generated by the outer try block. Execution of the program with one command-line argument generates a divide-by-zero exception from within the nested try block. Since the inner block does not catch this exception, it is passed on to the outer try block, where it is handled. If you execute the program with two command-line arguments, an array boundary exception is generated from within the inner try block. Here are sample runs that illustrate each case:

```
C:\>java NestTry

Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One

a = 1

Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One Two

a = 2

Array index out-of-bounds:

java.lang.ArrayIndexOutOfBoundsException:42
```

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. Here is the previous program recoded so that the nested try block is moved inside the method nesttry( ):

```
/* Try statements can be implicitly nested via

calls to methods. */

class MethNestTry {

static void nesttry(int 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);

}
```

```
}

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);

nesttry(a);

} catch(ArithmeticException e) {

System.out.println("Divide by 0: " + e);

}

}

}
```

The output of this program is identical to that of the preceding example.

throw

So far, you have only been catching exceptions that are thrown by the Java run-time system. However, it is 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. Primitive types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions.

There are two ways you can obtain a Throwable object:

- using a parameter in a catch clause,
- or creating one with the new operator.

The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

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);

}

}

}
```

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

```
Caught inside demoproc.

Recaught: java.lang.NullPointerException: demo
```

The program also illustrates how to create one of Java's standard exception objects. Pay close attention to this line:

```
throw new 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( ). It can also be obtained by a call to getMessage( ), which is defined by Throwable.

throws

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a throws clause in the method's declaration. A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses. All

other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result.

This is the general form of a method declaration that includes a throws clause:

type method-name(parameter-list) throws exception-list

{

// body of method

}

Here, exception-list is a comma-separated list of the exceptions that a method can throw. Following is an example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a throws clause to declare this fact, the program will not compile.

```
// This program contains an error and will not compile.

class ThrowsDemo {

static void throwOne() {

System.out.println("Inside throwOne.");

throw new IllegalAccessException("demo");

}

public static void main(String args[]) {

throwOne();

}

}
```

To make this example compile, you need to make two changes.

- First, you need to declare that throwOne( ) throws IllegalAccessException.
- Second, main( ) must define a try/catch statement that catches this exception.

The corrected example is shown here:

```
// This is now correct.

class ThrowsDemo {

static void throwOne() throws IllegalAccessException {

System.out.println("Inside throwOne.");

throw new IllegalAccessException("demo");

}

public static void main(String args[]) {

try {

throwOne();
```

```
} catch (IllegalAccessException e) {

System.out.println("Caught " + e);

}

}

}
```

Here is the output generated by running this example program:

```
inside throwOne

caught java.lang.IllegalAccessException: demo
```

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.

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 statementexecutes 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

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

Built in 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.

TABLE 10-2 Java's Checked Exceptions Defined in **java.lang**

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.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	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.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

TABLE 10-1 Java's Unchecked **RuntimeException** Subclasses Defined in **java.lang**

Creating own Exception Sub Classes:

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. They are shown in Table 10-3.

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**. 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. The **StackTraceElement** class gives your program access to information about each element in the trace, such as its method name.
Throwable initCause(Throwable *causeExc*)	Associates *causeExc* with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.
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.
String toString( )	Returns a **String** object containing a description of the exception. This method is called by **println( )** when outputting a **Throwable** object.

TABLE 10-3 The Methods Defined by **Throwable**

You may also wish to override one or more of these methods in exception classes that you create.

Exception defines four constructors. Two were added by JDK 1.4 to support chained exceptions, described in the next section. The other two are shown here:

Exception( )

Exception(String msg)

The first form creates an exception that has no description. The second form lets you specify a description of the exception. Although specifying a description when an exception is created is often useful, sometimes it is better to override toString( ). Here's why: The version of toString( ) defined by Throwable (and inherited by Exception) first displays the name of the exception followed by a colon, which is then followed by your description. By overriding toString( ), you can prevent the exception name and colon from being displayed. This makes for a cleaner output, which is desirable in some cases.

The following example declares a new subclass of Exception and then uses that subclass to signal an error condition in a method. It overrides the toString( ) method, allowing a carefully tailored description of the exception to be displayed.

```
// 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 result:

```
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```

Chained Exceptions

Beginning with JDK 1.4, a new feature has been incorporated into 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, two constructors and two methods were added to Throwable.

The constructors are shown here:

Throwable(Throwable causeExc)

Throwable(String msg, Throwable causeExc)

In the first form, causeExc is the exception that causes the current exception. That is, causeExc is the underlying reason that an exception occurred. The second form allows you to specify a description at the same time that you specify a cause exception. These two constructors have also been added to the Error, Exception, and RuntimeException classes.

The chained exception methods added to Throwable are getCause( ) and initCause( ).

These methods are shown in Table 10-3 and are repeated here for the sake of discussion.

Throwable getCause( )

Throwable initCause(Throwable causeExc)

The getCause( ) method returns the exception that underlies the current exception. If there is no underlying exception, null is returned. The initCause( ) method associates causeExc with the invoking exception and returns a reference to the exception. Thus, you can associate a cause with an exception after the exception has been created. However, the cause exception can be set only once. Thus, you can call initCause( ) only once for each exception object.

Furthermore, if the cause exception was set by a constructor, then you can’t set it again using initCause( ). In general, initCause( ) is used to set a cause for legacy exception classes that don’t support the two additional constructors described earlier. Here is an example that illustrates the mechanics of handling chained exceptions:

String Handling in Java :

The String class is defined in the java.lang package and hence is implicitly available to all the programs in Java. The String class is declared as final, which means that it cannot be subclassed. It extends the Object class and implements the Serializable, Comparable, and CharSequence interfaces.

Java implements strings as objects of type String. A string is a sequence of characters. Unlike most of the other languages, Java treats a string as a single value rather than as an array of characters.

The String objects are immutable, i.e., once an object of the String class is created, the string it contains cannot be changed. In other words, once a String object is created, the characters that comprise the string cannot be changed. Whenever any operation is performed on a String object, a new String object will be created while the original contents of the object will remain unchanged. However, at any time, a variable declared as a String reference can be changed to point to some other String object.

Why String is immutable in Java

Though there could be many possible answer for this question and only designer of String class can answer this, I think below three does make sense

1) Imagine StringPool facility without making string immutable, its not possible at all because in case of string pool one string object/literal e.g. "Test" has referenced by many reference variables , so if any one of them change the value others will be automatically gets affected i.e. lets say

String A = "Test"
String B = "Test"

Now String B called "Test".toUpperCase() which change the same object into "TEST" , so A will also be "TEST" which is not desirable.

2) String has been widely used as parameter for many java classes e.g. for opening network connection you can pass hostname and port number as stirng , you can pass database URL as string for opening database connection, you can open any file by passing name of file as argument to File I/O classes.

In case if String is not immutable, this would lead serious security threat , I mean some one can access to any file for which he has authorization and then can change the file name either deliberately or accidentally and gain access of those file.

3) Since String is immutable it can safely shared between many threads, which is very important for multithreaded programming.

String Vs StringBuffer and StringBuilder

String

Strings: A String represents group of characters. Strings are represented as String objects in java.

Creating Strings:

- We can declare a String variable and directly store a String literal using assignment operator.

  String str = "Hello";

- We can create String object using new operator with some data.

String s1 = new String ("Java");

- We can create a String by using character array also.

  char arr[] = { 'p','r','o','g','r','a','m'};

- We can create a String by passing array name to it, as:

  String s2 = new String (arr);

- We can create a String by passing array name and specifying which characters we need:

  String s3 = new String (str, 2, 3);

Here starting from 2nd character a total of 3 characters are copied into String s3.

String Class Methods:

Method	Description
String concat (String str)	Concatenates calling String with str. **Note:** + also used to do the same
int length ()	Returns length of a String
char charAt (int index)	Returns the character at specified location (from 0)
int compareTo (String str)	Returns a negative value if calling String is less than str, a positive value if calling String is greater than str or 0 if Strings are equal.
boolean equals (String str)	Returns true if calling String equals str. **Note:** == operator compares the references of the string objects. It does not compare the contents of the objects. equals () method compares the contents. While comparing the strings, equals () method should be used as it yields the correct result.
boolean equalsIgnoreCase (String str)	Same as above but ignores the case
boolean startsWith (String prefix)	Returns true if calling String starts with prefix
boolean endsWith (String suffix)	Returns true if calling String ends with suffix
int indexOf (String str)	Returns first occurrence of str in String.
int lastIndexOf(String str)	Returns last occurrence of str in the String. **Note:** Both the above methods return negative value, if str not found in calling String. Counting starts from 0.
String replace (char oldchar, char newchar)	returns a new String that is obtained by replacing all characters oldchar in String with newchar.
String substring (int beginIndex)	returns a new String consisting of all characters from beginIndex until the end of the String
String substring (int beginIndex, int endIndex)	returns a new String consisting of all characters from beginIndex until the endIndex.
String toLowerCase ()	converts all characters into lowercase
String toUpperCase ()	converts all characters into uppercase
String trim ()	eliminates all leading and trailing spaces

String represents a sequence of characters. It has fixed length of character sequence. Once a string object has been created than we can't change the character that comprise that string. It is immutable. This allows String to be shared. String object can be instantiated like any other object

String str = new String ("Stanford “);
str += "Lost!!";

Accessor methods: length(), charAt(i), getBytes(), getChars(istart,iend,gtarget[],itargstart), split(string,delim), toCharArray(), valueOf(g,iradix), substring(iStart [,iEndIndex)]) [returns up to but not including iEndIndex]

Modifier methods: concat(g), replace(cWhich, cReplacement), toLowerCase(), toUpperCase(), trim().

