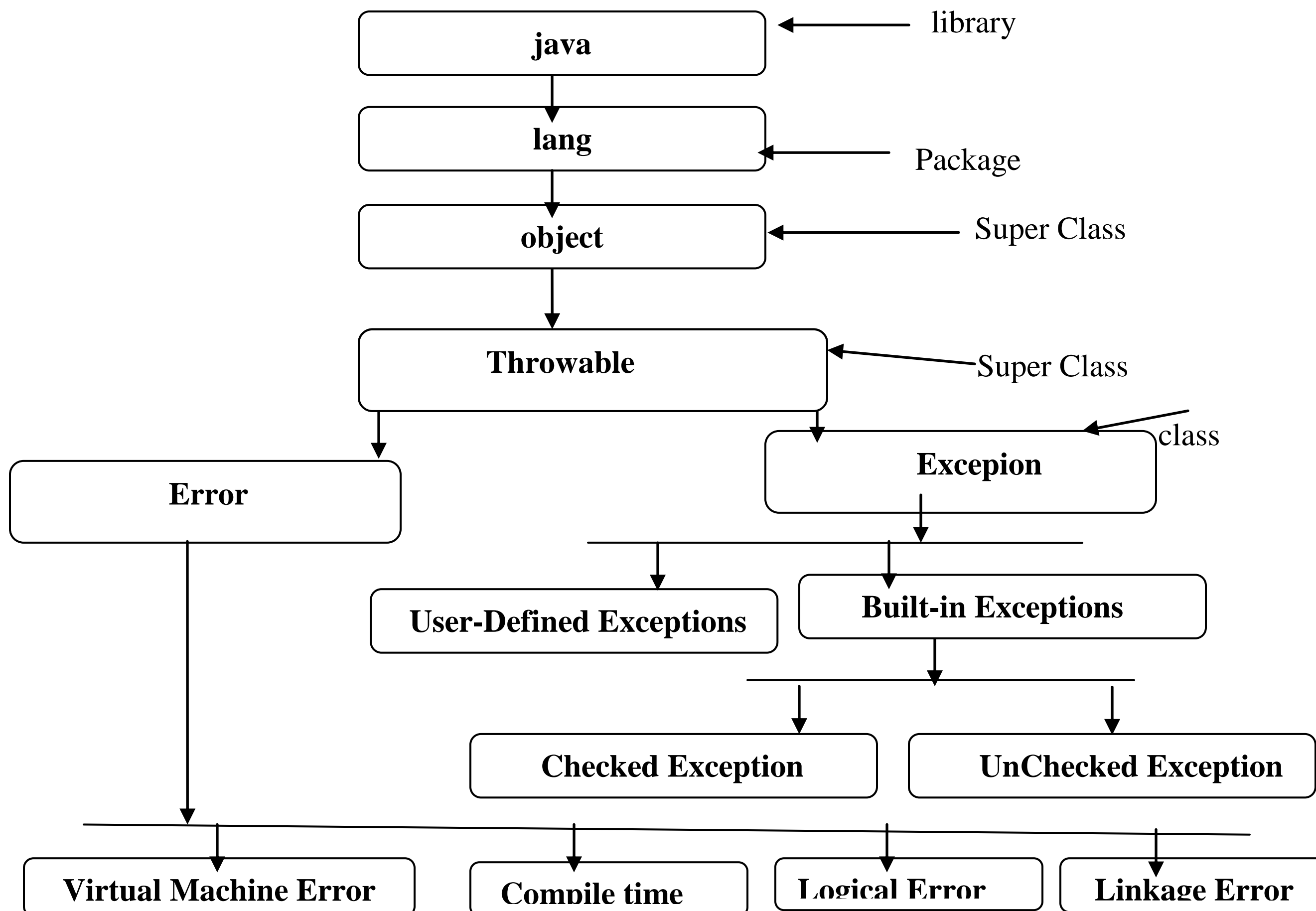
Unit-V

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.

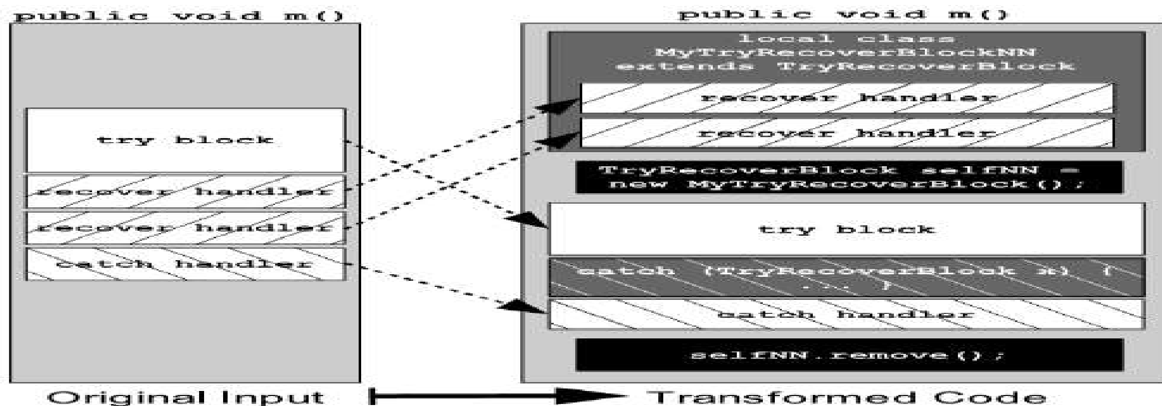


Fig. 2. Sketch of the basic transformation of a try-recover construct by the precompiler.

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:

```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac Mutable.java

D:\JQR>java  Mutable
Enter sur name : Chandra
Enter mid name : Sekhar
Enter last name : Azad
Full name = Chandra Sekhar Azad
In reverse =dazA rahkeS ardnahC

D:\JQR>_
```

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 chars 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.

Class Hierarchy

- java.lang.**Object**
  - java.util.**AbstractCollection**<E> (implements java.util.Collection<E>)
    - java.util.**AbstractList**<E> (implements java.util.List<E>)
      - java.util.**AbstractSequentialList**<E>
        - java.util.**LinkedList**<E> (implements java.lang.Cloneable, java.util.List<E>, java.util.Queue<E>, java.io.Serializable)
      - java.util.**ArrayList**<E> (implements java.lang.Cloneable, java.util.List<E>, java.util.RandomAccess, java.io.Serializable)
      - java.util.**Vector**<E> (implements java.lang.Cloneable, java.util.List<E>, java.util.RandomAccess, java.io.Serializable)
        - java.util.**Stack**<E>
    - java.util.**AbstractQueue**<E> (implements java.util.Queue<E>)
      - java.util.**PriorityQueue**<E> (implements java.io.Serializable)
    - java.util.**AbstractSet**<E> (implements java.util.Set<E>)
      - java.util.**EnumSet**<E> (implements java.lang.Cloneable, java.io.Serializable)
      - java.util.**HashSet**<E> (implements java.lang.Cloneable, java.io.Serializable, java.util.Set<E>)
        - java.util.**LinkedHashSet**<E> (implements java.lang.Cloneable, java.io.Serializable, java.util.Set<E>)
      - java.util.**TreeSet**<E> (implements java.lang.Cloneable, java.io.Serializable, java.util.SortedSet<E>)
  - java.util.**AbstractMap**<K,V> (implements java.util.Map<K,V>)
    - java.util.**EnumMap**<K,V> (implements java.lang.Cloneable, java.io.Serializable)
    - java.util.**HashMap**<K,V> (implements java.lang.Cloneable, java.util.Map<K,V>, java.io.Serializable)
      - java.util.**LinkedHashMap**<K,V> (implements java.util.Map<K,V>)
    - java.util.**IdentityHashMap**<K,V> (implements java.lang.Cloneable, java.util.Map<K,V>, java.io.Serializable)
    - java.util.**TreeMap**<K,V> (implements java.lang.Cloneable, java.io.Serializable, java.util.SortedMap<K,V>)
    - java.util.**WeakHashMap**<K,V> (implements java.util.Map<K,V>)
  - java.util.**Arrays**
  - java.util.**BitSet** (implements java.lang.Cloneable, java.io.Serializable)
  - java.util.**Calendar** (implements java.lang.Cloneable, java.lang.Comparable<T>, java.io.Serializable)
    - java.util.**GregorianCalendar**
  - java.util.**Collections**
  - java.util.**Currency** (implements java.io.Serializable)
  - java.util.**Date** (implements java.lang.Cloneable, java.lang.Comparable<T>, java.io.Serializable)
  - java.util.**Dictionary**<K,V>

  - java.util.**Hashtable**<K,V> (implements java.lang.Cloneable, java.util.Map<K,V>, java.io.Serializable)
    - java.util.**Properties**
- java.util.**EventListenerProxy** (implements java.util.EventListener)
- java.util.**EventObject** (implements java.io.Serializable)
- java.util.**FormattableFlags**
- java.util.**Formatter** (implements java.io.Closeable, java.io.Flushable)
- java.util.**Locale** (implements java.lang.Cloneable, java.io.Serializable)
- java.util.**Observable**
- java.security.**Permission** (implements java.security.Guard, java.io.Serializable)
  - java.security.**BasicPermission** (implements java.io.Serializable)
    - java.util.**PropertyPermission**
- java.util.**Random** (implements java.io.Serializable)
- java.util.**ResourceBundle**
  - java.util.**ListResourceBundle**
  - java.util.**PropertyResourceBundle**
- java.util.**Scanner** (implements java.util.Iterator<E>)
- java.util.**StringTokenizer** (implements java.util.Enumeration<E>)
- java.lang.**Throwable** (implements java.io.Serializable)
  - java.lang.**Exception**
    - java.io.**IOException**
      - java.util.**InvalidPropertiesFormatException**
    - java.lang.**RuntimeException**
      - java.util.**ConcurrentModificationException**
      - java.util.**EmptyStackException**
      - java.lang.**IllegalArgumentException**
        - java.util.**IllegalFormatException**
          - java.util.**DuplicateFormatFlagsException**
          - java.util.**FormatFlagsConversionMismatchException**
          - java.util.**IllegalFormatCodePointException**
          - java.util.**IllegalFormatConversionException**
          - java.util.**IllegalFormatFlagsException**
          - java.util.**IllegalFormatPrecisionException**
          - java.util.**IllegalFormatWidthException**
          - java.util.**MissingFormatArgumentException**
          - java.util.**MissingFormatWidthException**
          - java.util.**UnknownFormatConversionException**
          - java.util.**UnknownFormatFlagsException**
      - java.lang.**IllegalStateException**
        - java.util.**FormatterClosedException**
      - java.util.**MissingResourceException**

