# UNIT V

**Exception handling** - Concepts of exception handling, benefits of exception handling, Termination or resumptive models, exception hierarchy, usage of try, catch, throw, throws and finally, built in exceptions, creating own exception sub classes.
String handling, Exploring java.util.

## 5.1 CONCEPTS OF EXCEPTION HANDLING

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.

· **Runtime errors:** These are the errors that represent computer inefficiency. Insufficient memory to store data or inability of the microprocessor to execute some statement is examples to runtime errors. Runtime errors are detected by JVM at runtime.

· **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.

**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**.
  **e.g.:** ClassNotFoundException, NoSuchMethodException, NoSuchFieldException etc.
- Exceptions which occur at run time are called **Unchecked exceptions**.
  **eg:**ArrayIndexOutOfBoundsException, ArithmeticException, etc..

## Error Vs Exception

Error along with RuntimeException & their subclasses are unchecked exceptions. All other Exception classes are checked exceptions.

Checked exceptions are generally those from which a program can recover & it might be a good idea to recover from such exceptions programmatically. Examples include FileNotFoundException, ParseException, etc. A programmer is expected to check for these exceptions by using the try-catch block or throw it back to the caller.

On the other hand we have unchecked exceptions. These are those exceptions that might not happen if everything is in order, but they do occur. Examples include ArrayIndexOutOfBoundException, ClassCastException, etc. Many applications will use try-catch or throws clause for RuntimeExceptions & their subclasses but from the language perspective it is not required to do so. Do note that recovery from a RuntimeException is generally possible but the guys who designed the class/exception deemed it unnecessary for the end programmer to check for such exceptions.

Errors are also unchecked exception & the programmer is not required to do anything with these. In fact it is a bad idea to use a try-catch clause for Errors. Most often, recovery from an Error is not possible & the program should be allowed to terminate. Examples include OutOfMemoryError, StackOverflowError, etc.

Do note that although Errors are unchecked exceptions, we shouldn't try to deal with them.

Errors are derived from java.lang.Error, and Exceptions are derived from java.lang.Exception.

- ✓ An Error "indicates serious problems that a reasonable application should not try to catch."
- ✓ An Exception "indicates conditions that a reasonable application might want to catch."

## 5.2 BENEFITS OF EXCEPTION HANDLING

Exception handling provides the following advantages over ``traditional" error Management techniques:

**Separating Error Handling Code from ``regular'' one**
It separates the working/functional code from the error-handling code by way of try-catch clauses.

**Propagating Errors Up the Call Stack**
It allows a clean path for error propagation. If the called method encounters a situation it can't manage, it can throw an exception and let the calling method deal with it.

**Error Types and Error Differentiation**
By enlisting the compiler to ensure that "exceptional" situations are anticipated and accounted for, it enforces powerful coding.