Boolean test methods: contentEquals(g), endsWith(g), equals(g), equalsIgnoreCase(g), matches(g), regionMatches(i1,g2,i3,i4), regionMatches(bIgnoreCase,i1,g2,i3,i4), startsWith(g)

Integer test methods: compareTo(g) [returns 0 if object equals parameter, -1 if object is before parameter in sort order, +1 if otherwise], indexOf(g) [returns position of first occurrence of substring g in the string, -1 if not found], lastIndexOf(g) [returns position of last occurrence of substring g in the string, -1 if not found], length().

Constructors defined in the String class

The String class defines several constructors. The most common constructor of the String class is the one given below:

public String(String value)

This constructor constructs a new String object initialized with the same sequence of the characters passed as the argument. In other words, the newly created String object is the copy of the string passed as an argument to the constructor.

Other constructors defined in the String class are as follows:

public String()

This constructor creates an empty String object. However, the use of this constructor is unnecessary because String objects are immutable.

public String(char[] value)

This constructor creates a new String object initialized with the same sequence of characters currently contained in the array that is passed as the argument to it.

public String(char[] value, int startindex, int len)

This constructor creates a new String object initialized with the same sequence of characters currently contained in the subarray. This subarray is derived from the character array and the two integer values that are passed as arguments to the constructor. The int variable startindex represents the index value of the starting character of the subarray, and the int variable len represents the number of characters to be used to form the new String object.

public String(StringBuffer sbf)

This constructor creates a new String object that contains the same sequence of characters currently contained in the string buffer argument.

public String(byte[] asciichars)

The array of bytes that is passed as an argument to the constructor contains the ASCII character set. Therefore, this array of bytes is first decoded using the default charset of the platform. Then the constructor creates a new String object initialized with same sequence of characters obtained after decoding the array.

public String(byte[] asciiChars, int startindex, int len)

This constructor creates the String object after decoding the array of bytes and by using the subarray of bytes.

Special String Operations

Finding the length of string

The String class defines the length() method that determines the length of a string. The length of a string is the number of characters contained in the string. The signature of the length() method is given below:

public int length()

String Concatenation using the + operator

The + operator is used to concatenate two strings, producing a new String object as the result. For example,

```
String sale = "500";
String s = "Our daily sale is" + sale + "dollars";
System.out.println(s);
```

This code will display the string "Our daily sale is 500 dollars".

The + operator may also be used to concatenate a string with other data types. For example,

```
int sale = 500;
String s = "Our daily sale is" + sale + "dollars";
System.out.println(s);
```

This code will display the string "Our daily sale is 500 dollars". In this case, the variable sale is declared as int rather than String, but the output produced is the same. This is because the int value contained in the variable sale is automatically converted to String type, and then the + operator concatenates the two strings.

String Comparison

The String class defines various methods that are used to compare strings or substrings within strings. Each of them is discussed in the following sections:

Note: Since strings are stored as a memory address, the == operator can't be used for comparisons. Use equals() and equalsIgnoreCase() to do comparisons. A simple example is:

equals()

The equals() method is used to check whether the Object that is passed as the argument to the method is equal to the String object that invokes the method. It returns true if and only if the argument is a String object that represents the same sequence of characters as represented by the invoking object. The signature of the equals() method is as follows:

public boolean equals(Object str)

equalsIgnoreCase()

The equalsIgnoreCase() method is used to check the equality of the two String objects without taking into consideration the case of the characters contained in the two strings. It returns true if the two strings are of the same length and if the corresponding characters in the two strings are the same ignoring case. The signature of the equalsIgnoreCase() method is:

public boolean equalsIgnoreCase(Object str)

compareTo()

The compareTo() method is used in conditions where a Programmer wants to sort a list of strings in a predetermined order. The compareTo() method checks whether the string passed as an argument to the method is less than, greater than, or equal to the invoking string. A string is considered less than another string if it comes before it in alphabetical order. The signature of the compareTo() method is as follows:

public int compareTo(String str)

where, str is the String being compared to the invoking String. The compareTo() method returns an int value as the result of String comparison. The meaning of these values are given in the following table:

The String class also has the compareToIgnoreCase() method that compares two strings without taking into consideration their case difference. The signature of the method is given below:

public int compareToIgnoreCase(String str)

regionMatches()

The regionMatches() method is used to check the equality of two string regions where the two string regions belong to two different strings. The signature of the method is given below:

public boolean regionMatches(int startindex, String str2, int startindex2, int len)

There is also an overloaded version of the method that tests the equality of the substring ignoring the case of characters in the substring. Its signature is given below:

public boolean regionMatches(boolean ignoreCase, int startindex, String str2, int startindex2, int len)

In both signatures of the method, startindex specifies the starting index of the substring within the invoking string. The str2 argument specifies the string to be compared. The startindex2 specifies the starting index of the substring within the string to be compared. The len argument specifies the length of the substring being compared. However, in the latter signature of the method, the comparison is done ignoring the case of the characters in the substring only if the ignoreCase argument is true.

startsWith()

The startsWith() method is used to check whether the invoking string starts with the same sequence of characters as the substring passed as an argument to the method. The signature of the method is given

below:

public boolean startsWith(String prefix)

There is also an overloaded version of the startsWith() method with the following signature:

public boolean startsWith(String prefix, int startindex)

In both signatures of the method given above, the prefix denotes the substring to be matched within the invoking string. However, in the second version, the startindex denotes the starting index into the invoking string at which the search operation will commence.
endsWith()

The endsWith() method is used to check whether the invoking string ends with the same sequence of characters as the substring passed as an argument to the method. The signature of the method is given below:

public boolean endsWith(String prefix)

Modifying a String

The String objects are immutable. Therefore, it is not possible to change the original contents of a string. However, the following String methods can be used to create a new copy of the string with the required modification:

substring()

The substring() method creates a new string that is the substring of the string that invokes the method. The method has two forms:

public String substring(int startindex)
public String substring(int startindex, int endindex)

where, startindex specifies the index at which the substring will begin and endindex specifies the index at which the substring will end. In the first form where the endindex is not present, the substring begins at startindex and runs till the end of the invoking string.

Concat()

The concat() method creates a new string after concatenating the argument string to the end of the invoking string. The signature of the method is given below:

public String concat(String str)

replace()

The replace() method creates a new string after replacing all the occurrences of a particular character in the string with another character. The string that invokes this method remains unchanged. The general form of the method is given below:

public String replace(char old_char, char new_char)

trim()

The trim() method creates a new copy of the string after removing any leading and trailing whitespace. The signature of the method is given below:

public String trim(String str)

toUpperCase()

The toUpperCase() method creates a new copy of a string after converting all the lowercase letters in the invoking string to uppercase. The signature of the method is given below:

public String toUpperCase()

toLowerCase()

The toLowerCase() method creates a new copy of a string after converting all the uppercase letters in the invoking string to lowercase. The signature of the method is given below:

public String toLowerCase()

Searching Strings

The String class defines two methods that facilitate in searching a particular character or sequence of characters in a string. They are as follows:

IndexOf()

The indexOf() method searches for the first occurrence of a character or a substring in the invoking string. If a match is found, then the method returns the index at which the character or the substring first appears. Otherwise, it returns -1.

The indexOf() method has the following signatures:

public int indexOf(int ch)
public int indexOf(int ch, int startindex)
public int indexOf(String str)
public int indexOf(String str, int startindex)

lastIndexOf()

The lastIndexOf() method searches for the last occurrence of a character or a substring in the invoking string. If a match is found, then the method returns the index at which the character or the substring last appears. Otherwise, it returns –1.

The lastIndexOf() method has the following signatures:

public int lastIndexOf(int ch)
public int lastIndexOf (int ch, int startindex)
public int lastIndexOf (String str)
public int lastIndexOf (String str, int startindex)

Program : Write a program using some important methods of String class.

```
// program using String class methods
class StrOps
{   public static void main(String args [])
{  String str1 = "When it comes to Web programming, Java is #1.";
String str2 = new String (str1);
String str3 = "Java strings are powerful.";
int result, idx;    char ch;
System.out.println ("Length of str1: " + str1.length ());
 // display str1, one char at a time.
for(int i=0; i < str1.length(); i++)
System.out.print (str1.charAt (i));
System.out.println ();
if (str1.equals (str2) )
System.out.println ("str1 equals str2");
else
System.out.println ("str1 does not equal str2");
if (str1.equals (str3) )
 System.out.println ("str1 equals str3");
else
System.out.println ("str1 does not equal str3");
result = str1.compareTo (str3);
if(result == 0)
System.out.println ("str1 and str3 are equal");
else if(result < 0)
System.out.println ("str1 is less than str3");
else
```

```
System.out.println ("str1 is greater than str3");

str2 = "One Two Three One";     // assign a new string to str2

idx = str2.indexOf ("One");

System.out.println ("Index of first occurrence of One: " + idx);

idx = str2.lastIndexOf("One");

System.out.println ("Index of last occurrence of One: " + idx);

}

}
```

Output:

StringBuffer

StringBuffer: StringBuffer objects are mutable, so they can be modified. The methods that directly manipulate data of the object are available in StringBuffer class.