- java.util.**NoSuchElementException**
  - java.util.**InputMismatchException**
- java.util.**TooManyListenersException**

java.util.**Timer**
java.util.**TimerTask** (implements java.lang.Runnable)
java.util.**TimeZone** (implements java.lang.Cloneable, java.io.Serializable)

- java.util.**SimpleTimeZone**

java.util.**UUID** (implements java.lang.Comparable<T>, java.io.Serializable)

Interface Hierarchy

- java.util.**Comparator**<T>
- java.util.**Enumeration**<E>
- java.util.**EventListener**
- java.util.**Formattable**
- java.lang.**Iterable**<T>
  - java.util.**Collection**<E>
    - java.util.**List**<E>
    - java.util.**Queue**<E>
    - java.util.**Set**<E>
      - java.util.**SortedSet**<E>
- java.util.**Iterator**<E>
  java.util.**ListIterator**<E>
- java.util.**Map**<K,V>
  java.util.**SortedMap**<K,V>
- java.util.**Map.Entry**<K,V>
- java.util.**Observer**
- java.util.**RandomAccess**

Enum Hierarchy

- java.lang.**Object**
  - java.lang.**Enum**<E> (implements java.lang.Comparable<T>, java.io.Serializable)
    - java.util.**Formatter.BigDecimalLayoutForm**

Introduction

The Java 2 platform includes a *collections framework*. A *collection* is an object that represents a group of objects (such as the familiar Vector class). A collections framework is a unified architecture for representing and manipulating collections, allowing them to be manipulated independently of the details of their representation.

In order to handle group of objects we can use array of objects. If we have a class called Employ with members name and id, if we want to store details of 10 Employees, create an array of object to hold 10 Employ details.

Employ ob [] = new Employ [10];

- We cannot store different class objects into same array.
- Inserting element at the end of array is easy but at the middle is difficult.
- After retriving the elements from the array, in order to process the elements we dont have any methods

Collection Object:

- A collection object is an object which can store group of other objects.
- A collection object has a class called Collection class or Container class.
- All the collection classes are available in the package called 'java.util' (util stands for utility).
- Group of collection classes is called a Collection Framework.
- A collection object does not store the physical copies of other objects; it stores references of other objects.

The primary advantages of a collections framework are that it:

- **Reduces programming effort** by providing useful data structures and algorithms so you don't have to write them yourself.
- **Increases performance** by providing high-performance implementations of useful data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be easily tuned by switching implementations.
- **Provides interoperability between unrelated APIs** by establishing a common language to pass collections back and forth.
- **Reduces the effort required to learn APIs** by eliminating the need to learn multiple ad hoc collection APIs.
- **Reduces the effort required to design and implement APIs** by eliminating the need to produce ad hoc collections APIs.
- **Fosters software reuse** by providing a standard interface for collections and algorithms to manipulate them.

The collections framework consists of:

- **Collection Interfaces** - Represent different types of collections, such as sets, lists and maps. These interfaces form the basis of the framework.
- **General-purpose Implementations** - Primary implementations of the collection interfaces.

- **Legacy Implementations** - The collection classes from earlier releases, Vector and Hashtable, have been retrofitted to implement the collection interfaces.
- **Special-purpose Implementations** - Implementations designed for use in special situations. These implementations display nonstandard performance characteristics, usage restrictions, or behavior.
- **Concurrent Implementations** - Implementations designed for highly concurrent use.
- **Wrapper Implementations** - Add functionality, such as synchronization, to other implementations.
- **Convenience Implementations** - High-performance "mini-implementations" of the collection interfaces.
- **Abstract Implementations** - Partial implementations of the collection interfaces to facilitate custom implementations.
- **Algorithms** - Static methods that perform useful functions on collections, such as sorting a list.
- **Infrastructure** - Interfaces that provide essential support for the collection interfaces.
- **Array Utilities** - Utility functions for arrays of primitives and reference objects. Not, strictly speaking, a part of the Collections Framework, this functionality was added to the Java platform at the same time and relies on some of the same infrastructure.

Collection Interfaces

There are nine *collection interfaces*. The most basic interface is Collection. Five interfaces extend Collection: Set, List, SortedSet, Queue, and BlockingQueue. The other three collection interfaces, Map, SortedMap, and ConcurrentMap do not extend Collection, as they represent mappings rather than true collections. However, these interfaces contain *collection-view* operations, which allow them to be manipulated as collections.

All of the modification methods in the collection interfaces are labeled *optional*. Some implementations may not perform one or more of these operations, throwing a runtime exception (UnsupportedOperationException) if they are attempted. Implementations must specify in their documentation which optional operations they support. Several terms are introduced to aid in this specification:

- Collections that do not support any modification operations (such as add, remove and clear) are referred to as *unmodifiable*. Collections that are not unmodifiable are referred to *modifiable.*
- Collections that additionally guarantee that no change in the Collection object will ever be visible are referred to as *immutable*. Collections that are not immutable are referred to as *mutable*.
- Lists that guarantee that their size remains constant even though the elements may change are referred to as *fixed-size*. Lists that are not fixed-size are referred to as *variable-size*.
- Lists that support fast (generally constant time) indexed element access are known as *random access* lists. Lists that do not support fast indexed element access are known as

sequential access lists. The RandomAccess marker interface is provided to allow lists to advertise the fact that they support random access. This allows generic algorithms to alter their behavior to provide good performance when applied to either random or sequential access lists.

Some implementations may restrict what elements (or in the case of Maps, keys and values) may be stored. Possible restrictions include requiring elements to:

- Be of a particular type.
- Be non-null.
- Obey some arbitrary predicate.

Attempting to add an element that violates an implementation's restrictions results in a runtime exception, typically a ClassCastException, an IllegalArgumentException or a NullPointerException. Attempting to remove or test for the presence of an element that violates an implementation's restrictions may result in an exception, though some "restricted collections" may permit this usage.

Collection Implementations

Classes that implement the collection interfaces typically have names of the form <*Implementation-style*><*Interface*>. The general purpose implementations are summarized in the table below:

		Implementations				
		Hash Table	**Resizable Array**	**Balanced Tree**	**Linked List**	**Hash Table + Linked List**
Interfaces	**Set**	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap

The general-purpose implementations support all of the *optional operations* in the collection interfaces, and have no restrictions on the elements they may contain. They are unsynchronized, but the Collections class contains static factories called *synchronization wrappers* that may be used to add synchronization to any unsynchronized collection. All of the new implementations

have *fail-fast iterators*, which detect illegal concurrent modification, and fail quickly and cleanly (rather than behaving erratically).

- The AbstractCollection, AbstractSet, AbstractList, AbstractSequentialList and AbstractMap classes provide skeletal implementations of the core collection interfaces, to minimize the effort required to implement them. The API documentation for these classes describes precisely how each method is implemented so the implementer knows which methods should be overridden, given the performance of the "basic operations" of a specific implementation.
- Set: A Set represents a group of elements (objects) arranged just like an array. The set will grow dynamically when the elements are stored into it. A set will not allow duplicate elements.
- List: Lists are like sets but allow duplicate values to be stored.
- Queue: A Queue represents arrangement of elements in FIFO (First In First Out) order. This means that an element that is stored as a first element into the queue will be removed first from the queue.
- Map: Maps store elements in the form of key value pairs. If the key is provided its corresponding value can be obtained.

Retrieving Elements from Collections: Following are the ways to retrieve any element from a collection object:

- Using Iterator interface.
- Using ListIterator interface.
- Using Enumeration interface.

Iterator Interface: Iterator is an interface that contains methods to retrieve the elements one by one from a collection object. It retrieves elementsonly in forward direction. It has 3 methods:

Method	Description
boolean hasNext()	This method returns true if the iterator has more elements.
element next()	This method returns the next element in the iterator.
void remove()	This method removes the last element from the collection returned by the iterator.

ListIterator Interface: ListIterator is an interface that contains methods to retrieve the elements from a collection object, both in forward and reverse directions. It can retrieve the elements in forward and backward direction. It has the following important methods:

Method	Description
boolean hasNext()	This method returns true if the ListIterator has more elements when traversing the list in forward direction.
element next()	This method returns the next element.
void remove()	This method removes the list last element that was returned by the next () or previous () methods.
boolean hasPrevious()	This method returns true if the ListIterator has more elements when traversing the list in reverse direction.
element previous()	This method returns the previous element in the list.

Enumeration Interface: This interface is useful to retrieve elements one by one like Iterator. It has 2 methods.

Method	Description
boolean hasMoreElements()	This method tests Enumeration has any more elements.
element nextElement()	This returns the next element that is available in Enumeration.

HashSet Class: HashSet represents a set of elements (objects). It does not guarantee the order of elements. Also it does not allow the duplicate elements to be stored.

- We can write the HashSet class as: class HashSet<T>
- We can create the object as: HashSet<String> hs = new HashSet<String> ();

The following constructors are available in HashSet:

- HashSet();
- HashSet (int capacity); Here capacity represents how many elements can be stored into the HashSet initially. This capacity may increase automatically when more number of elements is being stored.

HashSet Class Methods:

Method	Description
boolean add(obj)	This method adds an element obj to the HashSet. It returns true if the element is added to the HashSet, else it returns false. If the same element is already available in the HashSet, then the present element is not added.
boolean remove(obj)	This method removes the element obj from the HashSet, if it is present. It returns true if the element is removed successfully otherwise false.
void clear()	This removes all the elements from the HashSet
boolean contains(obj)	This returns true if the HashSet contains the specified element obj.
boolean isEmpty()	This returns true if the HashSet contains no elements.
int size()	This returns the number of elements present in the HashSet.

Program : Write a program which shows the use of HashSet and Iterator.

```
//HashSet Demo
import java.util.*;
class HS
{  public static void main(String args[])
  {  //create a HashSet to store Strings
    HashSet <String> hs = new HashSet<String> ();
    //Store some String elements
    hs.add ("India");
    hs.add ("America");
    hs.add ("Japan");
    hs.add ("China");
    hs.add ("America");
    //view the HashSet
    System.out.println ("HashSet = " + hs);
    //add an Iterator to hs
    Iterator it = hs.iterator ();
    //display element by element using Iterator
    System.out.println ("Elements Using Iterator: ");
    while (it.hasNext() )
    {  String s = (String) it.next ();
      System.out.println(s);
    }
  }
}
```

Output:

```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac HS.java

D:\JQR>java  HS
HashSet = [America, China, Japan, India]
Elements Using Iterator:
America
China
Japan
India

D:\JQR>
```

LinkedHashSet Class: This is a subclass of HashSet class and does not contain any additional members on its own. LinkedHashSet internally uses a linked list to store the elements. It is a generic class that has the declaration:

class LinkedHashSet<T>

Stack Class: A stack represents a group of elements stored in LIFO (Last In First Out) order.

