JGI JAIN DEEMED-TO-BE UNIVERSITY | SCHOOL OF COMPUTER SCIENCE AND IT

Master of Computer Applications (MCA)

OOP's Using JAVA

20MCAC101

Unit – 2 : Thread Programming, Exceptions and Collections

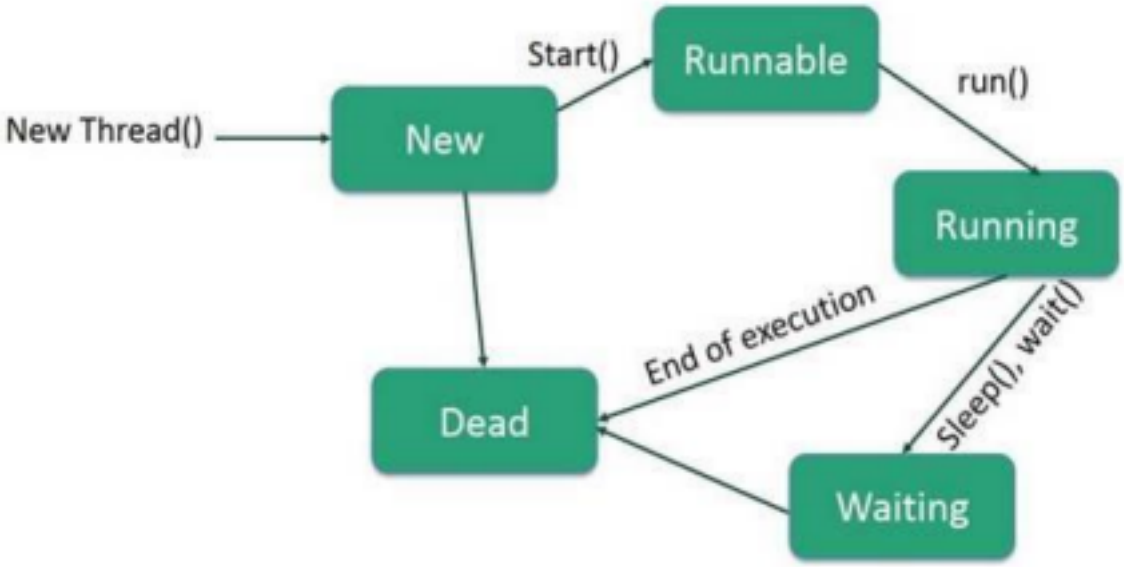
Prepared by
Raghavendra R
Assistant professor
School of CS & IT

r.raghavendra@jainuniversity.ac.in +91-86601-61661 raguramesh88@gmail.com
+91-96116-77907

MULTITHREADING

Multi-threading is a special type of multi-tasking in which a single program is divided into several pieces called thread and each thread is permitted to proceed simultaneously. Multi-threading enables you to write very efficient programs that makes maximum use of CPU because idle time can be kept to minimum.

Thread life cycle



• **New state:** After the creations of Thread instance the thread is in this state but before the start() method invocation. At this point, the thread is considered not alive.

• **Runnable (Ready-to-run) state:** A thread starts its life from Runnable state. A thread first enters runnable state after the invoking of start() method but a thread can return to this state after either running, waiting, sleeping or coming back from blocked state also. On this state a thread is waiting for a turn on the processor.

• **Running state:** A thread is in running state that means the thread is currently executing. There are several ways to enter in Runnable state but there is only one way to enter in Running state: the scheduler select a thread from runnable pool.

• **Dead state:** A thread can be considered dead when its run() method completes. If any thread comes on this state that means it cannot ever run again.

• **Blocked:** A thread can enter in this state because of waiting the resources that are hold by another thread.

Program:

```
class Thread1 implements Runnable
{
public void run()
```



```
{
for(int i=0; i<5; i++)
{ System.out.println(getName()+ "i=" +i);
}
Thread1()
```

```
{ start();
}
class threaddemo
{

public static void main(String ar[])

{
Thread1 t1= new Thread1();
Thraed1 t2= new Thraed1();
}
}
```

Thread Priorities

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another between 1 to 10.