Creating StringBuffer:

- We can create a StringBuffer object by using new operator and pass the string to the object, as: StringBuffer sb = new StringBuffer ("Kiran");
- We can create a StringBuffer object by first allotting memory to the StringBuffer object using new operator and later storing the String into it as:

```
StringBuffer sb = new StringBuffer (30);
```

In general a StringBuffer object will be created with a default capacity of 16 characters. Here, StringBuffer object is created as an empty object with a capacity for storing 30 characters. Even if we declare the capacity as 30, it is possible to store more than 30 characters into StringBuffer.

To store characters, we can use append () method as:

Sb.append ("Kiran");

This represents growable and writeable character sequence. It is mutable in nature. StringBuffer are safe to be used by multiple thread as they are synchronized but this brings performance penalty.
It defines 3-constructor:
• StringBuffer(); //initial capacity of 16 characters
• StringBuffer(int size); //The initial size
• StringBuffer(String str);

StringBuffer str = new StringBuffer ("Stanford ");
str.append("Lost!!");

StringBuffer Class Methods:

Method	Description
StringBuffer append (x)	x may be int, float, double, char, String or StringBuffer. It will be appended to calling StringBuffer
StringBuffer insert (int offset, x)	x may be int, float, double, char, String or StringBuffer. It will be inserted into the StringBuffer at offset.
StringBuffer delete (int start, int end)	Removes characters from start to end
StringBuffer reverse ()	Reverses character sequence in the StringBuffer
String toString ()	Converts StringBuffer into a String
int length ()	Returns length of the StringBuffer

Program : Write a program using some important methods of StringBuffer class.

```
// program using StringBuffer class methods

import java.io.*;

class Mutable

{  public static void main(String[] args) throws IOException

  {  // to accept data from keyboard

  BufferedReader br=new BufferedReader (new InputStreamReader (System.in));

    System.out.print ("Enter sur name : ");

    String sur=br.readLine ( );

    System.out.print ("Enter mid name : ");

    String mid=br.readLine ( );

    System.out.print ("Enter last name : ");
```

```
    String last=br.readLine ( );
    // create String Buffer object
    StringBuffer sb=new StringBuffer ( );
    // append sur, last to sb
    sb.append (sur);
    sb.append (last);
    // insert mid after sur
    int n=sur.length ( );
    sb.insert (n, mid);
    // display full name
    System.out.println ("Full name = "+sb);
    System.out.println ("In reverse ="+sb.reverse ( ));
  }
}
```

Output**:**

Accessor methods: capacity(), charAt(i), length(), substring(iStart [,iEndIndex)])

Modifier methods: append(g), delete(i1, i2), deleteCharAt(i), ensureCapacity(), getChars(srcBeg, srcEnd, target[], targetBeg), insert(iPosn, g), replace(i1,i2,gvalue), reverse(), setCharAt(iposn, c), setLength(),toString(g)

So the basic differences are…..

1. String is immutable but StringBuffer is not.

2. String is not threadsafe but StringBuffer is thread safe

3. String has concat() for append character but StringBuffer has append() method

4. while you create String like String str = new String(); it create 2 object 1 on heap and 1 on String Constant pool and that refered by str but in StringBuffer it Create 1 object on heap

StringBuilder

StringBuilder class is introduced in Java 5.0 version. This class is an alternative to the existing StringBuffer class. If you look into the operations of the both the classes, there is no difference. The only difference between StringBuilder and StringBuffer is that StringBuilder class is not synchronized so it gives better performance. Whenever there are no threading issues, its preferable to use StringBuilder. StringBuffer class can be replaced by StringBuilder with a simple search and replace with no compilation issue.

Accessor methods: capacity(), length(), charAt(i), indexOf(g), lastIndexOf(g)

Modifier methods: append(g), delete(i1, i2), insert(iPosn, g), getChars(i), setCharAt(iposn, c), substring(), replace(i1,i2,gvalue), reverse(), trimToSize(g), toString(g)

---

java.lang
Class StringBuilder
java.lang.Object
  ∟ **java.lang.StringBuilder**
All Implemented Interfaces:

Serializable, Appendable, CharSequence

---

public final class **StringBuilder**
extends Object
implements Serializable, CharSequence

A mutable sequence of characters. This class provides an API compatible with `StringBuffer`, but with no guarantee of synchronization. This class is designed for use as a drop-in replacement for `StringBuffer` in places where the string buffer was being used by a single thread (as is generally the case). Where possible, it is recommended that this class be used in preference to `StringBuffer` as it will be faster under most implementations.

The principal operations on a `StringBuilder` are the `append` and `insert` methods, which are overloaded so as to accept data of any type. Each effectively converts a given datum to a string and then appends or inserts the characters of that string to the string builder. The `append` method always adds these characters at the end of the builder; the `insert` method adds the characters at a specified point.

For example, if `z` refers to a string builder object whose current contents are "`start`", then the method call `z.append("le")` would cause the string builder to contain "`startle`", whereas `z.insert(4, "le")` would alter the string builder to contain "`starlet`".

In general, if sb refers to an instance of a `StringBuilder`, then `sb.append(x)` has the same effect as `sb.insert(sb.length(), x)`. Every string builder has a capacity. As long as the length of the character sequence contained in the string builder does not exceed the capacity, it is not necessary to allocate a new internal buffer. If the internal buffer overflows, it is automatically made larger.

Instances of `StringBuilder` are not safe for use by multiple threads. If such synchronization is required then it is recommended that `StringBuffer` be used.

Constructor Summary
StringBuilder() Constructs a string builder with no characters in it and an initial capacity of 16 characters.
StringBuilder(CharSequence seq) Constructs a string builder that contains the same characters as the specified CharSequence.
StringBuilder(int capacity) Constructs a string builder with no characters in it and an initial capacity specified by the capacity argument.
StringBuilder(String str) Constructs a string builder initialized to the contents of the specified string.

Method Summary	
StringBuilder	**append**(boolean b) Appends the string representation of the boolean argument to the sequence.
StringBuilder	**append**(char c) Appends the string representation of the char argument to this sequence.
StringBuilder	**append**(char[] str) Appends the string representation of the char array argument to this sequence.
StringBuilder	**append**(char[] str, int offset, int len) Appends the string representation of a subarray of the char array argument to this sequence.
StringBuilder	**append**(CharSequence s) Appends the specified character sequence to this Appendable.
StringBuilder	**append**(CharSequence s, int start, int end) Appends a subsequence of the specified CharSequence to this sequence.
StringBuilder	**append**(double d) Appends the string representation of the double argument to this sequence.

StringBuilder	**append**(float f) Appends the string representation of the float argument to this sequence.
StringBuilder	**append**(int i) Appends the string representation of the int argument to this sequence.
StringBuilder	**append**(long lng) Appends the string representation of the long argument to this sequence.
StringBuilder	**append**(Object obj) Appends the string representation of the Object argument.
StringBuilder	**append**(String str) Appends the specified string to this character sequence.
StringBuilder	**append**(StringBuffer sb) Appends the specified StringBuffer to this sequence.
StringBuilder	**appendCodePoint**(int codePoint) Appends the string representation of the codePoint argument to this sequence.
int	**capacity**() Returns the current capacity.
char	**charAt**(int index) Returns the char value in this sequence at the specified index.
int	**codePointAt**(int index) Returns the character (Unicode code point) at the specified index.
int	**codePointBefore**(int index) Returns the character (Unicode code point) before the specified index.
int	**codePointCount**(int beginIndex, int endIndex) Returns the number of Unicode code points in the specified text range of this sequence.
StringBuilder	**delete**(int start, int end) Removes the characters in a substring of this sequence.
StringBuilder	**deleteCharAt**(int index) Removes the char at the specified position in this sequence.
void	**ensureCapacity**(int minimumCapacity) Ensures that the capacity is at least equal to the specified minimum.