This means that the element which is stored as a last element into the stack will be the first element to be removed from the stack. Inserting the elements (Objects) into the stack is called push operation and removing the elements from stack is called pop operation. Searching for an element in stack is called peep operation. Insertion and deletion of elements take place only from one side of the stack, called top of the stack. We can write a Stack class as:

class Stack<E>

e.g.: Stack<Integer> obj = new Stack<Integer> ();

Stack Class Methods:

Method	Description
boolean empty()	this method tests whether the stack is empty or not. If the stack is empty then true is returned otherwise false.
element peek()	this method returns the top most object from the stack without removing it.
element pop()	this method pops the top-most element from the stack and returns it.
element push(element obj)	this method pushes an element obj onto the top of the stack and returns that element.
int search(Object obj)	This method returns the position of an element obj from the top of the stack. If the element (object) is not found in the stack then it returns -1.

Program : Write a program to perform different operations on a stack.

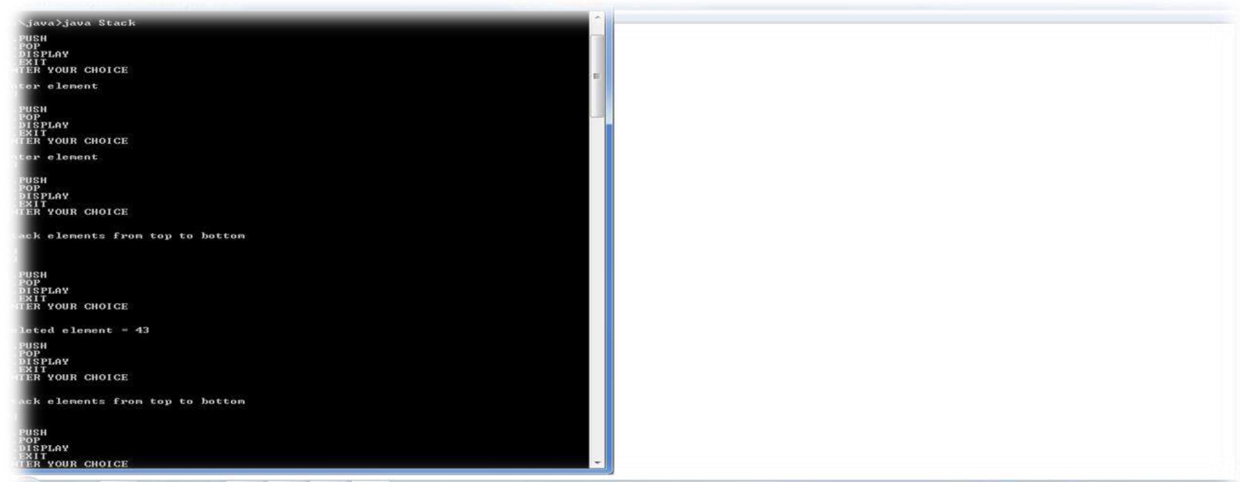
//pushing, popping, searching elements in a stack

import java.io.*;

```
import java.util.*;
class Stack1
{
int top=-1,st[]=new int[5];
void push(int el)
{
st[++top]=el;
}
int pop()
{
return(st[top--]);
}
void display()
{
System.out.println("\nStack elements from top to bottom\n");
for(int i=top;i>=0;i--)
System.out.println(st[i]);
}
boolean isFull()
{
return(top==5-1);
}
boolean isEmpty()
{
return(top==-1);
}
}
```

```
class Stack
{
public static void main(String a[])
{
Scanner sc=new Scanner(System.in);
Stack1 s=new Stack1();
int el=0,ch=1;
while(ch!=4)
{
System.out.println("\n1.PUSH\n2.POP\n3.DISPLAY\n4.EXIT");
System.out.println("ENTER YOUR CHOICE");
ch=sc.nextInt();
switch(ch)
{
case 1:if(s.isFull())
     System.out.println("\nstack is full");
     else
     {
     System.out.println("Enter element");
     el=sc.nextInt();
     s.push(el);
     }break;
case 2:if(s.isEmpty())
     System.out.println("\nstack is empty");
     else
     {
     el=s.pop();
```

```
      System.out.println("\nDeleted element = "+el);
      }break;
case 3:if(s.isEmpty())
      System.out.println("\nstack is empty");
      else
      s.display();
      break;
case 4:break;
default:System.out.println("\nEnter correct choice");
}
}
}
}
```



LinkedList Class: A linked list contains a group of elements in the form of nodes. Each node will have three fields- the data field contatins data and the link fields contain references to previous and next nodes.A linked list is written in the form of:

class LinkedList<E>

we can create an empty linked list for storing String type elements (objects) as:

LinkedList <String> ll = new LinkedList<String> ();

LinkedList Class methods:

Method	Description
boolean add (element obj)	This method adds an element to the linked list. It returns true if the element is added successfully.
void add(int position, element obj)	This method inserts an element obj into the linked list at a specified position.
void addFirst(element obj)	This method adds the element obj at the first position of the linked list.
void addLast(element obj)	This method adds the element obj at the last position of the linked list.
element removeFirst ()	This method removes the first element from the linked list and returns it.
element removeLast ()	This method removes the last element from the linked list and returns it.
element remove (int position)	This method removes an element at the specified position in the linked list.
void clear ()	This method removes all the elements from the linked list.
element get (int position)	This method returns the element at the specified position in the linked list.
element getFirst ()	This method returns the first element from the list.
element getLast ()	This method returns the last element from the list.
element set(int position, element obj)	This method replaces the element at the specified position in the list with the specified element obj.
int size ()	Returns number of elements in the linked list.
int indexOf (Object obj)	This method returns the index of the first occurrence of the specified element in the list, or -1 if the list does not contain the element.
int lastIndexOf (Object obj)	This method returns the index of the last occurrence of the specified element in the list, or -1 if the list does not contain the element.
Object[] toArray()	This method converts the linked list into an array of Object class type. All the elements of the linked list will be stored into the array in the same sequence.

Note: In case of LinkedList counting starts from 0 and we start counting from 1.

Program : Write a program that shows the use of LinkedList class.

```
import java.util.*;

//Linked List

class LinkedDemo

{  public static void main(String args[])

  {  LinkedList <String> ll = new LinkedList<String>();
```

```
  ll.add ("Asia");
  ll.add ("North America");
  ll.add ("South America");
  ll.add ("Africa");
  ll.addFirst ("Europe");
  ll.add (1,"Australia");
  ll.add (2,"Antarctica");
  System.out.println ("Elements in Linked List is : " + ll);
  System.out.println ("Size of the Linked List is : " + ll.size() );
 }
}
```

Output:

```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac LinkedDemo.java

D:\JQR>java  LinkedDemo
Elements in Linked List is : [Europe, Australia, Antarctica, Asia, North America
, South America, Africa]
Size of the Linked List is : 7

D:\JQR>
```

ArrayList Class: An ArrayList is like an array, which can grow in memory dynamically.

ArrayList is not synchronized. This means that when more than one thread acts simultaneously on the ArrayList object, the results may be incorrect in some cases.

ArrayList class can be written as: class ArrayList <E>

We can create an object to ArrayList as: ArrayList <String> arl = new ArrayList<String> ();

ArrayList Class Methods:

Method	Description
boolean add (element obj)	This method appends the specified element to the end of the ArrayList. If the element is added successfully then the method returns true.
void add(int position, element obj)	This method inserts the specified element at the specified positioin in the ArrayList.
element remove(int position)	This method removes the element at the specified position in the ArrayList and returns it.
boolean remove (Object	This method removes the first occurrence of the specified element

obj)	obj from the ArrayList, if it is present.
void clear ()	This method removes all the elements from the ArrayList.
element set(int position, element obj)	This method replaces an element at the specified position in the ArrayList with the specified element obj.
boolean contains (Object obj)	This method returns true if the ArrayList contains the specified element obj.
element get (int position)	This method returns the element available at the specified position in the ArrayList.
int size ()	Returns number of elements in the ArrayList.
int indexOf (Object obj)	This method returns the index of the first occurrence of the specified element in the list, or -1 if the list does not contain the element.
int lastIndexOf (Object obj)	This method returns the index of the last occurrence of the specified element in the list, or -1 if the list does not contain the element.
Object[] toArray ()	This method converts the ArrayLlist into an array of Object class type. All the elements of the ArrayList will be stored into the array in the same sequence.

Program : Write a program that shows the use of ArrayList class.

```
import java.util.*;
//ArrayList Demo
class ArrayListDemo
{  public static void main(String args[])
  {  ArrayList <String> al = new ArrayList<String>();
    al.add ("Asia");
    al.add ("North America");
    al.add ("South America");
    al.add ("Africa");
    al.add ("Europe");
    al.add (1,"Australia");
    al.add (2,"Antarctica");
    System.out.print ("Size of the Array List is: " + al.size ());
    System.out.print ("\nRetrieving elements in ArrayList using Iterator :");
    Iterator it = al.iterator ();
    while (it.hasNext () )
```

```
        System.out.print (it.next () + "\t");
    }
}
```

Output:

```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac ArrayListDemo.java

D:\JQR>java  ArrayListDemo
Size of the Array List is: 7
Retrieving elements in ArrayList using Iterator :Asia   Australia       Antarcti
ca      North America   South America   Africa  Europe
D:\JQR>_
```

Vector Class: Similar to ArrayList, but Vector is synchronized. It means even if several threads act on Vector object simultaneously, the results will be reliable.

Vector class can be written as: class Vector <E>

We can create an object to Vector as: Vector <String> v = new Vector<String> ();

Vector Class Methods:

Method	Description
boolean add(element obj)	This method appends the specified element to the end of the Vector. If the element is added successfully then the method returns true.
void add (int position, element obj)	This method inserts the specified element at the specified positioin in the Vector.
element remove (int position)	This method removes the element at the specified position in the Vector and returns it.
boolean remove (Object obj)	This method removes the first occurrence of the specified element obj from the Vector, if it is present.
void clear ()	This method removes all the elements from the Vector.
element set (int position, element obj)	This method replaces an element at the specified position in the Vector with the specified element obj.
boolean contains (Object obj)	This method returns true if the Vector contains the specified element obj.
element get (int position)	This method returns the element available at the specified position in the Vector.
int size ()	Returns number of elements in the Vector.