As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running. Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a *context switch.* The rules that determine when a context switch takes place are simple:

• *A thread can voluntarily relinquish control.* This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.

• *A thread can be preempted by a higher-priority thread.* In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing—by a higher priority thread.

Basically, as soon as a higher-priority thread wants to run, it does. This is called *preemptive multitasking.* In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated. For operating systems such as Windows 98, threads of equal priority are time-sliced automatically in round-robin fashion. For other types of operating systems, such as Solaris 2.*x,* threads of equal priority must voluntarily yield control to their peers. If they don't, the other threads will not run.

Caution : Problems can arise from the differences in the way that operating systems context switch threads of equal priority.

Synchronization

Because multithreading introduces an asynchronous behavior to your programs, there must be a way for you to enforce synchronicity when you need it. For example, if you want two threads to communicate and share a complicated data structure, such as a linked list, you need some way to

ensure that they don't conflict with each other. That is, you must prevent one thread from writing data while another thread is in the middle of reading it. For this purpose, Java implements an elegant twist on an age-old model of interprocess synchronization: the *monitor*. The monitor is a control mechanism first defined by C.A.R.
Hoare. You can think of a monitor as a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time. Most multithreaded systems expose monitors as objects that your program must explicitly acquire and manipulate. Java provides a cleaner solution. There is no class "Monitor"; instead, each object has its own implicit monitor that is automatically entered when one of the object's **synchronized** methods is called.

Once a thread is inside a method, no other thread can call any other synchronized method on the same object. This enables you to write very clear and concise multithreaded code, because synchronization support is built in to the language.
There are two ways that you can synchronize your code. Both involve the use of
the **synchronized** keyword,

Using Synchronized Methods
The synchronized block

For thread coordination wait (), notify () and notifyAll () are used:

Allows two threads to cooperate
Based on a single shared lock object

When a thread calls wait(), the following occurs:

the thread releases the object lock.
thread state is set to blocked.
thread is placed in the wait set.

When a thread calls notify(), the following occurs:

selects an arbitrary thread *T* from the wait set.
moves *T* to the entry set.
sets *T* to Runnable.
T can now compete for the object's lock again.

Use notifyAll() if there may be more than one waiting thread. Notify () selects an arbitrary thread from the wait set. This may not be the thread that you want to be selected. Java does not allow you to specify the thread to be selected. notifyAll () removes ALL threads from the wait set and places them in the entry set. This allows the threads to decide among themselves who should

proceed next. notifyAll() is a conservative strategy that works best when multiple threads may be in the wait set.
class BlockingQueue extends Queue

```
{
public synchronized Object remove()
{
while (isEmpty())
{
wait(); // really this.wait()
}
return super.remove();
}
public synchronized void add(Object o)
{
super.add(o);
notifyAll(); // this.notifyAll()
}

}
```

Exception Handling

An exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error. In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes. This approach is as cumbersome as it is troublesome. Java's exception handling avoids these problems and, in the process, brings run-time error management into the object oriented world.

Exception Handling Fundamentals

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed.

Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.

Java exception handling is managed using five keywords:

- try: A suspected code segment is kept inside try block.

• catch: The remedy is written within catch block.

• throw: Whenever run-time error occurs, the code must throw an exception. • throws: If a method cannot handle any exception by its own and some subsequent methods needs to handle them, then a method can be specified with throws keyword with its declaration.

• finally: block should contain the code to be executed after finishing try-block.