## 5.3 TERMINATION VS. RESUMPTION MODELS

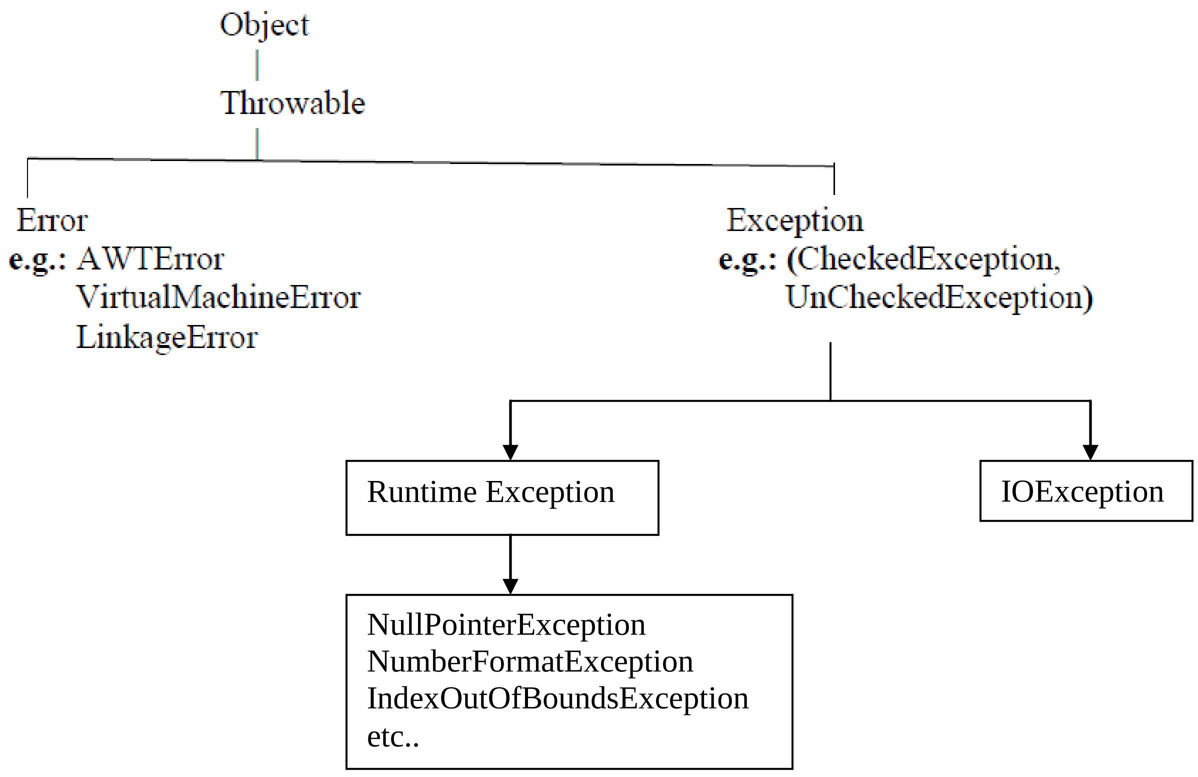
**Two models in exception handling theory:**

- ✓ Unrecoverable error cause termination
- ✓ To resume program execution after recovering form error is resumption

There are two basic models in exception-handling theory. In ***termination*** (which is what Java and C++ support), you assume the error is so critical there's no way to get back to where the exception occurred. Whoever threw the exception decided that there was no way to salvage the situation, and they don't *want* to come back.

The alternative is called ***resumption***. It means that the exception handler is expected to do something to rectify the situation, and then the faulting method is retried, presuming success the second time. If you want resumption, it means you still hope to continue execution after the exception is handled. In this case, your exception is more like a method call – which is how you should set up situations in Java in which you want resumption-like behavior. (That is, don't throw an exception; call a method that fixes the problem.) Alternatively, place your **try** block inside a **while** loop that keeps reentering the **try** block until the result is satisfactory.

## 5.4 EXCEPTION HIERARCHY



A feature built into the Java language is that **Error**s and **RuntimeException**s (and their subclasses) are what are called **unchecked exceptions**:

- unchecked exceptions can be thrown "at any time";
- **methods don't explicitly have to declare** that they can throw an unchecked exception;
- callers **don't have to handle them** explicitly.

An exception can be handled by the programmer where as an error cannot be handled by the programmer. When there is an exception the programmer should do the following tasks:

· If the programmer suspects any exception in program statements, he should write them inside try block.

```
try
{
statements;
}
```

· When there is an exception in try block JVM will not terminate the program abnormally. JVM stores exception details in an exception stack and then JVM jumps into catch block.The programmer should display exception details and any message to the user in catch block.

```
catch ( ExceptionClass obj)
{ statements;
}
```

· Programmer should close all the files and databases by writing them inside finally block. Finally block is executed whether there is an exception or not.

```
finally
{ statements;
}
```

Performing above tasks is called Exception Handling.

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed before try block ends
}
```

## 5.5 Using try and catch

To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a **try** block.
Each **try** block, must include a matching **catch** clause

```
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.
```

## 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, 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.

```
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.");
}
}
```

### 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.