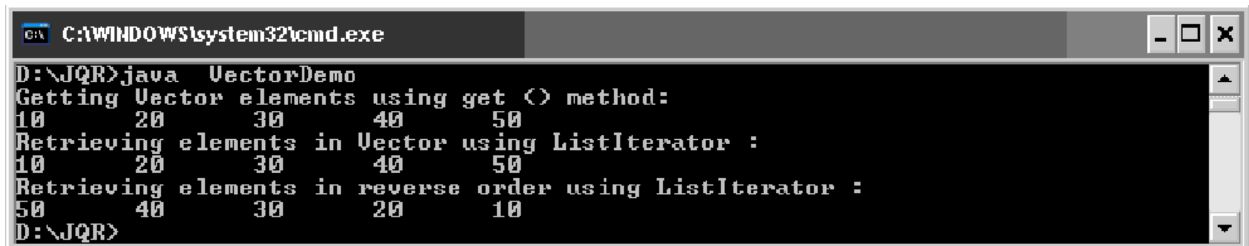
int indexOf (Object obj)	This method returns the index of the first occurrence of the specified element in the Vector, or -1 if the Vector does not contain the element.
int lastIndexOf (Object obj)	This method returns the index of the last occurrence of the specified element in the Vector, or -1 if the Vector does not contain the element.
Object[] toArray ( )	This method converts the Vector into an array of Object class type. All the elements of the Vector will be stored into the array in the same sequence.
int capacity ()	This method returns the current capacity of the Vector.

Program : Write a program that shows the use of Vector class.

```
import java.util.*;
//Vector Demo
class VectorDemo
{  public static void main(String args[])
  {  Vector <Integer> v = new Vector<Integer> ();
    int x[] = {10,20,30,40,50};
    //When x[i] is stored into v below, x[i] values are converted into Integer Objects
                    //and stored into v. This is auto boxing.
    for (int i = 0; i<x.length; i++)
      v.add(x[i]);
    System.out.println ("Getting Vector elements using get () method: ");
    for (int i = 0; i<v.size(); i++)
      System.out.print (v.get (i) + "\t");
    System.out.println ("\nRetrieving elements in Vector using ListIterator :");
    ListIterator lit = v.listIterator ();
    while (lit.hasNext () )
      System.out.print (lit.next () + "\t");
    System.out.println ("\nRetrieving elements in reverse order using ListIterator :");
    while (lit.hasPrevious () )
      System.out.print (lit.previous () + "\t");
```

```
  }
}
```

Output:



HashMap Class: HashMap is a collection that stores elements in the form of key-value pairs. If key is provided later its corresponding value can be easily retrieved from the HAshMap. Key should be unique. HashMap is not synchronized and hence while using multiple threads on HashMap object, we get unreliable results.

We can write HashMap class as: class HashMap<K, V>

For example to store a String as key and an integer object as its value, we can create the

HashMap as: HashMap<String, Integer> hm = new HashMap<String, Integer> ();

The default initial capacity of this HashMap will be taken as 16 and the load factor as 0.75. Load factor represents at what level the HashMap capacity should be doubled. For example, the product of capacity and load factor = 16 * 0.75 = 12. This represents that after storing 12th key-value pair into the HashMap, its capacity will become 32.

HashMap Class Methods:

Method	Description
value put (key, value)	This method stores key-value pair into the HashMap.
value get (Object key)	This method returns the corresponding value when key is given. If the key does not have a value associated with it, then it returns null.
Set<K> keyset()	This method, when applied on a HashMap converts it into a set where only keys will be stored.
Collection <V> values()	This method, when applied on a HashMap object returns all the values of the HashMap into a Collection object.
value remove (Object key)	This method removes the key and corresponding value from the HashMap.
void clear ()	This method removes all the key-value pairs from the map.
boolean isEmpty ()	This method returns true if there are no key-value pairs in the HashMap.
int size ()	This method returns number of key-value pairs in the HashMap.

Program : Write a program that shows the use of HashMap class.

```
//HashMap Demo
import java.util.*;
class HashMapDemo
{  public static void main(String args[])
  {  HashMap<Integer, String> hm = new HashMap<Integer, String> ();
    hm.put (new Integer (101),"Naresh");
    hm.put (new Integer (102),"Rajesh");
    hm.put (new Integer (103),"Suresh");
    hm.put (new Integer (104),"Mahesh");
    hm.put (new Integer (105),"Ramesh");
    Set<Integer> set = new HashSet<Integer>();
    set = hm.keySet();
    System.out.println (set);
  }
}
```

Output:

Hashtable Class: Hashtable is a collection that stores elements in the form of key-value pairs. If key is provided later its corresponding value can be easily retrieved from the HAshtable. Keys should be unique. Hashtable is synchronized and hence while using multiple threads on Hashtable object, we get reliable results.

We can write Hashtable class as: class Hashtable<K,V>

For example to store a String as key and an integer object as its value, we can create the

Hashtable as: Hashtable<String, Integer> ht = new Hashtable<String, Integer> ();

The default initial capacity of this Hashtable will be taken as 11 and the load factor as 0.75. Load factor represents at what level the Hashtable capacity should be doubled. For

example, the product of capacity and load factor = 11 * 0.75 = 8.25. This represents that after storing 8th key-value pair into the Hashtable, its capacity will become 22.

Hashtable Class Methods:

Method	Description
value put(key, value)	This method stores key-value pair into the Hashtable.
value get(Object key)	This method returns the corresponding value when key is given. If the key does not have a value associated with it, then it returns null.
Set<K> keyset()	This method, when applied on a Hashtable converts it into a set where only keys will be stored.
Collection <V> values()	This method, when applied on a Hashtable object returns all the values of the Hashtable into a Collection object.
value remove(Object key)	This method removes the key and corresponding value from the Hashtable.
void clear()	This method removes all the key-value pairs from the Hashtable.
boolean isEmpty()	This method returns true if there are no key-value pairs in the Hashtable.
int size()	This method returns number of key-value pairs in the Hashtable.

Program : Write a program that shows the use of Hashtable class.

```
//Hashtable Demo

import java.util.*;

class HashtableDemo

{  public static void main(String args[])

  {

    Hashtable<Integer, String> ht = new Hashtable<Integer, String> ();

    ht.put (new Integer (101),"Naresh");

    ht.put (new Integer (102),"Rajesh");

    ht.put (new Integer (103),"Suresh");

    ht.put (new Integer (104),"Mahesh");

    ht.put (new Integer (105),"Ramesh");

    Enumeration e = ht.keys ();

    while ( e.hasMoreElements () )

    {
```

```
      Integer i1 = (Integer) e.nextElement ();
      System.out.println (i1 + "\t" + ht.get (i1));
    }
  }
}
```

Output:

Arrays Class: Arrays class provides methods to perform certain operations on any single dimensional array. All the methods of the Arrays class are static, so they can be called in the form of Arrays.methodname ().

Arrays Class Methods:

Method	Description
static void sort (array)	This method sorts all the elements of an array into ascending order. This method internally uses QuickSort algorithm.
static void sort (array, int start, int end)	This method sorts the elements in the range from start to end within an array into ascending order.
static int binarySearch (array, element)	This method searches for an element in the array and returns its position number. If the element is not found in the array, it returns a negative value. Note that this method acts only on an array which is sorted in ascending order. This method internally uses BinarySearch algorithm.
static boolean equals (array1, array2)	This method returns true if two arrays, that is array1 and array2 are equal, otherwise false.
static array copyOf (source-array, int n)	This method copies n elements from the source-array into another array and returns the array.
static void fill (array, value)	This method fills the array with the specified value. It means that all the elements in the array will receive that value.

Program : Write a program to sort given numbers using sort () method of Arrays Class.

```
import java.util.*;

//Arrays Demo

class ArraysDemo

{  public static void main(String args[])

  {

int x[] = {40,50,10,30,20};

    Arrays.sort( x );

    for (int i=0;i<x.length;i++)

      System.out.print(x[i] + "\t");

  }

}
```

Output:

StringTokenizer: The StringTokenizer class is useful to break a String into small pieces called tokens. We can create an object to StringTokenizer as:

StringTokenizer st = new StringTokenizer (str, "delimeter");

StringTokenizer Class Methods:

Method	Description
String nextToken()	Returns the next token from the StringTokenizer
boolean hasMoreTokens()	Returns true if token is available and returns false if not available
int countTokens()	Returns the number of tokens available.

Program : Write a program that shows the use of StringTokenizer object.

```
//cutting the String into tokens
import java.util.*;
class STDemo
{
public static void main(String args[])
  {  //take a String
    String str = "Java is an OOP Language";
    //brake wherever a space is found
    StringTokenizer st = new StringTokenizer (str," ");
    //retrieve tokens and display
    System.out.println ("The tokens are: ");
    while ( st.hasMoreTokens () )
    {
      String s = st.nextToken ();
      System.out.println (s );
    }
  }
}
```

Output:

Calendar: This class is useful to handle date and time. We can create an object to Calendar class as: Calendar cl = Calendar.getInstance ();

Calendar Class Methods:

Method	Description
int get(Constant)	This method returns the value of the given Calendar constant. Examples of Constants are Calendar.DATE, Calendar.MONTH, Calendar.YEAR, Calendar.MINUTE, Calendar.SECOND, Calendar.Hour
void set(int field, int value)	This method sets the given field in Calendar Object to the given value. For example, cl.set(Calendar.DATE,15);
String toString()	This method returns the String representation of the Calendar object.
boolean equals(Object obj)	This method compares the Calendar object with another object obj and returns true if they are same, otherwise false.

Program : Write a program to display System time and date.

```
//To display system time and date
import java.util.*;
class Cal
{  public static void main(String args[])
  {  Calendar cl = Calendar.getInstance ();
    //Retrieve Date
    int dd = cl.get (Calendar.DATE);
    int mm = cl.get (Calendar.MONTH);
    ++mm;
    int yy = cl.get (Calendar.YEAR);
    System.out.println ("Current Date is : " + dd + "-" + mm + "-" + yy );
    //Retrieve Time
    int hh = cl.get (Calendar.HOUR);
    int mi = cl.get (Calendar.MINUTE);
    int ss = cl.get (Calendar.SECOND);
    System.out.println ("Current Time is : " + hh + ":" + mi + ":" +ss);
  }
```

}

Output:

Date Class: Date Class is also useful to handle date and time. Once Date class object is created, it should be formatted using the following methods of DateFormat class of java.text package.We can create an object to Date class as: Date dd = new Date ();

Once Date class object is created, it should be formatted using the methods of DateFormat class of java.text package.