void	**getChars**(int srcBegin, int srcEnd, char[] dst, int dstBegin) Characters are copied from this sequence into the destination character array dst.
int	**indexOf**(String str) Returns the index within this string of the first occurrence of the specified substring.
int	**indexOf**(String str, int fromIndex) Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
StringBuilder	**insert**(int offset, boolean b) Inserts the string representation of the boolean argument into this sequence.
StringBuilder	**insert**(int offset, char c) Inserts the string representation of the char argument into this sequence.
StringBuilder	**insert**(int offset, char[] str) Inserts the string representation of the char array argument into this sequence.
StringBuilder	**insert**(int index, char[] str, int offset, int len) Inserts the string representation of a subarray of the str array argument into this sequence.
StringBuilder	**insert**(int dstOffset, CharSequence s) Inserts the specified CharSequence into this sequence.
StringBuilder	**insert**(int dstOffset, CharSequence s, int start, int end) Inserts a subsequence of the specified CharSequence into this sequence.
StringBuilder	**insert**(int offset, double d) Inserts the string representation of the double argument into this sequence.
StringBuilder	**insert**(int offset, float f) Inserts the string representation of the float argument into this sequence.
StringBuilder	**insert**(int offset, int i) Inserts the string representation of the second int argument into this sequence.
StringBuilder	**insert**(int offset, long l) Inserts the string representation of the long argument into this sequence.
StringBuilder	**insert**(int offset, Object obj) Inserts the string representation of the Object argument into this character sequence.
StringBuilder	**insert**(int offset, String str)

	Inserts the string into this character sequence.
int	**lastIndexOf**(String str) Returns the index within this string of the rightmost occurrence of the specified substring.
int	**lastIndexOf**(String str, int fromIndex) Returns the index within this string of the last occurrence of the specified substring.
int	**length**() Returns the length (character count).
int	**offsetByCodePoints**(int index, int codePointOffset) Returns the index within this sequence that is offset from the given index by codePointOffset code points.
StringBuilder	**replace**(int start, int end, String str) Replaces the characters in a substring of this sequence with characters in the specified String.
StringBuilder	**reverse**() Causes this character sequence to be replaced by the reverse of the sequence.
void	**setCharAt**(int index, char ch) The character at the specified index is set to ch.
void	**setLength**(int newLength) Sets the length of the character sequence.
CharSequence	**subSequence**(int start, int end) Returns a new character sequence that is a subsequence of this sequence.
String	**substring**(int start) Returns a new String that contains a subsequence of characters currently contained in this character sequence.
String	**substring**(int start, int end) Returns a new String that contains a subsequence of characters currently contained in this sequence.
String	**toString**() Returns a string representing the data in this sequence.
void	**trimToSize**() Attempts to reduce storage used for the character sequence.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.lang.CharSequence
charAt, length, subSequence

Constructor Detail

StringBuilder

public **StringBuilder**()

Constructs a string builder with no characters in it and an initial capacity of 16 characters.

StringBuilder

public **StringBuilder**(int capacity)

Constructs a string builder with no characters in it and an initial capacity specified by the capacity argument.

Parameters:

capacity - the initial capacity.

Throws: NegativeArraySizeException - if the capacity argument is less than 0.

StringBuilder

public **StringBuilder**(String str)

Constructs a string builder initialized to the contents of the specified string. The initial capacity of the string builder is 16 plus the length of the string argument.

Parameters:

str - the initial contents of the buffer.

Throws: NullPointerException - if str is null

StringBuilder

public **StringBuilder**(CharSequence seq)

Constructs a string builder that contains the same characters as the specified CharSequence. The initial capacity of the string builder is 16 plus the length of the CharSequence argument.

Parameters:

seq - the sequence to copy.

Throws: NullPointerException - if seq is null

Method Detail

append

public StringBuilder **append**(Object obj)

Appends the string representation of the Object argument.

The argument is converted to a string as if by the method `String.valueOf`, and the characters of that string are then appended to this sequence.

Parameters:

obj - an Object.

Returns: a reference to this object.

append

public StringBuilder **append**(String str)

Appends the specified string to this character sequence.

The characters of the `String` argument are appended, in order, increasing the length of this sequence by the length of the argument. If `str` is `null`, then the four characters `"null"` are appended.

Let *n* be the length of this character sequence just prior to execution of the `append` method. Then the character at index *k* in the new character sequence is equal to the character at index *k* in the old character sequence, if *k* is less than *n*; otherwise, it is equal to the character at index *k*-*n* in the argument `str`.

Parameters:

str - a string.

Returns: a reference to this object.

append

public StringBuilder **append**(StringBuffer sb)

Appends the specified StringBuffer to this sequence.

The characters of the `StringBuffer` argument are appended, in order, to this sequence, increasing the length of this sequence by the length of the argument. If `sb` is `null`, then the four characters `"null"` are appended to this sequence.

Let *n* be the length of this character sequence just prior to execution of the `append` method. Then the character at index *k* in the new character sequence is equal to the character at index *k* in the old character sequence, if *k* is less than *n*; otherwise, it is equal to the character at index *k*-*n* in the argument `sb`.

Parameters:

sb - the StringBuffer to append.

Returns: a reference to this object.

append

public StringBuilder **append**(CharSequence s)

Description copied from interface: Appendable

Appends the specified character sequence to this Appendable.

Depending on which class implements the character sequence `csq`, the entire sequence may not be appended. For instance, if `csq` is a `CharBuffer` then the subsequence to append is defined by the buffer's position and limit.

Specified by:

append in interface Appendable

Parameters:

s - The character sequence to append. If csq is null, then the four characters "null" are appended to this Appendable.

Returns: A reference to this Appendable

Throws: IndexOutOfBoundsException

append

public StringBuilder **append**(CharSequence s,int start, int end)

Appends a subsequence of the specified CharSequence to this sequence.

Characters of the argument `s`, starting at index `start`, are appended, in order, to the contents of this sequence up to the (exclusive) index `end`. The length of this sequence is increased by the value of `end - start`.

Let *n* be the length of this character sequence just prior to execution of the `append` method. Then the character at index *k* in this character sequence becomes equal to the character at index *k* in this sequence, if *k* is less than *n*; otherwise, it is equal to the character at index *k*+*start*-*n* in the argument `s`.

If `s` is `null`, then this method appends characters as if the s parameter was a sequence containing the four characters `"null"`.

Specified by:

append in interface Appendable

Parameters:

s - the sequence to append.

start - the starting index of the subsequence to be appended.

end - the end index of the subsequence to be appended.

Returns: a reference to this object.

Throws: IndexOutOfBoundsException - if start or end are negative, or start is greater than end or end is greater than s.length()

---

append

public StringBuilder **append**(char[] str)

Appends the string representation of the char array argument to this sequence.

The characters of the array argument are appended, in order, to the contents of this sequence. The length of this sequence increases by the length of the argument.

The overall effect is exactly as if the argument were converted to a string by the method `String.valueOf(char[])` and the characters of that string were then `appended` to this character sequence.

Parameters:

str - the characters to be appended.

Returns: a reference to this object.

append

public StringBuilder **append**(char[] str, int offset, int len)

Appends the string representation of a subarray of the char array argument to this sequence.

Characters of the `char` array `str`, starting at index `offset`, are appended, in order, to the contents of this sequence. The length of this sequence increases by the value of `len`.

The overall effect is exactly as if the arguments were converted to a string by the method `String.valueOf(char[],int,int)` and the characters of that string were then `appended` to this character sequence.

Parameters:

str - the characters to be appended.

offset - the index of the first char to append.

len - the number of chars to append.

Returns: a reference to this object.

append

public StringBuilder **append**(boolean b)
Appends the string representation of the boolean argument to the sequence.

The argument is converted to a string as if by the method `String.valueOf`, and the characters of that string are then appended to this sequence.

Parameters:

b - a boolean.

Returns: a reference to this object.

append

public StringBuilder **append**(char c)
Appends the string representation of the char argument to this sequence.

The argument is appended to the contents of this sequence. The length of this sequence increases by `1`.

The overall effect is exactly as if the argument were converted to a string by the method `String.valueOf(char)` and the character in that string were then `appended` to this character sequence.

Specified by:

append in interface Appendable

Parameters:

c - a char.

Returns: a reference to this object.

append

public StringBuilder **append**(int i)
Appends the string representation of the int argument to this sequence.

The argument is converted to a string as if by the method `String.valueOf`, and the characters of that string are then appended to this sequence.

Parameters:

i - an int.

Returns: a reference to this object.

append

public StringBuilder **append**(long lng)

Appends the string representation of the long argument to this sequence.

The argument is converted to a string as if by the method `String.valueOf`, and the characters of that string are then appended to this sequence.

Parameters:

lng - a long.

Returns: a reference to this object.

append

public StringBuilder **append**(float f)

Appends the string representation of the float argument to this sequence.

The argument is converted to a string as if by the method `String.valueOf`, and the characters of that string are then appended to this string sequence.

Parameters:

f - a float.

Returns: a reference to this object.

append

public StringBuilder **append**(double d)

Appends the string representation of the double argument to this sequence.

The argument is converted to a string as if by the method `String.valueOf`, and the characters of that string are then appended to this sequence.

Parameters:

d - a double.

Returns: a reference to this object.

appendCodePoint

public StringBuilder **appendCodePoint**(int codePoint)

Appends the string representation of the codePoint argument to this sequence.

The argument is appended to the contents of this sequence. The length of this sequence increases by `Character.charCount(codePoint)`.

The overall effect is exactly as if the argument were converted to a `char` array by the method `Character.toChars(int)` and the character in that array were then `appended` to this character sequence.