```
class NestTry {
public static void main(String args[]) {
try {
int a = args.length;
int b = 42 / a;
System.out.println("a = " + a);
try { // nested try block
if(a==1) a = a/(a-a); // division by zero

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);
}
}
}
```

## throw

throw clause can be used to throw out user defined exceptions. It is useful to create an exception object and throw it out of try block

```
class ThrowDemo
{
static void Demo( )
{
 try
{
 System.out.println ("inside method");
throw new NullPointerException("my data");
}
catch (NullPointerException ne)
{
System.out.println ("ne");
}
}
public static void main(String args[])
{
ThrowDemo.Demo ( );
}
}
```

## 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 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.

**throws clause is useful to escape from handling an exception. throws clause is useful to throw out any exception without handling it.**

```
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[])
throws IOException
{
Sample s=new Sample ( );
s.accept ( );
}
}
```

## finally

- ✓ **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.
- ✓ The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause.

```
class FinallyDemo
{
static void procA()
{
try
{
System.out.println("inside procA");
throw new RuntimeException("demo");
}
finally
{
System.out.println("procA's finally");
}
}
.
static void procB()
{
try
{
System.out.println("inside procB");
return;
}
finally
{
System.out.println("procB's finally");
}
}

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();
}
}
```

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
```

## 5.6 Java’s Built-in Exceptions

Inside the standard package **java.lang**, Java defines several exception classes. The most general of these exceptions are subclasses of the standard type **RuntimeException**. Since **java.lang** is implicitly imported into all Java programs, most exceptions derived from **RuntimeException** are automatically available.

| **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. |
| 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. |

## 5.7 Creating Your Own Exception Subclasses

These are the exceptions created by the programmer.

**Creating user defined exceptions:**

- Write user exception class extending Exception class.

**e.g.:** class MyException extends Exception

- Write a default constructor in the user exception class

**e.g.:** MyException ( ) { }

- Write a parameterized constructor with String as a parameter, from there call the parameterized constructor of Exception class.

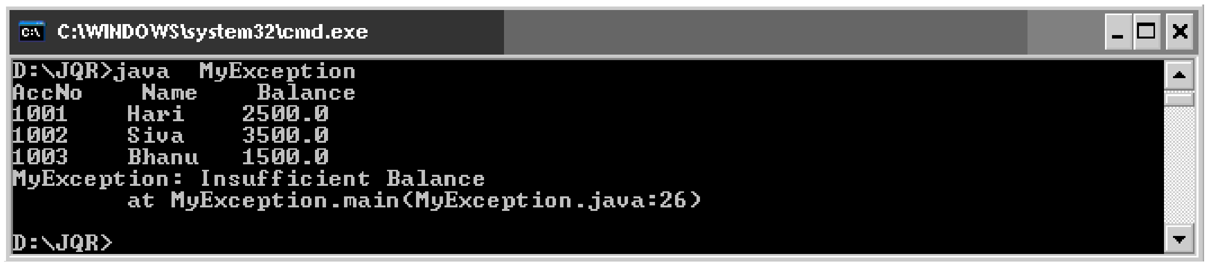
**e.g.:** MyException (String str)

```
{
super (str);
}
```

- Whenever required create user exception object and throw it using throw statement.

Ex: - throw me;

```
class MyException extends Exception
{
int accno[] = {1001,1002,1003,1004,1005};
String name[] = {"Hari","Siva","Bhanu","Rama","Chandu"};
double bal[] = {2500,3500,1500,1000,6000};
MyException()
{
}
MyException(String str)
{
super(str);
}
public static void main(String args[])
{
try
{
MyException me = new MyException("");
System.out.println("AccNo \t Name \t Balance ");
for(int i=0;i<5;i++)
{
System.out.println(me.accno[i]+ "\t" + me.name[i] + "\t" +
me.bal[i] );
if( me.bal[i] < 2000 )
{
MyException me1 = new MyException
("Insufficient Balance");
throw me1;
}
}
}
catch(MyException e)
{
e.printStackTrace();
}
}
}
```


C:\WINDOWS\system32\cmd.exe

```
D:\JQR>java  MyException
AccNo     Name     Balance
1001     Hari     2500.0
1002     Siva     3500.0
1003     Bhanu    1500.0
MyException: Insufficient Balance
        at MyException.main(MyException.java:26)

D:\JQR>
```

## 5.8 STRING HANDLING

In Java a *string* is a sequence of characters. But, unlike many other languages that implement strings as character arrays, Java implements strings as objects of type **String**.