DateFormat class Methods:

- DateFormat fmt = DateFormat.getDateInstance(formatconst, region);

This method is useful to store format information for date value into DateFormat object fmt.

- DateFormat fmt = DateFormat.getTimeInstance(formatconst, region);

This method is useful to store format information for time value into DateFormat object fmt.

- DateFormat fmt = DateFormat.getDateTimeInstance(formatconst, formatconst, region);

This method is useful to store format information for date value into DateFormat object fmt.

Formatconst	**Example (region=Locale.UK)**
DateFormat.FULL	03 september 2007 19:43:14 O'Clock GMT + 05:30
DateFormat.LONG	03 september 2007 19:43:14 GMT + 05:30
DateFormat.MEDIUM	03-sep-07 19:43:14
DateFormat.SHORT	03/09/07 19:43

Program : Write a program that shows the use of Date class.

```
//Display System date and time using Date class

import java.util.*;

import java.text.*;

class MyDate

{
```

```
public static void main(String args[])
 {  Date d = new Date ();
   DateFormat fmt = DateFormat.getDateTimeInstance (DateFormat.MEDIUM,
DateFormat.SHORT, Locale.UK);
   String str = fmt.format (d);
   System.out.println (str);
 }
}
```

Output:

Design Goals

The main design goal was to produce an API that was reasonably small, both in size, and, more importantly, in "conceptual weight." It was critical that the new functionality not seem alien to current Java programmers; it had to augment current facilities, rather than replacing them. At the same time, the new API had to be powerful enough to provide all the advantages described above.

To keep the number of core interfaces small, the interfaces do not attempt to capture such subtle distinctions as mutability, modifiability, and resizability. Instead, certain calls in the core interfaces are *optional*, allowing implementations to throw an UnsupportedOperationException to indicate that they do not support a specified optional operation. Of course, collection implementers must clearly document which optional operations are supported by an implementation.

To keep the number of methods in each core interface small, an interface contains a method only if either:

1. It is a truly *fundamental operation*: a basic operations in terms of which others could be reasonably defined,
2. There is a compelling performance reason why an important implementation would want to override it.

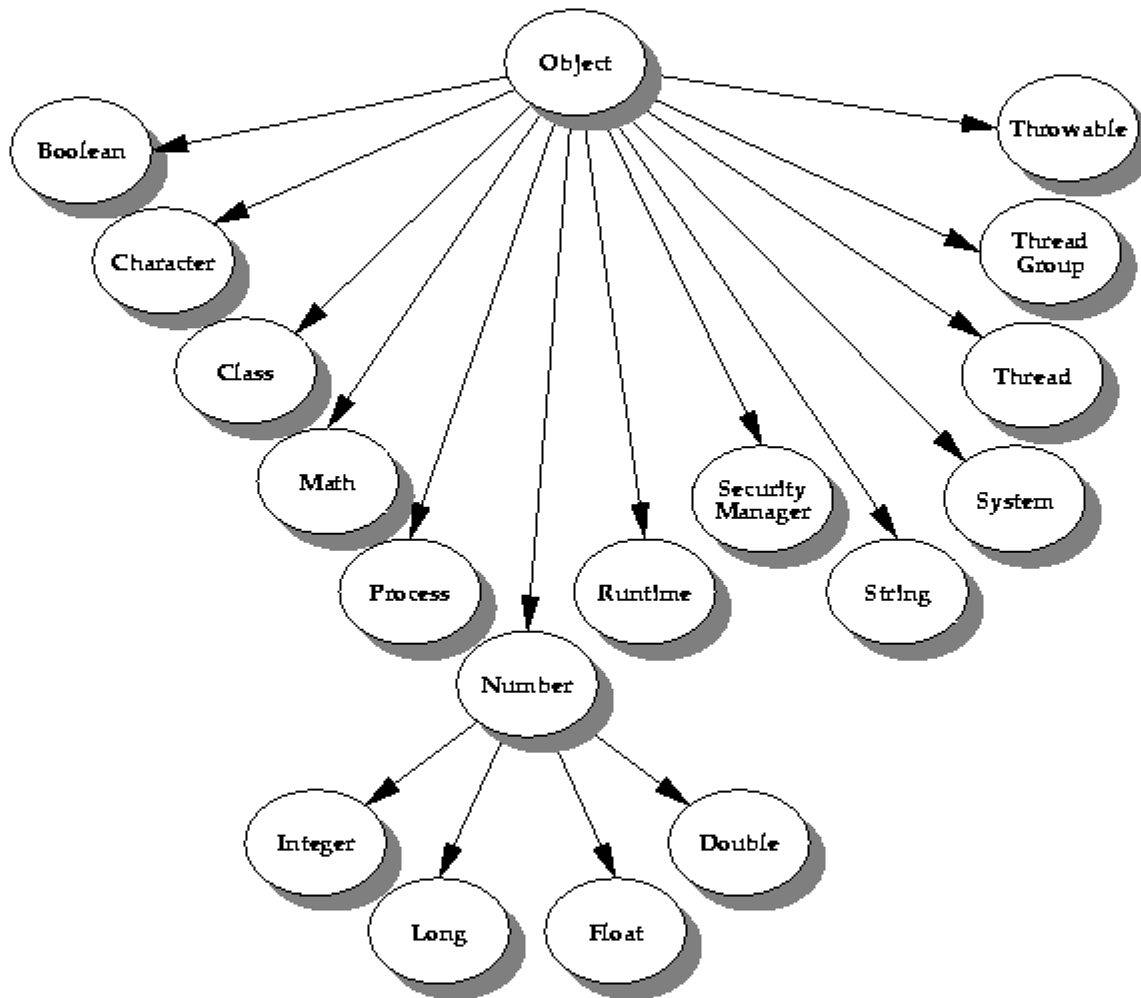
It was critical that all reasonable representations of collections interoperate well. This included arrays, which cannot be made to implement the Collection interface directly without changing the language. Thus, the framework includes methods to allow collections to be dumped into arrays, arrays to be viewed as collections, and maps to be viewed as collections.

MULTI THREADING

Multi Threading:

Java Language Classes

The java.lang package contains the collection of base types (language types) that are always imported into any given compilation unit. This is where you'll find the declarations of Object (the root of the class hierarchy) and Class, plus threads, exceptions, wrappers for the primitive data types, and a variety of other fundamental classes.



This picture illustrates the classes in java.lang, excluding all the exception and error classes.

Note the Boolean, Character, and Number classes--these classes are "wrapper" classes for the primitive types. You use these classes in applications where the primitive types must be stored as objects. Note also the Throwable class--this is the root class for all exceptions and errors.

Simply put, **a *thread* is a program's path of execution**. Most programs written today run as a single thread, causing problems when multiple events or actions need to occur at the same time. Let's say, for example, a program is not capable of drawing pictures while reading keystrokes. The program must give its full attention to the keyboard input lacking the ability to handle more than one event at a time. **The ideal solution to this problem is the seamless execution of two or more sections of a program at the same time. Threads allow us to do this.**

Multiprogramming is a rudimentary form of parallel processing in which several programs are run at the same time on a uniprocessor.Since there is only one processor, there can be no true simultaneous execution of different programs. Instead, the operating system executes part of one program, then part of another, and so on. To the user it appears that all programs are executing at the same time.

Multitasking, in an operating system, is allowing a user to perform more than one computer task (such as the operation of an application program) at a time. The operating system is able to keep track of where you are in these tasks and go from one to the other without losing information

Multithreading is the ability of a program to manage its use by more than one thread at a time .Dispatchable atomic units of the program are executing simultaneously.

Multithreaded applications deliver their potent power by running many threads concurrently within a single program. From a logical point of view, multithreading means multiple lines of a single program can be executed at the same time, however, it is not the same as starting a program twice and saying that there are multiple lines of a program being executed at the same time. In this case, the operating system is treating the programs as two separate and distinct processes.

Uses of Threads:

- Threads are used in designing serverside programs to handle multiple clients at a time.
- Threads are used in games and animations.
- We can reduce the idle time of processor.
- Performance of processor is improved.
- Reduces interferences between execution and user interface.

The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the main thread of your program, because it is the one that is executed when your program begins.

The main thread is important for two reasons:

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

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

Although the main thread is created automatically when your program is started, it can be controlled through a Thread object. To do so, you must obtain a reference to it by calling the method currentThread( ), which is a public static member of Thread. Its general form is shown here:

static Thread currentThread( )

This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

Let's begin by reviewing the following example:

Program : Write a program to know the currently running Thread

```
//Currently running thread

class Current

{

public static void main(String args[])

  {

System.out.println ("This is first statement");

   Thread t = Thread.currentThread ();

   System.out.println ("Current Thread: " + t);

   System.out.println ("Its name: " + t.getName ());

   System.out.println ("Its priority:" + t.getPriority ());

  }

}
```

Output:

```
D:\JQR>javac Current.java

D:\JQR>java  Current
This is first statement
Current Thread: Thread[main,5,main]
Its name: main
Its priority:5

D:\JQR>
```

Differences between multi threading and multitasking:

MULTI THREADING	MULTI TASKING
1). More than one thread running simultaneously	1). More than one process running simultaneously
2). Its part of a program	2). Its a program.
3).it is a light-weight process.	3). It is a heavy-weight process.
4). Threads are divided into sub threads	4). Process is divided into threads.
5). Within the process threads are communicated.	5). Inter process communication is difficulty
6). Context switching between threads is cheaper.	6). Context switching between process is costly
7). It is controlled by Java(JVM)	7). It is controlled by operating System.
8).It is a specialized form of multi tasking	8). It is a generalized form of multi threading.
9). Example: java's automatic garbage collector.	8). Program compilation at command prompt window and preparing documentation at MS-Office.

What is Garbage Collection?
Garbage Collection in computer science is a form of automatic memory management. The garbage collector, or just collector, attempts to reclaim garbage, or memory occupied by objects that are no longer in use by the program. Garbage collection does not traditionally manage limited resources other than memory that typical programs use, such as network sockets, database handles, user interaction windows, and file and device descriptors.