Parameters:

codePoint - a Unicode code point

Returns: a reference to this object.

delete

public StringBuilder **delete**(int start,int end)

Removes the characters in a substring of this sequence. The substring begins at the specified start and extends to the character at index end - 1 or to the end of the sequence if no such character exists. If start is equal to end, no changes are made.

Parameters:

start - The beginning index, inclusive.

end - The ending index, exclusive.

Returns: This object.

Throws: StringIndexOutOfBoundsException - if start is negative, greater than length(), or greater than end.

deleteCharAt

public StringBuilder **deleteCharAt**(int index)

Removes the char at the specified position in this sequence. This sequence is shortened by one char.

Note: If the character at the given index is a supplementary character, this method does not remove the entire character. If correct handling of supplementary characters is required, determine the number of `char`s to remove by calling `Character.charCount(thisSequence.codePointAt(index))`, where `thisSequence` is this sequence.

Parameters:

index - Index of char to remove

Returns: This object.

Throws: StringIndexOutOfBoundsException - if the index is negative or greater than or equal to length().

replace

public StringBuilder **replace**(int start, int end,String str)

Replaces the characters in a substring of this sequence with characters in the specified String. The substring begins at the specified start and extends to the character at index end - 1 or to the end of the sequence if no such character exists. First the characters in the substring are removed and then the specified String is inserted at start. (This sequence will be lengthened to accommodate the specified String if necessary.)

Parameters:

start - The beginning index, inclusive.

end - The ending index, exclusive.

str - String that will replace previous contents.

Returns:

This object.

Throws:

StringIndexOutOfBoundsException - if start is negative, greater than length(), or greater than end.

insert

public StringBuilder **insert**(int index,char[] str,int offset, int len)

Inserts the string representation of a subarray of the str array argument into this sequence. The subarray begins at the specified offset and extends len chars. The characters of the subarray are inserted into this sequence at the position indicated by index. The length of this sequence increases by len chars.

Parameters:

index - position at which to insert subarray.

str - A char array.

offset - the index of the first char in subarray to be inserted.

len - the number of chars in the subarray to be inserted.

Returns: This object

Throws:

StringIndexOutOfBoundsException - if index is negative or greater than length(), or offset or len are negative, or (offset+len) is greater than str.length.

insert

public StringBuilder **insert**(int offset,Object obj)

Inserts the string representation of the Object argument into this character sequence.

The second argument is converted to a string as if by the method `String.valueOf`, and the characters of that string are then inserted into this sequence at the indicated offset.

The offset argument must be greater than or equal to `0`, and less than or equal to the length of this sequence.

Parameters:

offset - the offset.

obj - an Object.

Returns: a reference to this object.

Throws: StringIndexOutOfBoundsException - if the offset is invalid.

insert

public StringBuilder **insert**(int offset, String str)
Inserts the string into this character sequence.

The characters of the `String` argument are inserted, in order, into this sequence at the indicated offset, moving up any characters originally above that position and increasing the length of this sequence by the length of the argument. If `str` is `null`, then the four characters `"null"` are inserted into this sequence.

The character at index *k* in the new character sequence is equal to:

- the character at index *k* in the old character sequence, if *k* is less than offset
- the character at index *k*-offset in the argument str, if *k* is not less than offset but is less than offset+str.length()
- the character at index *k*-str.length() in the old character sequence, if *k* is not less than offset+str.length()

The offset argument must be greater than or equal to `0`, and less than or equal to the length of this sequence.

Parameters:

offset - the offset.

str - a string.

Returns: a reference to this object.

Throws: StringIndexOutOfBoundsException - if the offset is invalid.

insert

public StringBuilder **insert**(int offset,char[] str)
Inserts the string representation of the char array argument into this sequence.

The characters of the array argument are inserted into the contents of this sequence at the position indicated by `offset`. The length of this sequence increases by the length of the argument.

The overall effect is exactly as if the argument were converted to a string by the method `String.valueOf(char[])` and the characters of that string were then `inserted` into this character sequence at the position indicated by `offset`.

Parameters:

offset - the offset.

str - a character array.

Returns: a reference to this object.

Throws: StringIndexOutOfBoundsException - if the offset is invalid.

insert

public StringBuilder **insert**(int dstOffset,CharSequence s)
Inserts the specified CharSequence into this sequence.

The characters of the `CharSequence` argument are inserted, in order, into this sequence at the indicated offset, moving up any characters originally above that position and increasing the length of this sequence by the length of the argument s.

The result of this method is exactly the same as if it were an invocation of this object's insert(dstOffset, s, 0, s.length()) method.

If `s` is `null`, then the four characters `"null"` are inserted into this sequence.

Parameters:

dstOffset - the offset.

s - the sequence to be inserted

Returns: a reference to this object.

Throws: IndexOutOfBoundsException - if the offset is invalid.

insert

public StringBuilder **insert**(int dstOffset,CharSequence s, int start, int end)
Inserts a subsequence of the specified CharSequence into this sequence.

The subsequence of the argument `s` specified by `start` and `end` are inserted, in order, into this sequence at the specified destination offset, moving up any characters originally above that position. The length of this sequence is increased by `end - start`.

The character at index *k* in this sequence becomes equal to:

- the character at index *k* in this sequence, if *k* is less than dstOffset
- the character at index *k*+start-dstOffset in the argument s, if *k* is greater than or equal to dstOffset but is less than dstOffset+end-start
- the character at index *k*-(end-start) in this sequence, if *k* is greater than or equal to dstOffset+end-start

The dstOffset argument must be greater than or equal to `0`, and less than or equal to the length of this sequence.

The start argument must be nonnegative, and not greater than `end`.

The end argument must be greater than or equal to `start`, and less than or equal to the length of s.

If `s` is `null`, then this method inserts characters as if the s parameter was a sequence containing the four characters `"null"`.

Parameters:

dstOffset - the offset in this sequence.

s - the sequence to be inserted.

start - the starting index of the subsequence to be inserted.

end - the end index of the subsequence to be inserted.

Returns: a reference to this object.

Throws:

IndexOutOfBoundsException - if dstOffset is negative or greater than this.length(), or start or end are negative, or start is greater than end or end is greater than s.length()

insert

public StringBuilder **insert**(int offset, boolean b)

Inserts the string representation of the boolean argument into this sequence.

The second argument is converted to a string as if by the method `String.valueOf`, and the characters of that string are then inserted into this sequence at the indicated offset.

The offset argument must be greater than or equal to `0`, and less than or equal to the length of this sequence.

Parameters:

offset - the offset.

b - a boolean.

Returns: a reference to this object.

Throws: StringIndexOutOfBoundsException - if the offset is invalid.

insert

public StringBuilder **insert**(int offset,char c)

Inserts the string representation of the char argument into this sequence.

The second argument is inserted into the contents of this sequence at the position indicated by `offset`. The length of this sequence increases by one.

The overall effect is exactly as if the argument were converted to a string by the method `String.valueOf(char)` and the character in that string were then `inserted` into this character sequence at the position indicated by `offset`.

The offset argument must be greater than or equal to `0`, and less than or equal to the length of this sequence.

Parameters:

offset - the offset.

c - a char.

Returns: a reference to this object.

Throws: IndexOutOfBoundsException - if the offset is invalid.

insert

public StringBuilder **insert**(int offset,int i)
Inserts the string representation of the second int argument into this sequence.

The second argument is converted to a string as if by the method `String.valueOf`, and the characters of that string are then inserted into this sequence at the indicated offset.

The offset argument must be greater than or equal to `0`, and less than or equal to the length of this sequence.

Parameters:

offset - the offset.

i - an int.

Returns: a reference to this object.

Throws: StringIndexOutOfBoundsException - if the offset is invalid.

insert

public StringBuilder **insert**(int offset,long l)
Inserts the string representation of the long argument into this sequence.

The second argument is converted to a string as if by the method `String.valueOf`, and the characters of that string are then inserted into this sequence at the position indicated by `offset`.

The offset argument must be greater than or equal to `0`, and less than or equal to the length of this sequence.

Parameters:

offset - the offset.

l - a long.

Returns: a reference to this object.

Throws: StringIndexOutOfBoundsException - if the offset is invalid.

insert

public StringBuilder **insert**(int offset,float f)

Inserts the string representation of the float argument into this sequence.

The second argument is converted to a string as if by the method `String.valueOf`, and the characters of that string are then inserted into this sequence at the indicated offset.

The offset argument must be greater than or equal to `0`, and less than or equal to the length of this sequence.

Parameters:

offset - the offset.

f - a float.

Returns: a reference to this object.

Throws: StringIndexOutOfBoundsException - if the offset is invalid.

insert

public StringBuilder **insert**(int offset,double d)

Inserts the string representation of the double argument into this sequence.

The second argument is converted to a string as if by the method `String.valueOf`, and the characters of that string are then inserted into this sequence at the indicated offset.

The offset argument must be greater than or equal to `0`, and less than or equal to the length of this sequence.

Parameters:

offset - the offset.

d - a double.

Returns: a reference to this object.