### The String Constructors

The **String** class supports several constructors. To create an empty **String**, you call the default constructor. For example,
String s = new String();
will create an instance of **String** with no characters in it.

### String Length

The length of a string is the number of characters that it contains. To obtain this value, call the **length( )** method, shown here:
int length( )
The following fragment prints “3”, since there are three characters in the string **s**:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

## String Concatenation

In general, Java does not allow operators to be applied to **String** objects. The one exception to this rule is the **+** operator, which concatenates two strings, producing a **String** object as the result. This allows you to chain together a series of **+** operations.

For example, the following fragment concatenates three strings:

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);
```

This displays the string “He is 9 years old.”

## Character Extraction

The **String** class provides a number of ways in which characters can be extracted from a **String** object. Each is examined here. Although the characters that comprise a string within a **String** object cannot be indexed as if they were a character array, many of the **String** methods employ an index (or offset) into the string for their operation. Like arrays, the string indexes begin at zero.

### charAt( )

To extract a single character from a **String**, you can refer directly to an individual character via the **charAt( )** method. It has this general form:

char charAt(int *where*)

Here, *where* is the index of the character that you want to obtain.

### getChars( )

If you need to extract more than one character at a time, you can use the **getChars( )** method. It has this general form:

void getChars(int *sourceStart*, int *sourceEnd*, char *target*[ ], int *targetStart*)

### getBytes( )

There is an alternative to **getChars( )** that stores the characters in an array of bytes. This method is called **getBytes( )**, and it uses the default character-to-byte conversions provided by the platform. Here is its simplest form:

byte[ ] getBytes( )

**toCharArray( )**

If you want to convert all the characters in a **String** object into a character array, the easiest way is to call **toCharArray( )**. It returns an array of characters for the entire string. It has this general form:

char[ ] toCharArray( )

**String Comparison**

The **String** class includes several methods that compare strings or substrings within strings. Each is examined here.

**equals( ) and equalsIgnoreCase( )**

To compare two strings for equality, use **equals( )**. It has this general form:

boolean equals(Object *str*)

**startsWith( ) and endsWith( )**

**String** defines two routines that are, more or less, specialized forms of **regionMatches( )**. The **startsWith( )** method determines whether a given **String** begins with a specified string. Conversely, **endsWith( )** determines whether the **String** in question ends with a specified string. They have the following general forms:

boolean startsWith(String *str*)
boolean endsWith(String *str*)

**equals( ) Versus ==**

It is important to understand that the **equals( )** method and the **==** operator perform two different operations. As just explained, the **equals( )** method compares the characters inside a **String** object. The **==** operator compares two object references to see whether they refer to the same instance. The following program shows how two different **String** objects can contain the same characters, but references to these objects will not compare as equal:

```
// equals() vs ==
class EqualsNotEqualTo {
public static void main(String args[]) {
String s1 = "Hello";
String s2 = new String(s1);
System.out.println(s1 + " equals " + s2 + " -> " +
s1.equals(s2));
System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
}
}
```

The variable **s1** refers to the **String** instance created by “**Hello**”. The object referred to by **s2** is created with **s1** as an initializer. Thus, the contents of the two **String** objects are identical, but they are distinct objects. This means that **s1** and **s2** do not refer to the same objects and are, therefore, not **==**, as is shown here by the output of the preceding example:

```
Hello equals Hello -> true
Hello == Hello -> false
```

## compareTo( )

Often, it is not enough to simply know whether two strings are identical. For sorting applications, you need to know which is *less than, equal to,* or *greater than* the next. A string is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order. The **String** method **compareTo( )** serves this purpose. It has this general form:

int compareTo(String *str*)

Here, *str* is the **String** being compared with the invoking **String**. The result of the comparison is returned and is interpreted as shown here:

| Value | Meaning |
|---|---|
| Less than zero | The invoking string is less than *str*. |
| Greater than zero | The invoking string is greater than *str*. |
| Zero | The two strings are equal. |

## Searching Strings

The **String** class provides two methods that allow you to search a string for a specified character or substring:
■ **indexOf( )** Searches for the first occurrence of a character or substring.
■ **lastIndexOf( )** Searches for the last occurrence of a character or substring.

## substring( )

You can extract a substring using **substring( )**. It has two forms. The first is
String substring(int *startIndex*)

## concat( )

You can concatenate two strings using **concat( )**, shown here:
String concat(String *str*)

## replace( )

The **replace( )** method replaces all occurrences of one character in the invoking string with another character. It has the following general form:

String replace(char *original*, char *replacement*)
Here, *original* specifies the character to be replaced by the character specified by *replacement.* The resulting string is returned. For example,
String s = "Hello".replace('l', 'w');
puts the string “Hewwo” into **s**.

## trim( )

The **trim( )** method returns a copy of the invoking string from which any leading and trailing whitespace has been removed. It has this general form:
String trim( )
Here is an example:
String s = " Hello World ".trim();

## Changing the Case of Characters Within a String

The method **toLowerCase( )** converts all the characters in a string from uppercase to lowercase. The **toUpperCase( )** method converts all the characters in a string from lowercase to uppercase. Nonalphabetical characters, such as digits, are unaffected.
Here are the general forms of these methods:

String toLowerCase( )
String toUpperCase( )

## STRINGBUFFER

**StringBuffer** is a peer class of **String** that provides much of the functionality of strings. As you know, **String** represents fixed-length, immutable character sequences. In contrast, **StringBuffer** represents growable and writeable character sequences. **StringBuffer** may have characters and substrings inserted in the middle or appended to the end. **StringBuffer** will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth. Java uses both classes heavily, but many programmers deal only with **String** and let Java manipulate **StringBuffer**s behind the scenes by using the overloaded **+** operator.

**StringBuffer Constructors**
**StringBuffer** defines these three constructors:

StringBuffer( )
StringBuffer(int *size*)
StringBuffer(String *str*)

**<u>length( ) and capacity( )</u>**

The current length of a **StringBuffer** can be found via the **length( )** method, while the total allocated capacity can be found through the **capacity( )** method. They have the following general forms:
int length( )
int capacity( )

**<u>ensureCapacity( )</u>**

If you want to preallocate room for a certain number of characters after a **StringBuffer** has been constructed, you can use **ensureCapacity( )** to set the size of the buffer. This is useful if you know in advance that you will be appending a large number of small strings to a **StringBuffer**. **ensureCapacity( )** has this general form:

void ensureCapacity(int *capacity*)
Here, *capacity* specifies the size of the buffer.