History:
Garbage collection was invented by John McCarthy around 1959 to solve problems in Lisp.

Basic principle of Garbage Collection:

The basic principles of garbage collection are:

- Find data objects in a program that cannot be accessed in the future
- Reclaim the resources used by those objects

It is kind of interesting to know how the objects without reference are found. Java normally finds all the objects that have reference and then regards rest of the objects are reference less – which is in fact a very smart way of finding the unreferenced java objects.

Types of Java Garbage Collectors/Garbage Collection Algorithms:
On J2SE 5.0 and above, one can normally find the following types of Java Garbage collectors that the programmers can normally choose to do a garbage collection through JVM Parameters.

The Serial Collector:

- JVM Option Parameter: -XX:+UseSerialGC

The Throughput Collector or The Parallel Collector:

- JVM Option Parameter: -XX:+UseParallelGC
- Young Generation GC done in parallel threads
- Tenured Generation GC done in serial threads.

Parallel Old Generation Collector:

- JVM Option Parameter: -XX:+UseParallelOldGC
- Certain phases of an 'Old Generation' collection can be performed in parallel, speeding up a old generation collection.

The Concurrent Low Pause Collector:

- JVM Option Parameter -Xincgc or -XX:+UseConcMarkSweepGC
- The concurrent collector is used to collect the tenured generation and does most of the collection concurrently with the execution of the application. The application is paused for short periods during the collection.
- A parallel version of the young generation copying collector is used with the concurrent collector.
- The concurrent low pause collector is used if the option -XX:+UseConcMarkSweepGC is passed on the command line.
- -XX:+UseConcMarkSweepGC -XX:+UseParNewGC
  - Selects the Concurrent Mark Sweep collector.
  - This collector may deliver better response time properties for the application (i.e., low application pause time).
  - It is a parallel and mostly-concurrent collector and and can be a good match for the threading ability of an large multi-processor systems.

The incremental (sometimes called train) low pause collector:
JVM Option Parameter: -XX:+UseTrainGC
This collector has not changed since the J2SE Platform version 1.4.2 and is currently not under active development.
It will not be supported in future releases.

Note that -XX:+UseParallelGC should not be used with -XX:+UseConcMarkSweepGC .

The argument parsing in the J2SE Platform starting with version 1.4.2 should only allow legal combination of command line options for garbage collectors, but earlier releases may not detect all illegal combination and the results for illegal combination are unpredictable.

Benefits of Garbage Collection:
Garbage collection frees the programmer from manually dealing with memory deallocation. As a result, certain categories of bugs are eliminated or substantially reduced:

- Dangling pointer bugs, which occur when a piece of memory is freed while there are still pointers to it, and one of those pointers is then used. By then the memory may have been re-assigned to another use, with unpredictable results.
- Double free bugs, which occur when the program tries to free a region of memory that has already been freed, and perhaps already been allocated again.
- Certain kinds of memory leaks, in which a program fails to free memory occupied by objects that will not be used again, leading, over time, to memory exhaustion.

Disadvantages of Garbage Collection:

- Consumes computing resources in deciding what memory is to be freed, reconstructing facts that may have been known to the programmer often leading to decreased or uneven performance.
- Interaction with memory hierarchy effects can make this overhead intolerable in circumstances that are hard to predict or to detect in routine testing.
- The moment when the garbage is actually collected can be unpredictable, resulting in stalls scattered throughout a session.
- Memory may leak despite the presence of a garbage collector, if references to unused objects are not themselves manually disposed of. This is described as a logical memory leak.The belief that garbage collection eliminates all leaks leads many programmers not to guard against creating such leaks.
- In virtual memory environments, it can be difficult for the garbage collector to notice when collection is needed, resulting in large amounts of accumulated garbage, a long, disruptive collection phase, and other programs' data swapped out.
- Garbage collectors often exhibit poor locality (interacting badly with cache and virtual memory systems), occupy more address space than the program actually uses at any one time, and touch otherwise idle pages.
- Garbage collectors may cause thrashing, in which a program spends more time copying data between various grades of storage than performing useful work.

Thread Life Cycle:

Thread States (Life-Cycle of a Thread): The life cycle of a thread contains several states. At any time the thread falls into any one of the states.

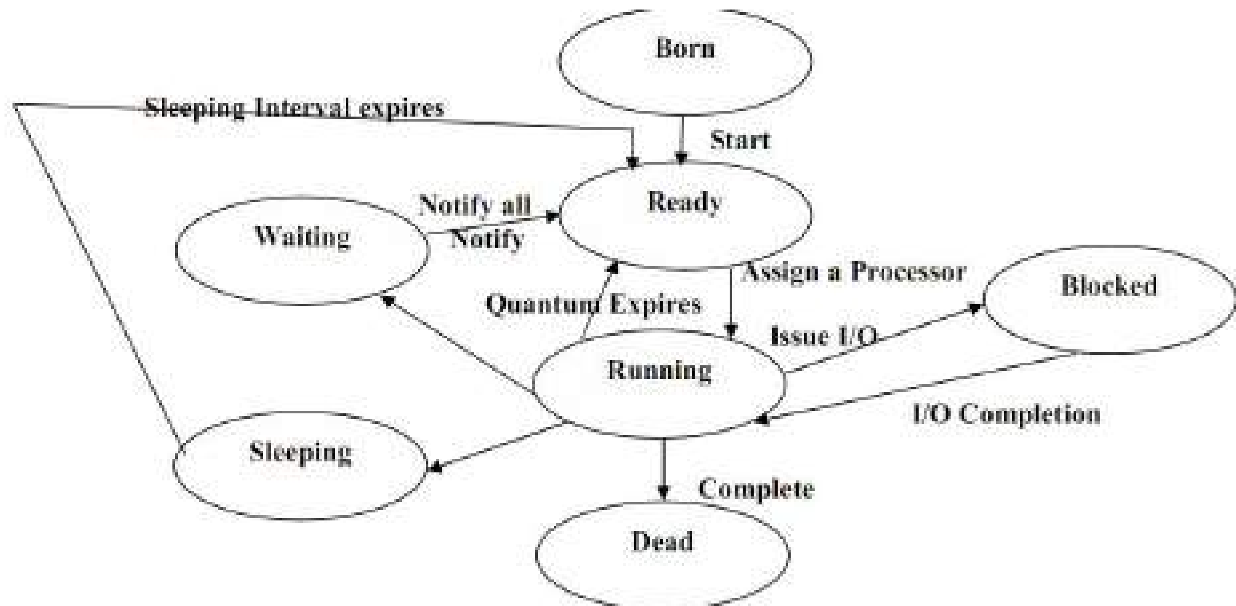


Fig: Thread Life Cycle

- The thread that was just created is in the born state.
- The thread remains in this state until the threads start method is called. This causes the thread to enter the ready state.
- The highest priority ready thread enters the running state when system assigns a processor to the thread i.e., the thread begins executing.
- When a running thread calls wait the thread enters into a waiting state for the particular object on which wait was called. Every thread in the waiting state for a given object becomes ready on a call to notify all by another thread associated with that object.
- When a sleep method is called in a running thread that thread enters into the suspended (sleep) state. A sleeping thread becomes ready after the designated sleep time expires. A sleeping thread cannot use a processor even if one is available.
- A thread enters the dead state when its run () method completes (or) terminates for any reason. A dead thread is eventually be disposed of by the system.
- One common way for a running thread to enter the blocked state is when the thread issues an input or output request. In this case a blocked thread becomes ready when the input or output waits for completes. A blocked thread can't use a processor even if one is available.

Creating Threads:

We know that in every java program, there is a main thread available already. Apart from this main thread, we can also create our owen threads in a program. The following steps should be used.

- Write a class that extends Thread class or implements Runnable interface this is available in lang package.
  Class Myclass extends Thread (or)
  Class Myclass implements Runnable
- Write public void run () method in that class. This is the method by default executed by any thread.
  ```
  Public void run()
  {
          Statements;
  }
  ```
- Create an object to my class, so that the run() method is available for execution.
  Myclass obj=new Myclass();
- Create a thread and attach it to the object.
  Thread t=new Thread(obj);
  or

  To create a Thread, we can use the following forms:

  ```
  Thread t1 = new Thread ();
  Thread t2 = new Thread (obj);
  Thread t3 = new Thread (obj, "thread-name");
  ```
- Start running the threads.
  t.start();

Syntactical code for creating and running the thread:

```
Class Myclass extends Thread (or)
Class Myclass implements Runnable
{
Public void run()
{
        Statements;
}

}
Class Demo
{
 Public static void main(String args[])throws InterruptedException
{
Myclass obj=new Myclass();
```

```
Thread t=new Thread(obj);
t.start();
}
}
```

Thread Class Methods:

- To know the currently running thread: *Thread t = Thread.currentThread ();*
- To start a thread: *t.start ();*
- To stop execution of a thread for a specific time: *Thread.sleep (milliseconds);*
- To get the name of the thread: *String name = t.getName ();*
- To set the new name to the thread: *t.setName ("New Name");*
- To get the priority of the thread: *int priority = t.getPriority();*
- To set the priority of the thread: *t.setPriority (int priority);*
- Thread priorities can change from 1 to 10. We can also use the following constants to represent priorities: *Thread.MAX_PRIORITY value is 10*

  Thread.MIN_PRIORITY value is 1

  Thread.NORM_PRIORITY value is 5

- To test if a thread is still alive: *t.isAlive () returns true/false*
- To wait till a thread dies: *t.join ();*
- To send a notification to a waiting thread: *obj.notify ();*
- To send notification to all waiting threads: *obj.notifyAll ();*
- To wait till the obj is released (till notification is sent): *obj.wait ();*

Method	Description
final boolean isAlive( )	Returns **true** if the thread is still active. Otherwise, it returns **false**.
final boolean isDaemon( )	Returns **true** if the thread is a daemon thread. Otherwise, it returns **false**.
boolean isInterrupted( )	Returns **true** if the thread is interrupted. Otherwise, it returns **false**.
final void join( ) throws InterruptedException	Waits until the thread terminates.
final void join(long *milliseconds*) throws InterruptedException	Waits up to the specified number of milliseconds for the thread on which it is called to terminate.
final void join(long *milliseconds*, int *nanoseconds*) throws InterruptedException	Waits up to the specified number of milliseconds plus nanoseconds for the thread on which it is called to terminate.
void run( )	Begins execution of a thread.
void setContextClassLoader(ClassLoader *cl*)	Sets the class loader that will be used by the invoking thread to *cl*.
final void setDaemon(boolean *state*)	Flags the thread as a daemon thread.
static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler *e*)	Sets the default uncaught exception handler to *e*.
final void setName(String *threadName*)	Sets the name of the thread to that specified by *threadName*.
final void setPriority(int *priority*)	Sets the priority of the thread to that specified by *priority*.
void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler *e*)	Sets the invoking thread's default uncaught exception handler to *e*.
static void sleep(long *milliseconds*) throws InterruptedException	Suspends execution of the thread for the specified number of milliseconds.
static void sleep(long *milliseconds*, int *nanoseconds*) throws InterruptedException	Suspends execution of the thread for the specified number of milliseconds plus nanoseconds.
void start( )	Starts execution of the thread.
String toString( )	Returns the string equivalent of a thread.
static void yield( )	The calling thread yields the CPU to another thread.