Throws: StringIndexOutOfBoundsException - if the offset is invalid.

indexOf

public int **indexOf**(String str)

Returns the index within this string of the first occurrence of the specified substring. The integer returned is the smallest value *k* such that:

this.toString().startsWith(str, *k*)

is true.

Parameters:

str - any string.

Returns:

if the string argument occurs as a substring within this object, then the index of the first character of the first such substring is returned; if it does not occur as a substring, -1 is returned.

Throws: NullPointerException - if str is null.

indexOf

public int **indexOf**(String str, int fromIndex)

Returns the index within this string of the first occurrence of the specified substring, starting at the specified index. The integer returned is the smallest value k for which:

```
k >= Math.min(fromIndex, str.length()) &&
        this.toString().startsWith(str, k)
```

If no such value of *k* exists, then -1 is returned.

Parameters:

str - the substring for which to search.

fromIndex - the index from which to start the search.

Returns:

the index within this string of the first occurrence of the specified substring, starting at the specified index.

Throws: NullPointerException - if str is null.

lastIndexOf

public int **lastIndexOf**(String str)

Returns the index within this string of the rightmost occurrence of the specified substring. The rightmost empty string "" is considered to occur at the index value this.length(). The returned index is the largest value *k* such that

```
this.toString().startsWith(str, k)
```

is true.

Parameters:

str - the substring to search for.

Returns:

if the string argument occurs one or more times as a substring within this object, then the index of the first character of the last such substring is returned. If it does not occur as a substring, -1 is returned.

Throws: NullPointerException - if str is null.

lastIndexOf

public int **lastIndexOf**(String str, int fromIndex)

Returns the index within this string of the last occurrence of the specified substring. The integer returned is the largest value *k* such that:

```
k <= Math.min(fromIndex, str.length()) &&
        this.toString().startsWith(str, k)
```

If no such value of *k* exists, then -1 is returned.

Parameters:

str - the substring to search for.

fromIndex - the index to start the search from.

Returns:

the index within this sequence of the last occurrence of the specified substring.

Throws: NullPointerException - if str is null.

reverse

public StringBuilder **reverse**()

Causes this character sequence to be replaced by the reverse of the sequence. If there are any surrogate pairs included in the sequence, these are treated as single characters for the reverse operation. Thus, the order of the high-low surrogates is never reversed. Let *n* be the character length of this character sequence (not the length in char values) just prior to execution of the reverse method. Then the character at index *k* in the new character sequence is equal to the character at index *n-k-1* in the old character sequence.

Note that the reverse operation may result in producing surrogate pairs that were unpaired low-surrogates and high-surrogates before the operation. For example, reversing "\uDC00\uD800" produces "\uD800\uDC00" which is a valid surrogate pair.

Returns: a reference to this object.

---

toString

public String **toString**()

Returns a string representing the data in this sequence. A new String object is allocated and initialized to contain the character sequence currently represented by this object. This String is then returned. Subsequent changes to this sequence do not affect the contents of the String.

Specified by:

toString in interface CharSequence

Returns: a string representation of this sequence of characters.

length

public int **length**()

Returns the length (character count).

Specified by:

length in interface CharSequence

Returns: the length of the sequence of characters currently represented by this object

capacity

public int **capacity**()

Returns the current capacity. The capacity is the amount of storage available for newly inserted characters, beyond which an allocation will occur.

Returns: the current capacity

ensureCapacity

public void **ensureCapacity**(int minimumCapacity)

Ensures that the capacity is at least equal to the specified minimum. If the current capacity is less than the argument, then a new internal array is allocated with greater capacity. The new capacity is the larger of:

- The minimumCapacity argument.
- Twice the old capacity, plus 2.

If the minimumCapacity argument is nonpositive, this method takes no action and simply returns.

Parameters:

minimumCapacity - the minimum desired capacity.

trimToSize

public void **trimToSize**()

Attempts to reduce storage used for the character sequence. If the buffer is larger than necessary to hold its current sequence of characters, then it may be resized to become more space efficient. Calling this method may, but is not required to, affect the value returned by a subsequent call to the capacity() method.

setLength

public void **setLength**(int newLength)

Sets the length of the character sequence. The sequence is changed to a new character sequence whose length is specified by the argument. For every nonnegative index *k* less than newLength, the character at index *k* in the new character sequence is the same as the character at index *k* in the old sequence if *k* is less than the length of the old character sequence; otherwise, it is the null character

'\u0000'. In other words, if the newLength argument is less than the current length, the length is changed to the specified length.

If the `newLength` argument is greater than or equal to the current length, sufficient null characters (`'\u0000'`) are appended so that length becomes the `newLength` argument.

The `newLength` argument must be greater than or equal to `0`.

Parameters:

newLength - the new length

Throws: IndexOutOfBoundsException - if the newLength argument is negative.

charAt

public char **charAt**(int index)

Returns the char value in this sequence at the specified index. The first char value is at index 0, the next at index 1, and so on, as in array indexing.

The index argument must be greater than or equal to `0`, and less than the length of this sequence.

If the `char` value specified by the index is a surrogate, the surrogate value is returned.

Specified by:

charAt in interface CharSequence

Parameters:

index - the index of the desired char value.

Returns: the char value at the specified index.

Throws: IndexOutOfBoundsException - if index is negative or greater than or equal to length().

codePointAt

public int **codePointAt**(int index)

Returns the character (Unicode code point) at the specified index. The index refers to char values (Unicode code units) and ranges from 0 to length() - 1.

If the `char` value specified at the given index is in the high-surrogate range, the following index is less than the length of this sequence, and the `char` value at the following index is in the low-surrogate range, then the supplementary code point corresponding to this surrogate pair is returned. Otherwise, the `char` value at the given index is returned.

Parameters:

index - the index to the char values

Returns: the code point value of the character at the index

Throws: IndexOutOfBoundsException - if the index argument is negative or not less than the length of this sequence.

codePointBefore

public int **codePointBefore**(int index)

Returns the character (Unicode code point) before the specified index. The index refers to char values (Unicode code units) and ranges from 1 to length().

If the `char` value at `(index - 1)` is in the low-surrogate range, `(index - 2)` is not negative, and the `char` value at `(index - 2)` is in the high-surrogate range, then the supplementary code point value of the surrogate pair is returned. If the `char` value at `index - 1` is an unpaired low-surrogate or a high-surrogate, the surrogate value is returned.

Parameters:

index - the index following the code point that should be returned

Returns: the Unicode code point value before the given index.

Throws: IndexOutOfBoundsException - if the index argument is less than 1 or greater than the length of this sequence.

codePointCount

public int **codePointCount**(int beginIndex,int endIndex)

Returns the number of Unicode code points in the specified text range of this sequence. The text range begins at the specified beginIndex and extends to the char at index endIndex - 1. Thus the length (in chars) of the text range is endIndex-beginIndex. Unpaired surrogates within this sequence count as one code point each.

Parameters:

beginIndex - the index to the first char of the text range.

endIndex - the index after the last char of the text range.

Returns: the number of Unicode code points in the specified text range

Throws: IndexOutOfBoundsException - if the beginIndex is negative, or endIndex is larger than the length of this sequence, or beginIndex is larger than endIndex.

offsetByCodePoints

public int **offsetByCodePoints**(int index, int codePointOffset)

Returns the index within this sequence that is offset from the given index by codePointOffset code points. Unpaired surrogates within the text range given by index and codePointOffset count as one code point each.

Parameters:

index - the index to be offset

codePointOffset - the offset in code points

Returns: the index within this sequence

Throws:

IndexOutOfBoundsException - if index is negative or larger then the length of this sequence, or if codePointOffset is positive and the subsequence starting with index has fewer than codePointOffset code points, or if codePointOffset is negative and the subsequence before index has fewer than the absolute value of codePointOffset code points.

getChars

public void **getChars**(int srcBegin, int srcEnd,char[] dst,int dstBegin)

Characters are copied from this sequence into the destination character array dst. The first character to be copied is at index srcBegin; the last character to be copied is at index srcEnd-1. The total number of characters to be copied is srcEnd-srcBegin. The characters are copied into the subarray of dst starting at index dstBegin and ending at index:

dstbegin + (srcEnd-srcBegin) - 1

Parameters:

srcBegin - start copying at this offset.

srcEnd - stop copying at this offset.

dst - the array to copy the data into.

dstBegin - offset into dst.

Throws: NullPointerException - if dst is null.

IndexOutOfBoundsException - if any of the following is true:

- srcBegin is negative
- dstBegin is negative
- the srcBegin argument is greater than the srcEnd argument.
- srcEnd is greater than this.length().
- dstBegin+srcEnd-srcBegin is greater than dst.length

setCharAt

public void **setCharAt**(int index, char ch)

The character at the specified index is set to ch. This sequence is altered to represent a new character sequence that is identical to the old character sequence, except that it contains the character ch at position index.

The index argument must be greater than or equal to `0`, and less than the length of this sequence.

Parameters:

index - the index of the character to modify.

ch - the new character.

Throws: IndexOutOfBoundsException - if index is negative or greater than or equal to length().

substring

public String **substring**(int start)

Returns a new String that contains a subsequence of characters currently contained in this character sequence. The substring begins at the specified index and extends to the end of this sequence.