**<u>setLength( )</u>**

To set the length of the buffer within a **StringBuffer** object, use **setLength( )**. Its general form is shown here:
void setLength(int *len*)

Here, *len* specifies the length of the buffer. This value must be nonnegative. When you increase the size of the buffer, null characters are added to the end of the existing buffer. If you call **setLength( )** with a value less than the current value returned by **length( )**, then the characters stored beyond the new length will be lost.

**<u>charAt( ) and setCharAt( )</u>**

The value of a single character can be obtained from a **StringBuffer** via the **charAt( )** method. You can set the value of a character within a **StringBuffer** using **setCharAt( )**. Their general forms are shown here:

char charAt(int *where*)
void setCharAt(int *where*, char *ch*)

**getChars( )**

To copy a substring of a **StringBuffer** into an array, use the **getChars( )** method. It has this general form:

void getChars(int *sourceStart*, int *sourceEnd*, char *target*[ ], int *targetStart*)

**append( )**

The **append( )** method concatenates the string representation of any other type of data to the end of the invoking **StringBuffer** object. It has overloaded versions for all the built-in types and for **Object**. Here are a few of its forms:

StringBuffer append(String *str*)
StringBuffer append(int *num*)
StringBuffer append(Object *obj*)

**insert( )**

The **insert( )** method inserts one string into another. It is overloaded to accept values of all the simple types, plus **String**s and **Object**s. Like **append( )**, it calls **String.valueOf( )** to obtain the string representation of the value it is called with. This string is then inserted into the invoking **StringBuffer** object. These are a few of its forms:

StringBuffer insert(int *index*, String *str*)
StringBuffer insert(int *index*, char *ch*)
StringBuffer insert(int *index*, Object *obj*)

**reverse( )**

You can reverse the characters within a **StringBuffer** object using **reverse( )**, shown here:

StringBuffer reverse( )

**delete( ) and deleteCharAt( )**

Java 2 added to **StringBuffer** the ability to delete characters using the methods **delete( )** and **deleteCharAt( )**. These methods are shown here:

StringBuffer delete(int *startIndex*, int *endIndex*)
StringBuffer deleteCharAt(int *loc*)

### replace( )

Another method added to **StringBuffer** by Java 2 is **replace( )**. It replaces one set of characters with another set inside a **StringBuffer** object. Its signature is shown here:

StringBuffer replace(int *startIndex*, int *endIndex*, String *str*)

### substring( )

Java 2 also added the **substring( )** method, which returns a portion of a **StringBuffer**. It has the following two forms:

String substring(int *startIndex*)
String substring(int *startIndex*, int *endIndex*)

## Exploring java.util

The **java.util** package contains one of Java's most powerful subsystems: collections.

### Collection Framework

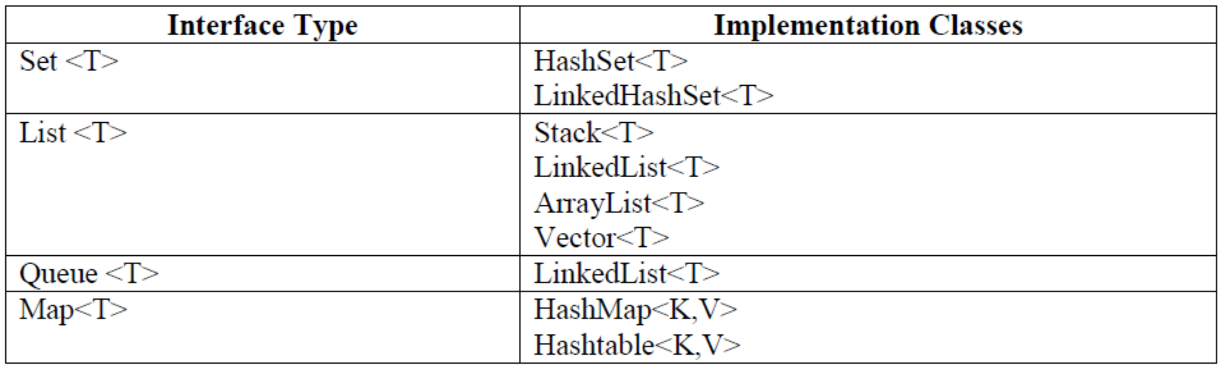
In order to handle group of objects we can use array of objects. If we have a class called Employ with members name and id, if we want to store details of 10 Employees, create an array of object to hold 10 Employ details.

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 retrieving 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.
· All the collection classes in java.util package are the implementation classes of different interfaces.



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

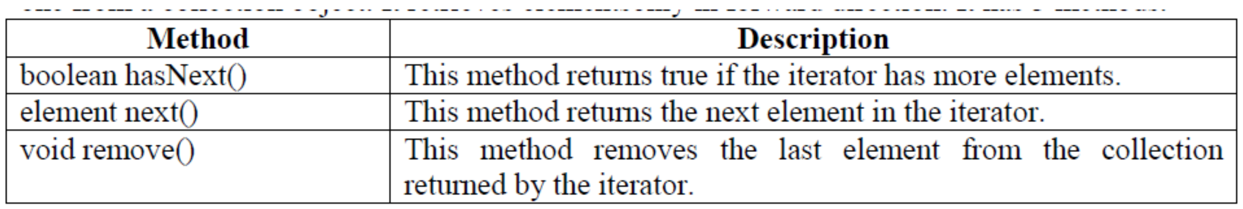
**Set:** A Set represents a group of elements (objects) arranged just like an array. The set will grow dynamically when the elements are stored into it. A set will not allow duplicate elements.
· **List:** Lists are like sets but allow duplicate values to be stored.
· **Queue:** A Queue represents arrangement 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 elements only 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. |