Program : Write a program to create and run a Thread.

```
//creating and running a Thread
class MyThread extends Thread
{  public void run ()
  {  for (int i = 0;i<100;i++)
```

```
    {
      System.out.print (i + "\t");
    }
  }
}
class TDemo
{  public static void main(String args[])
  {  MyThread obj = new MyThread ();
    Thread t = new Thread (obj);
    t.start ();
  }
}
```

Output:

```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac TDemo.java

D:\JQR>java  TDemo
0       1       2       3       4       5       6       7       8       9
10      11      12      13      14      15      16      17      18      19
20      21      22      23      24      25      26      27      28      29
30      31      32      33      34      35      36      37      38      39
40      41      42      43      44      45      46      47      48      49
50      51      52      53      54      55      56      57      58      59
60      61      62      63      64      65      66      67      68      69
70      71      72      73      74      75      76      77      78      79
80      81      82      83      84      85      86      87      88      89
90      91      92      93      94      95      96      97      98      99

D:\JQR>
```

As a result, the numbers will be displayed starting from 1 to 99 using a for loop. If u want to terminate the program in the middle, u can press Ctrl+C from the keyboard. This leads to abnormal program termination. It means the entire program is terminated, not just the thread.

If we want to terminate only the thread that is running the code inside run() method, we should devise our own mechanism.if we press Ctrl+C, we are abnormally terminating the program. This is dangerous. Abnormal program termination may cause loss of data and lead to unreliable results. So,we should terminate the thread only, not the program.hown can we terminate the thread smoothly is the question now.

Terminating the thread:

A thread will terminate automatically when it comes out of run() method. To terminate the thread on our own logic. For the following steps can be used

1.Create the Boolean type variable and initialize it to false.

Boolean stop=false;

3.Let us assume that we want to terminate the thread when the user presses <Enter> key. So, when the user presses that button, make the Boolean type variable as true.

Stop=true;

3.Check this variable in run() method and when it is true, make the thread return from the run() method.

```
Public void run()
{
        If(stop==true) return;
}
```

Program to showing how to terminate the thread by pressing the enter button

```
import java.io.*;
class MyThread implements Runnable
{
        boolean stop=false;
        public void run ()
        {
                for (int i = 0;i<=100000;i++)
                {
                        System.out.print (i + "\t");
                        if(stop==true) return;  //come out of run
                }
        }
}
```

```
class TDemo
{
    public static void main(String args[]) throws IOException
    {
        MyThread obj = new MyThread ();
        Thread t = new Thread (obj);
        t.start ();//stop the thread when enter key is pressed
        System.in.read();
        obj.stop=true;
    }
}
```

Output:

```
0       1       2…..
100     101     102….
Press <Enter> to stop the thread at any time.
```

What is the difference between 'extends thread' and 'implements Runnable' ? which one is advantageous?

'extends thread' and 'implements Runnable'-both are functionally same. But when we write extends Thread, there is no scope toextend another class, as multiple inheritance is not supported in java.

Class MyClass extends Thread,AnotherClass //invalid

If we write implements Runnable, then still there is scope to extend another class.

Class MyClass extends AnotherClass implements Runnable //valid

This is definitely advantageous when the programmer wants to use threads and also wants to access the features of another class.

Single tasking using a thread:

A thread can be employed to execute one task at a time. Suppose there are 3 tasks to be executed. We can create a thread and pass the 3 tasks one by one to the thread. For this purpose, we can write all these tasks separately in separate methods; task1(), task2(), task3(). Then these methods should be called from run() method, one by one. Remember, a thread executes only the code inside the run() method. It can never execute other methods unless they are called from run().

Note: public void run() method is executed by the thread by default.

```
//single tasking using a thread
class MyThread implements Runnable
{
	public void run()
	{
		//executes tasks one by one by calling the methods.
		task1();
		task2();
		task3();
	}
	void task1()
	{
		System.out.println("this is task1");
	}
	void task2()
	{
		System.out.println("this is task2");
	}
	void task3()
	{
		System.out.println("this is task3");
	}
}
class Sin
{
	public static void main(String args[])
```

```
        {
                MyThread obj=new MyThread();
                Thread t1=new Thread(obj);
                t1.start();
        }
}
```

Output: java Sin

```
This is task1()
This is task2()
This is task3()
```

In this program, a single thread t1 is used to execute three tasks.

Multi Tasking Using Threads:

In multi taskin, several tasks are executed at a time. For this purpose, we need more than one thread. For example, to perform 2 tasks, we can take 2 threads and attach them to the 2 tasks. Then those tasks are simultaneously executed by the two threads. Using more than one thread is called 'multi threading'.

Program : Write a program to create more than one thread.

```
//using more than one thread is called Multi Threading
class Theatre extends Thread
{  String str;
  Theatre (String str)
  {  this.str = str;
  }
public void run()
  {  for (int i = 1; i <= 10 ; i++)
    {  System.out.println (str + " : " + i);
      try
      {  Thread.sleep (2000);
```

```
      }
      catch (InterruptedException ie)   {    ie.printStackTrace ();     }
    }
  }
}
class TDemo1
{  public static void main(String args[])
  {  Theatre obj1 = new Theatre ("Cut Ticket");
    Theatre obj2 = new Theatre ("Show Chair");
    Thread t1 = new Thread (obj1);
    Thread t2 = new Thread (obj2);
    t1.start ();
    t2.start ();
  }
}
```

Output:

```
C:\WINDOWS\system32\command.com
D:\JQR>javac TDemo1.java

D:\JQR>java TDemo1
Cut Ticket : 1
Show Chair : 1
Cut Ticket : 2
Show Chair : 2
Cut Ticket : 3
Show Chair : 3
Cut Ticket : 4
Show Chair : 4
Cut Ticket : 5
Show Chair : 5
Cut Ticket : 6
Show Chair : 6
Cut Ticket : 7
Show Chair : 7
Cut Ticket : 8
Show Chair : 8
Cut Ticket : 9
Show Chair : 9
Cut Ticket : 10
Show Chair : 10

D:\JQR>
```

In the preceding example, we have used 2 threads on the 2 objects of TDemo1 class. First we have taken a String variable str in Theatre class. Then we passed two strings- cut ticket and show chair into that variable from TDemo1 class. When t1. start () is executed, it starts execution run () method code showing cut ticket. Note that in run () method, we used: Thread. sleep (2000) is a static method in Thread class, which is used to suspend execution of a thread for some specified milliseconds. Since this method can throw InterruptedException, we caught it in catch block. When Thread t1 is suspended immediately t2. start () will make the thread t2 to execute and when it encounters Thread.sleep(2000), it will suspend for specified time meanwhile t1 will

get executed respectively. In this manner, both the threads are simultaneously executed.synchronizing threads, daemon threads, thread groups.

Multiple Threads Acting on Single Object:

First letus see why 2 threads shouldshare same object(samerun()method). We write an object to represent one task. If there is a different task,we take another object. When two people (threads) want to perform same task then they need same object (run () method) to be executed each time. Take the case of railway reservation. Every day several people want reservation of a berth for them. The procedure to reserve the berth is same for all the people. So we need some object with same run () method to be executed repeatedly for all the people (threads).

Let us think that only one berth is available in a train and two passengers (threads) are asking for that berth in two different counters. The clerks at different counters sent a request to the server to allot that berth to their passengers. Let us see now to whom that berth is allotted.

Program : Write a program to create multiple threads and make the threads to act on single object.

```
//Thread unsafe –Two threads acting  on same object.

class Reserve implements Runnable

{ //available berths are 1

            int available = 1;

            int wanted;

           //accept wanted berths at runtime

          Reserve (int i)

          {

                    wanted = i;

           }

           public void run()

           {//display available  berths

           System.out.println ("Number of berths available: " + available);

//if available berths more  thanwanted  betrhs

           if ( available >= wanted)

          {//get the name of the passenger

                    String name = Thread.currentThread ().getName ();
```

```
                System.out.println (wanted + " berths alloted to: " + name);
                try
                {
                    Thread.sleep (2000); // wait for priniting the ticket
                     available = available - wanted;
               //update the no.of  available berths
                }  catch (InterruptedException ie)
               {  ie.printStackTrace ();    }
           }
           else
           {    System.out.println ("Sorry, no berths available");
           }
       }
}
class UnSafe
{
           public static void main(String args[])
           {
                Reserve obj = new Reserve (1);
                Thread t1 =new Thread (obj);
                Thread t2 = new Thread (obj);
                t1.setName ("First Person");
                t2.setName ("Second Person");
               t1.start ();
               t2.start ();
           }
}
```

Output: java UnSafe

Number of berths available:1

1 berth is allotted to First person

Number of berths available:1

1 berth is allotted to Second person

Please observe the output in the preceding program. It is absurd. It has allotted the same berth to both the passengers. Since both the threads are acting on the same object simultaneously, then the result is unreliable.

What is the solution for this problem?

Ans: Thread Synchronization

Synchronizing Threads or Thread Synchronization or Thread Safe: When a thread is acting on an object preventing other threads from acting on the same object is called Thread Synchronization or Thread Safe. The object on which the threads are synchronized is called ‘synchronized object’.

The Object on which the Threads are synchronized is called synchronized object or Mutex (Mutually Exclusive Lock).Synchronized object is like a locked object, locked on a thread. It is like a room with only one door. A person has entered the room and locked form it from behind. The second person who wants to enter the room should wait till the first person comes out. In this way, a thread also locks the object after entering it. Then the next thread cannot enter it till the first thread comes out. This means the object is locked mutually on threads. So, this object is called ‘mutex’.