Parameters:

start - The beginning index, inclusive.

Returns: The new string.

Throws: StringIndexOutOfBoundsException - if start is less than zero, or greater than the length of this object.

subSequence

public CharSequence **subSequence**(int start, int end)

Returns a new character sequence that is a subsequence of this sequence.

An invocation of this method of the form

sb.subSequence(begin, end)
behaves in exactly the same way as the invocation

sb.substring(begin, end)
This method is provided so that this class can implement the CharSequence interface.

Specified by:

subSequence in interface CharSequence

Parameters:

start - the start index, inclusive.

end - the end index, exclusive.

Returns:

the specified subsequence.

Throws: IndexOutOfBoundsException - if start or end are negative, if end is greater than length(), or if start is greater than end

substring

public String **substring**(int start, int end)

Returns a new String that contains a subsequence of characters currently contained in this sequence. The substring begins at the specified start and extends to the character at index end - 1.

Parameters:

start - The beginning index, inclusive.

end - The ending index, exclusive.

Returns: The new string.

Throws: StringIndexOutOfBoundsException - if start or end are negative or greater than length(), or start is greater than end.

Analyzing a string token-by-token

Tokenization in Java consists of two separate issues: the case where tokenization is on a character-by-character basis, and the case where tokenization is done on the basis of a separator character. The former case is well-supported in the Java platform, by way of the StringTokenizer class. The latter must be approached algorithmically.

Analyzing a string character-by-character

You will use:

• a String with your input in it

• a char to hold individual chars

• a for-loop

• the String.charAt() method

• the String.length() method

• the String.indexOf() method

The method String.charAt() returns the character at an indexed position in the input string. For example, the following code fragment analyzes an input word character-by-character and prints out a message if the input word contains a coronal consonant:

```
// the next two lines show construction of a String with a constant

String input = new String ("mita");
String coronals = new String("sztdSZ");
int index;
char tokenizedInput;
// the String.length() method returns the length of a String. you
// subtract 1 from the length because String indices are zero-based.
for (index = 0; index < input.length() - 1; index++) {
tokenizedInput = input.charAt(index);
// String.indexOf() returns -1 if the string doesn't contain the character
// in question. if it doesn't return -1, then you know that it
// does contain the character in question.
if (coronals.indexOf(tokenizedInput) != -1){
System.out.print("The word <");
```

```
System.out.print(input);
System.out.print("contains the coronal consonant <);
System.out.print(tokenizedInput);
System.out.println(">.");
}
}
```

This produces the output The word contains the coronal consonant .

Analyzing a string word-by-word

You will use:

• the StringTokenizer class

• the StringTokenizer.hasMoreTokens() method

• the StringTokenizer.nextToken() method

• a while-loop

```
// make a new String object
String input = new String("im ani le?acmi ma ani");
// make a new tokenizer object. note that you pass it the
// string that you want parsed
StringTokenizer tokenizer = new StringTokenizer(input);
// StringTokenizer.hasMoreTokens() returns true as long as
// there's more data in it that hasn't yet been given to you
while (tokenizer.hasMoreTokens()) {
// StringTokenizer.nextToken() returns the
// next token that the StringTokenizer is holding.
// (of course, the first time you call it, that
// will be the first token in the input. :-) )
String currentToken = tokenizer.nextToken();
// ...and now you can do whatever you like with
// that token!
checkForCoronalConsonants(currentToken);
```

Exploring java.util:

Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

Interface Type	Implementation Classes
Set <T>	HashSet<T> LinkedHashSet<T>
List <T>	Stack<T> LinkedList<T> ArrayList<T> Vector<T>
Queue <T>	LinkedList<T>
Map<T>	HashMap<K,V> Hashtable<K,V>

Interface Summary	
Collection<E>	The root interface in the *collection hierarchy*.
Comparator<T>	A comparison function, which imposes a *total ordering* on some collection of objects.
Enumeration<E>	An object that implements the Enumeration interface generates a series of elements, one at a time.
EventListener	A tagging interface that all event listener interfaces must extend.
Formattable	The Formattable interface must be implemented by any class that needs to perform custom formatting using the 's' conversion specifier of Formatter.
Iterator<E>	An iterator over a collection.
List<E>	An ordered collection (also known as a *sequence*).
ListIterator<E>	An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list.
Map<K,V>	An object that maps keys to values.
Map.Entry<K,V>	A map entry (key-value pair).
Observer	A class can implement the Observer interface when it wants to be informed of changes in observable objects.
Queue<E>	A collection designed for holding elements prior to processing.
RandomAccess	Marker interface used by List implementations to indicate that they support fast (generally constant time) random access.
Set<E>	A collection that contains no duplicate elements.

SortedMap<K,V>	A map that further guarantees that it will be in ascending key order, sorted according to the *natural ordering* of its keys (see the Comparable interface), or by a comparator provided at sorted map creation time.
SortedSet<E>	A set that further guarantees that its iterator will traverse the set in ascending element order, sorted according to the *natural ordering* of its elements (see Comparable), or by a Comparator provided at sorted set creation time.

Class Summary	
AbstractCollection<E>	This class provides a skeletal implementation of the Collection interface, to minimize the effort required to implement this interface.
AbstractList<E>	This class provides a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a "random access" data store (such as an array).
AbstractMap<K,V>	This class provides a skeletal implementation of the Map interface, to minimize the effort required to implement this interface.
AbstractQueue<E>	This class provides skeletal implementations of some Queue operations.
AbstractSequentialList<E>	This class provides a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a "sequential access" data store (such as a linked list).
AbstractSet<E>	This class provides a skeletal implementation of the Set interface to minimize the effort required to implement this interface.
ArrayList<E>	Resizable-array implementation of the List interface.
Arrays	This class contains various methods for manipulating arrays (such as sorting and searching).
BitSet	This class implements a vector of bits that grows as needed.
Calendar	The Calendar class is an abstract class that provides methods for converting between a specific instant in time and a set of calendar fields such as YEAR, MONTH, DAY_OF_MONTH, HOUR, and so on, and for manipulating the calendar fields, such as getting the date of the next week.
Collections	This class consists exclusively of static methods that operate on or return collections.

Currency	Represents a currency.
Date	The class Date represents a specific instant in time, with millisecond precision.
Dictionary<K,V>	The Dictionary class is the abstract parent of any class, such as Hashtable, which maps keys to values.
EnumMap<K extends Enum<K>,V>	A specialized Map implementation for use with enum type keys.
EnumSet<E extends Enum<E>>	A specialized Set implementation for use with enum types.
EventListenerProxy	An abstract wrapper class for an EventListener class which associates a set of additional parameters with the listener.
EventObject	The root class from which all event state objects shall be derived.
FormattableFlags	FomattableFlags are passed to the Formattable.formatTo() method and modify the output format for Formattables.
Formatter	An interpreter for printf-style format strings.
GregorianCalendar	GregorianCalendar is a concrete subclass of Calendar and provides the standard calendar system used by most of the world.
HashMap<K,V>	Hash table based implementation of the Map interface.
HashSet<E>	This class implements the Set interface, backed by a hash table (actually a HashMap instance).
Hashtable<K,V>	This class implements a hashtable, which maps keys to values.
IdentityHashMap<K,V>	This class implements the Map interface with a hash table, using reference-equality in place of object-equality when comparing keys (and values).
LinkedHashMap<K,V>	Hash table and linked list implementation of the Map interface, with predictable iteration order.
LinkedHashSet<E>	Hash table and linked list implementation of the Set interface, with predictable iteration order.
LinkedList<E>	Linked list implementation of the List interface.

ListResourceBundle	ListResourceBundle is an abstract subclass of ResourceBundle that manages resources for a locale in a convenient and easy to use list.
Locale	A Locale object represents a specific geographical, political, or cultural region.
Observable	This class represents an observable object, or "data" in the model-view paradigm.
PriorityQueue<E>	An unbounded priority queue based on a priority heap.
Properties	The Properties class represents a persistent set of properties.
PropertyPermission	This class is for property permissions.
PropertyResourceBundle	PropertyResourceBundle is a concrete subclass of ResourceBundle that manages resources for a locale using a set of static strings from a property file.
Random	An instance of this class is used to generate a stream of pseudorandom numbers.
ResourceBundle	Resource bundles contain locale-specific objects.
Scanner	A simple text scanner which can parse primitive types and strings using regular expressions.
SimpleTimeZone	SimpleTimeZone is a concrete subclass of TimeZone that represents a time zone for use with a Gregorian calendar.
Stack<E>	The Stack class represents a last-in-first-out (LIFO) stack of objects.
StringTokenizer	The string tokenizer class allows an application to break a string into tokens.
Timer	A facility for threads to schedule tasks for future execution in a background thread.
TimerTask	A task that can be scheduled for one-time or repeated execution by a Timer.
TimeZone	TimeZone represents a time zone offset, and also figures out daylight savings.
TreeMap<K,V>	Red-Black tree based implementation of the SortedMap interface.
TreeSet<E>	This class implements the Set interface, backed by a TreeMap instance.