```
import java.util.*;
class HS
{
```

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

**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. |

Write a program to perform different operations on a stack.

```
//pushing, popping, searching elements in a stack
import java.util.*;
class StackDemo
{
public static void main(String args[])
{ //create an empty stack to contain Integer objects
Stack<Integer> st = new Stack<Integer>();
st.push (new Integer(10) );
st.push (new Integer(20) );
st.push (new Integer(30) );
st.push (new Integer(40) );
st.push (new Integer(50) );
System.out.println (st);
System.out.println ("Element at top of the stack is : " + st.peek() );
System.out.println ("Removing element at the TOP of the stack : " + st.pop());
System.out.println ("The new stack is : " + st);
}
}
```

**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. |

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() );
}}
```

**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> ();

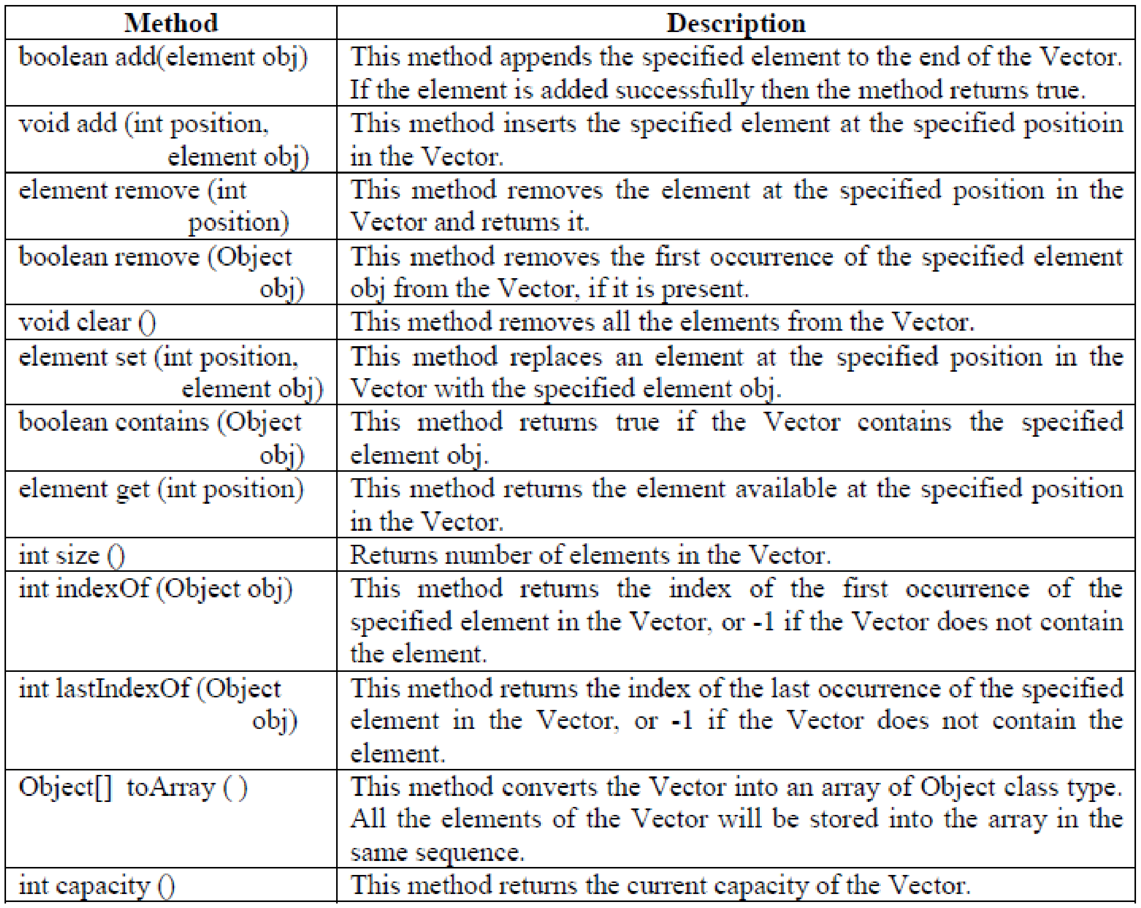
Write a program that shows the use of ArrayList class.

```
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");
} }
```