Thread synchronization is done in two ways:

- Using synchronized block we can synchronize a block of statements.

e.g.: *synchronized (obj)*

```
{
  statements;
}
```

Here, object represents the object to be locked or synchronized. The statements inside the synchronized block are all available to only one thread at a time. They are not available to more than one thread simultaneously.

- To synchronize an entire method code we can use synchronized word before method name

e.g.: *synchronized void method ()*

```
{
    Stmts;
}
```

Now the statements inside the method are not available to more than one thread at a time. This method code is synchronized.

Write a program to thread synchronization by using synchronized block.

```
//Thread synchronization- Two threads acting on same object

//Multiple Threads acting on single object

class Reserve implements Runnable

{  int available = 1;

 int wanted;

 Reserve (int i)

 {  wanted = i;

 }

 public void run()

 {  synchronized (this)

   {  System.out.println ("Number of berths available: " + available);

    if ( available >= wanted)

    {  String name = Thread.currentThread ().getName ();

     System.out.println (wanted + " berths alloted to: " + name);

     try

     {  Thread.sleep (2000); // wait for priniting the ticket

       available = available - wanted;

     }

     catch (InterruptedException ie)

     {  ie.printStackTrace ();    }

    }
```

```
      else
      {    System.out.println ("Sorry, no berths available");
      }
    }
  }
}
class Safe
{  public static void main(String args[])
  {  Reserve obj = new Reserve (1);
    Thread t1 =new Thread (obj);
    Thread t2 = new Thread (obj);
    t1.setName ("First Person");
    t2.setName ("Second Person");
    t1.start ();
    t2.start ();
  }
}
```

Output:

```
C:\WINDOWS\system32\command.com
D:\JQR>javac Safe.java

D:\JQR>java Safe
Number of berths available: 1
1 berths alloted to: First Person
Number of berths available: 0
Sorry, no berths available

D:\JQR>
```

Write a program to thread synchronization by using synchronized keyword before the method name

/ Write a Java program that creates three threads. First thread displays “Good Morning” every one second, the second thread displays “Hello” every two seconds and the third thread displays “Welcome” every threeseconds. */**

```
class A extends Thread
{
synchronized public void run()
{
try
{
while(true)
{
sleep(10);
System.out.println("good morning");
}
}
catch(Exception e)
{
}
}
}
class B extends Thread
{
synchronized public void run()
{
try
{
while(true)
{
```

```
sleep(20);
System.out.println("hello");
}
}
catch(Exception e)
{        }
}
}
class C extends Thread
{
synchronized public void run()
{
try
{
while(true)
{
sleep(30);
System.out.println("welcome");
}
}
catch(Exception e)
{       }
}
}
class ThreadDemo
{
public static void main(String args[])
```

```
{
A t1=new A();
B t2=new B();
C t3=new C();
t1.start();
t2.start();
t3.start();
}
}
```

Output:

```
Press Cntrl+C to exit
Good morning
Hello
Good morning
Welcome
Good morning
Hello
Good morning
Good morning
Hello
```

InterThread Communication:

Thread Communication: In some cases two or more threads should communicate with each other. One thread output may be send as input to other thread. For example, a consumer thread is waiting for a Producer to produce the data (or some goods). When the Producer thread completes production of data, then the Consumer thread should take that data and use it.

In producer class we take a StringBuffer object to store data, in this case; we take some numbers from 1 to 5. These numbers are added to StringBuffer object. Until producer completes placing the data into StringBuffer the consumer has to wait. Producer sends a notification immediately after the data production is over.

/** Write a Java program that correctly implements producer consumer problem using the concept of interthread communication.*/

```
//java program for producer and consumer--inter thread communication

class Producer implements Runnable

{

        Thread1 t;

        Producer(Thread1 t)

        {

                this.t=t;

                new Thread(this,"Producer").start();

        }

        public void run()

        {

                int i=0;

                while (true)

                {

                        t.put(i++);

                }

        }

}

class Consumer implements Runnable

{

   Thread1 t;

        Consumer(Thread1 t)

        {

                this.t=t;

                new Thread(this,"Consumer").start();

        }
```

```
	public void run()
	{
		int i=0;
		while (true)
		{
			t.get();
		}
	}
}
class  ProducerConsumer
{
	public static void main(String[] args)
	{
		Thread1 t=new Thread1();
		System.out.println("Press Control+c to exit");
		new Producer(t);
		new Consumer(t);


	}
}
```

Output:

```
Press Control+C to exit
Put:0
Get:0
Put:1
Get1
…………….
```

Deamon Threads:

Daemon threads are sometimes called "service" threads that normally run at a low priority and provide a basic service to a program or programs when activity on a machine is reduced. An example of a daemon thread that is continuously running is the garbage collector thread. This thread, provided by the JVM, will scan programs for variables that will never be accessed again and free up their resources back to the system. A thread can set the daemon flag by passing a true boolean value to the setDaemon() method. If a false boolean value is passed, the thread will become a user thread. However, this must occur before the thread has been started.

A deamon thread is a thread that executes continuously. Daemon threads are service providers for other threads or objects. It generally provides a background processing.

- To make a thread t as a deamon thread, we can use setDeamon() method as:

  t.setDeamon(true);

- To know if a thread is deamon or not, isDeamon is useful.

  boolean x=t.isDaemon().

Write a example program for setting a thread as a daemon thread

```
public class DaemonDemo extends Thread

{

    public void run()

    {

        for(int i=0;i<5;i++)

        System.out.println(this.getName()+" :"+i);

    }

    public static void main(String args[])

    {

        DaemonDemo d1=new DaemonDemo();

        DaemonDemo d2=new DaemonDemo();

        d1.setName("Daemon thread");
```

```
        d2.setName("Normal thread");

        d1.setDaemon(true);

        d1.setPriority(Thread.MIN_PRIORITY);

        d1.start();

        d2.start();

    }

}
```

Output:

Daemon thread:0

Normal thread:0

………….

Daemon thread:4

Normal thread:4

Thread Groups:

Thread Group: A ThreadGroup represents a group of threads. The main advantage of taking several threads as a group is that by using a single method, we will be able to control all the threads in the group.

Thread groups offer a convenient way to manage groups of threads as a unit. This is particularly valuable in situations in which you want to suspend and resume a number of related threads. For example, imagine a program in which one set of threads is used for printing a document, another set is used to display the document on the screen, and another set saves the document to a disk file. If printing is aborted, you will want an easy way to stop all threads related to printing. Thread groups offer this convenience.

- Creating a thread group: *ThreadGroup tg = new ThreadGroup (“groupname”);*
- To add a thread to this group (tg):*Thread t1 = new Thread (tg, targetobj, “threadname”);*
- To add another thread group to this group (tg):

  ThreadGroup tg1 = new ThreadGroup (tg, “groupname”);

- To know the parent of a thread: *tg.getParent ();*
- To know the parent thread group: *t.getThreadGroup ();*

This returns a ThreadGroup object to which the thread t belongs.

- To know the number of threads actively running in a thread group: *t.activeCount ();*
- To change the maximum priority of a thread group tg: *tg.setMaxPriority ();*

Method	Description
int activeCount( )	Returns the number of threads in the group plus any groups for which this thread is a parent.
int activeGroupCount( )	Returns the number of groups for which the invoking thread is a parent.
final void checkAccess( )	Causes the security manager to verify that the invoking thread may access and/or change the group on which **checkAccess( )** is called.
final void destroy( )	Destroys the thread group (and any child groups) on which it is called.
int enumerate(Thread *group*[ ])	The threads that comprise the invoking thread group are put into the *group* array.
int enumerate(Thread *group*[ ], boolean *all*)	The threads that comprise the invoking thread group are put into the *group* array. If *all* is **true**, then threads in all subgroups of the thread are also put into *group*.
int enumerate(ThreadGroup *group*[ ])	The subgroups of the invoking thread group are put into the *group* array.
int enumerate(ThreadGroup *group*[ ], boolean *all*)	The subgroups of the invoking thread group are put into the *group* array. If *all* is **true**, then all subgroups of the subgroups (and so on) are also put into *group*.
final int getMaxPriority( )	Returns the maximum priority setting for the group.
final String getName( )	Returns the name of the group.
final ThreadGroup getParent( )	Returns **null** if the invoking **ThreadGroup** object has no parent. Otherwise, it returns the parent of the invoking object.
final void interrupt( )	Invokes the **interrupt( )** method of all threads in the group.

TABLE 16-18 The Methods Defined by **ThreadGroup**

Program : Write a program to demonstrate the creation of thread group.

```
//Using ThreadGroup

import java.io.*;

class WhyTGroups

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

  {  Reservation res = new Reservation ();

    Cancellation can = new Cancellation ();

    //Create a ThreadGroup

    ThreadGroup tg = new ThreadGroup ("Reservation Group");
```

```
    //Create 2 threads and add them to thread group
    Thread t1 = new Thread (tg, res, "First Thread");
    Thread t2 = new Thread (tg, res, "Second Thread");
    //Create another thread group as a child to tg
    ThreadGroup tg1 = new ThreadGroup (tg, "Cancellation Group");
    Thread t3 = new Thread (tg1, can, "Third Thread");
    Thread t4 = new Thread (tg1, can, "Fourth Thread");
    //find parent group of tg1
    System.out.println ("Parent of tg1 = " + tg1.getParent ());
    //set maximum priority
    tg1.setMaxPriority (7);
    System.out.println ("Thread group of t1 = " + t1.getThreadGroup ());
    System.out.println ("Thread group of t3 = " + t3.getThreadGroup ());
    t1.start ();
    t2.start ();
    t3.start ();
    t4.start ();
    System.out.println ("Number of threads in this group : " + tg.activeCount () );
  }
}
class Reservation extends Thread
{  public void run ()
  {  System.out.println ("I am Reservation Thread");
  }
}
class Cancellation extends Thread
{  public void run ()
```

```
  {  System.out.println ("I am Cancellation Thread");
  }
}
```

Output:

```
D:\JQR>javac WhyTGroups.java

D:\JQR>java  WhyTGroups
Parent of tg1 = java.lang.ThreadGroup[name=Reservation Group,maxpri=10]
Thread group of t1 = java.lang.ThreadGroup[name=Reservation Group,maxpri=10]
Thread group of t3 = java.lang.ThreadGroup[name=Cancellation Group,maxpri=7]
I am Reservation Thread
Number of threads in this group : 4
I am Cancellation Thread
I am Cancellation Thread
I am Reservation Thread

D:\JQR>_
```