UUID	A class that represents an immutable universally unique identifier (UUID).
Vector<E>	The Vector class implements a growable array of objects.
WeakHashMap<K,V>	A hashtable-based Map implementation with *weak keys*.

Enum Summary	
Formatter.BigDecimalLayoutForm	

Exception Summary	
ConcurrentModificationException	This exception may be thrown by methods that have detected concurrent modification of an object when such modification is not permissible.
DuplicateFormatFlagsException	Unchecked exception thrown when duplicate flags are provided in the format specifier.
EmptyStackException	Thrown by methods in the Stack class to indicate that the stack is empty.
FormatFlagsConversionMismatchException	Unchecked exception thrown when a conversion and flag are incompatible.
FormatterClosedException	Unchecked exception thrown when the formatter has been closed.
IllegalFormatCodePointException	Unchecked exception thrown when a character with an invalid Unicode code point as defined by Character.isValidCodePoint(int) is passed to the Formatter.
IllegalFormatConversionException	Unchecked exception thrown when the argument corresponding to the format specifier is of an incompatible type.
IllegalFormatException	Unchecked exception thrown when a format string contains an illegal syntax or a format specifier that is incompatible with the given arguments.
IllegalFormatFlagsException	Unchecked exception thrown when an illegal combination flags is given.

IllegalFormatPrecisionException	Unchecked exception thrown when the precision is a negative value other than -1, the conversion does not support a precision, or the value is otherwise unsupported.
IllegalFormatWidthException	Unchecked exception thrown when the format width is a negative value other than -1 or is otherwise unsupported.
InputMismatchException	Thrown by a Scanner to indicate that the token retrieved does not match the pattern for the expected type, or that the token is out of range for the expected type.
InvalidPropertiesFormatException	Thrown to indicate that an operation could not complete because the input did not conform to the appropriate XML document type for a collection of properties, as per the Properties specification.
MissingFormatArgumentException	Unchecked exception thrown when there is a format specifier which does not have a corresponding argument or if an argument index refers to an argument that does not exist.
MissingFormatWidthException	Unchecked exception thrown when the format width is required.
MissingResourceException	Signals that a resource is missing.
NoSuchElementException	Thrown by the nextElement method of an Enumeration to indicate that there are no more elements in the enumeration.
TooManyListenersException	The TooManyListenersException Exception is used as part of the Java Event model to annotate and implement a unicast special case of a multicast Event Source.
UnknownFormatConversionException	Unchecked exception thrown when an unknown conversion is given.
UnknownFormatFlagsException	Unchecked exception thrown when an unknown flag is given.

Stream based I/O

I/O Basics

- In fact, aside from print( ) and println( ), none of the I/O methods have been used significantly.
- The reason is simple: most real applications of Java are not text-based, console programs.
- Rather, they are graphically oriented applets that rely upon Java's Abstract Window Toolkit (AWT) for interaction with the user.

Streams

- Java programs perform I/O through streams.
- A stream is an abstraction 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 to which they are linked differ. Thus, the same I/O classes and methods can be applied to any type of device.
- This means that 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.
- Streams are a clean way to deal with input/output without having every part of your code understand the difference between a keyboard and a network, for example. Java implements streams within class hierarchies defined in the java.io package.

Byte Streams and Character Streams

- Java defines two types of streams:

  Byte &
  Character.

- Byte streams provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data.
- Character streams provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized.

The Byte Stream Classes

- Byte streams are defined by using two class hierarchies.
- At the top are two abstract classes:

  InputStream &

  OutputStream.

- Each of these abstract classes has several concrete subclasses, that handle the differences between various devices, such as disk files, network connections, and even memory buffers.

BYTE STREAM CLASSES

BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An inut 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 writes to a file
FilterInputStream	Implements InputStream
FilterOutputStream	Implements OutputStream
InputStream	Abstract class that describes stream input
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 “unget,” which returns a byte to the input stream
RandomAccessFile	Supports random access file I/O
SequenceInputStream	Input stream that is a compination of two or more input streams that will be read sequentially, one after the other.

The Character Stream Classes

- Character streams are defined by using two class hierarchies.
- At the top are two abstract classes,

  Reader
  &
  Writer.
- These abstract classes handle Unicode character streams.

- The abstract classes Reader and Writer define several key methods that the other stream classes implement.
- Two of the most important methods are read( ) and write( ), which read and write characters of data, respectively.
- These methods are overridden by derived stream classes.

CHARACTER STREAM 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
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
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

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,
&
err.

- These fields are declared as public and static within System.
- This means that they can be used by any other part of your 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.
- 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.

Using Byte Streams

- Reading Console Input:
- 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 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.
- 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.

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.
- It returns –1 when the end of the stream is encountered.

Program 1
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');
}
}
```

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

String readLine( ) throws IOException Program

```
import java.io.*;
class BRReadLines
{
public static void main(String
args[]) throws IOException
{
// create a BufferedReader using System.in
BufferedReader br = new
BufferedReader(new
InputStreamReader(System.in));
String str;
System.out.println("Enter lines of
text."); System.out.println("Enter 'stop'
to quit."); 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.

- 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.
- The simplest form of write() defined by PrintStream is

void write(int byteval) Program 3

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

PrintWriter Class

- 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)

- 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. Program 4

```
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 using Byte Streams

- 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 &
FileOutputStream,

- which create an object of one of these classes, specifying the name of the file as an argument to

- the constructor.
- both classes support additional, overridden constructors, the following are the forms that we will be using:

  FileInputStream(String filename)throws FileNotFoundException FileOutputStram(String filename throws FileNotFoundException
- When you are done with a file, you should 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

To Show a Text
file Program 5

```
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();
}
}
```

To Copy a Text
file Program 6

```
import
java.io.*; class
CopyFile {

public static void main(String
args[]) throws IOException
{

int i;
FileInputStream
fin;

FileOutputStream
fout; try {

// open input
file try {

fin = new
FileInputStream(args[0]); }
catch(FileNotFoundException e) {

System.out.println("Input File Not
Found"); return;
}
```

```
// open output file
try {
fout = new FileOutputStream(args[1]); }
catch(FileNotFoundException e) {
System.out.println("Error Opening Output File"); return;
}
} catch(ArrayIndexOutOfBoundsException e) { System.out.println("Usage: CopyFile From To"); return;
}
// Copy File
try
{
do
{
i = fin.read();
if(i != -1) fout.write(i);
} while(i != -1);
} catch(IOException e) {
System.out.println("File Error");
}
fin.close();
fout.close();
}
}
```

RandomAccessFile

- RandomAccessFile encapsulates a random-access file.
- It is not derived from InputStream or OutputStream.
- Instead, it implements the interfaces DataInput and DataOutput, which define the basic I/O methods.
- It also supports positioning requests—that is, you can position the file pointer within the file.
- It has these two constructors:

```
RandomAccessFile(File fileObj, String access) throws FileNotFoundException
RandomAccessFile(String filename, String access) throws FileNotFoundException
```

File I/O using character streams

FileReader:

- The FileReader class creates a Reader that you can use to read the contents of a file.
- Its two most commonly used constructors are shown here:

```
FileReader(String filePath) FileReader(File fileObj)
```

Program 7

```
import
java.io.*;
class FileReaderDemo {
public static void main(String args[]) throws Exception
{ FileReader fr = new
FileReader("FileReaderDemo.java"); BufferedReader
br = new BufferedReader(fr);
String s;
while((s = br.readLine()) !=
null) { System.out.println(s);
}
fr.close();
}
}
```

FileWriter:

- FileWriter creates a Writer that you can use to write to a file.
- Its most commonly used constructors are shown here: FileWriter(String filePath)

  FileWriter(String filePath, boolean append) FileWriter(File fileObj)

  FileWriter(File fileObj, boolean append) Program 8

```
import java.io.*;
class FileWriterDemo {
public static void main(String args[]) throws
Exception { String source = "Now is the time for all
good men\n"
+ " to come to the aid of their country\n"
+ " and pay their due taxes.";
char buffer[] = new char[source.length()];
source.getChars(0, source.length(), buffer,
0); FileWriter f0 = new
FileWriter("file1.txt"); for (int i=0; i <
buffer.length; i += ) { f0.write(buffer[i]);
}
f0.close();
FileWriter f1 = new
FileWriter("file.txt"); f1.write(buffer);
f1.close();
FileWriter f = new FileWriter("file3.txt");
f.write(buffer,buffer.length-
buffer.length/4,buffer.length/4); f.close();
}
}
```

Wrappers

- Java uses simple types, such as int and char, for performance reasons.
- These data types are not part of the object hierarchy. They are passed by value to methods and cannot be directly passed by reference.
- Also, there is no way for two methods to refer to the same instance of an int.
- At times, you will need to create an object representation for one of these simple types.
- To store a simple type in one of these classes, you need to wrap the simple type in a class.
- To address this need, Java provides classes that correspond to each of the simple types.
- In essence, these classes encapsulate, or wrap, the simple types within a class.
- Thus, they are commonly referred to as type wrappers.