**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. |

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");
}
}
```

**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. Keys 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 keyvalue pair into the HashMap, its capacity will become 32.

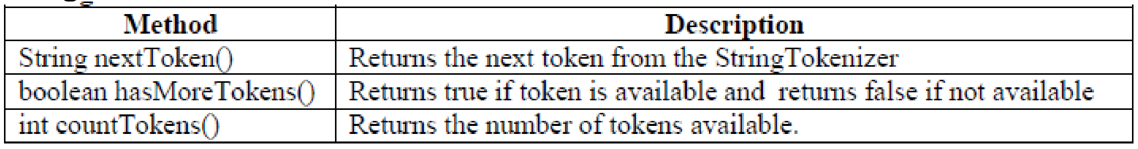
Write a program that shows the use of HashMap class.

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

**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");

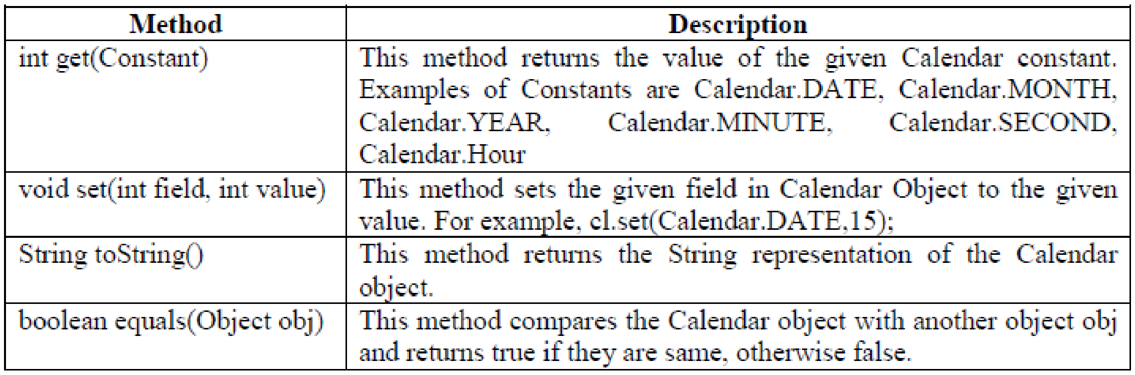
**StringTokenizer Class Methods:**



| 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. |

**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. |

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);
}
}
```

**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);

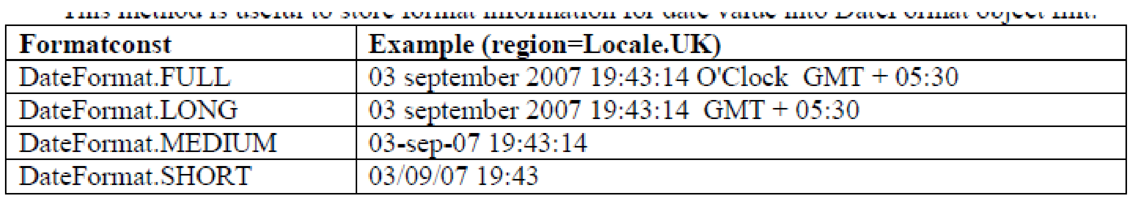
This method is useful to store format information for date value into DateFormat object fmt.

· 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.



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 |

```
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);
}
}
```