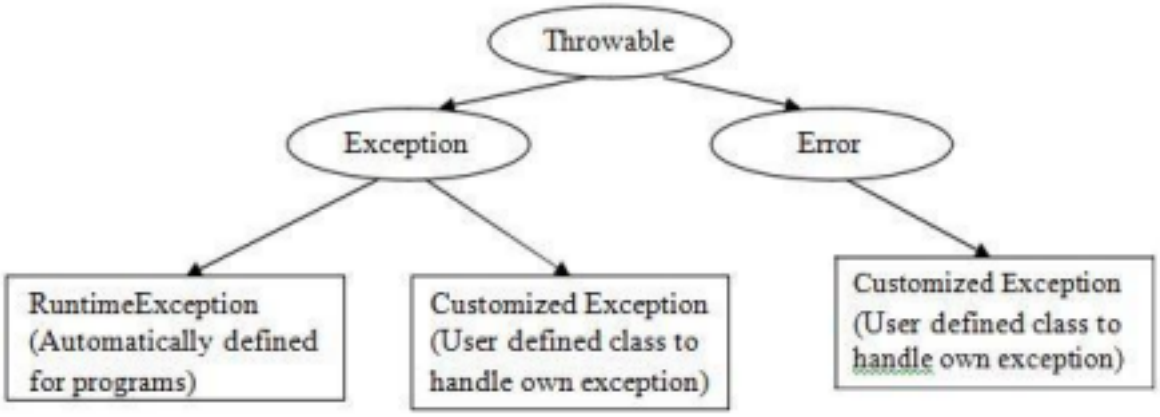
The general form of exception handling is –

```
try
 {
// block of code to monitor errors
 }
catch (ExceptionType1 exOb)
 {
// exception handler for ExceptionType1
 }
catch (ExceptionType2 exOb)
 {

// exception handler for ExceptionType2
 }
...
 ….
finally
 {
// block of code to be executed after try block ends
 }
```

Exception Types

All the exceptions are the derived classes of built-in class viz. Throwable. It has two subclasses viz. Exception and Error.



Exception class is used for exceptional conditions that user programs should catch. We can inherit from this class to create our own custom exception types. There is an important subclass of Exception, called RuntimeException. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

Error class defines exceptions that are not expected to be caught under normal circumstances by our program. Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

Uncaught Exceptions

Let us see, what happens if we do not handle exceptions.

```
class Exc0
{
public static void main(String args[])
{
int d = 0;
int a = 42 / d;
}
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception. This causes the execution of Exc0 to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately. Since, in the above program, we have not supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system.

Any un-caught exception is handled by default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program. Here is the exception generated when above example is executed:

java.lang.ArithmeticException: / by zero at Exc0.main(Exc0.java:6)

The stack trace displays class name, method name, file name and line number causing the exception. Also, the type of exception thrown viz. ArithmeticException which is the subclass of Exception is displayed. The type of exception gives more information about what type of error has occurred. The stack trace will always show the sequence of method invocations that led up to the error.

```
class Exc1
{
static void subroutine()
{
int d = 0;
int a = 10 / d;
}
public static void main(String args[])
{
Exc1.subroutine();
}
}
```

The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

java.lang.ArithmeticException: / by zero

at Exc1.subroutine(Exc1.java:6)

at Exc1.main(Exc1.java:10)

Using try and catch

Handling the exception by our own is very much essential as. We can display appropriate error message instead of allowing Java run-time to display stack-trace. It prevents the program from automatic (or abnormal) termination. To handle run-time error, we need to enclose the suspected code within try block.

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

Output:

```
Division by zero.
After catch 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.

```
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<10; i++)
{
try
{
b = r.nextInt();
c = r.nextInt();
a = 12345 / (b/c);
```

```
} catch (ArithmeticException e)
{
System.out.println("Division by zero."); a = 0;
}
System.out.println("a: " + a);
}
}
}
```

The output of above program is not predictable exactly, as we are generating random numbers. But, the loop will execute 10 times. In each iteration, two random numbers (b and c) will be generated. When their division results in zero, then exception will be caught. Even after exception, loop will continue to execute.

Displaying a Description of an Exception: We can display this description in a println() statement by simply passing the exception as an argument. This is possible because Throwable overrides the toString() method (defined by Object) so that it returns a string containing a description of the exception.

```
catch (ArithmeticException e)
{
System.out.println("Exception: " + e);
a = 0;
}
```

Now, whenever exception occurs, the output will be –

Exception: java.lang.ArithmeticException: / by zero

Multiple Catch Claues

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.

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

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.

While using multiple catch blocks, we should give the exception types in a hierarchy of subclass to superclass. Because, catch statement that uses a superclass will catch all exceptions of its own type plus all that of its subclasses. Hence, the subclass exception given after superclass exception is never caught and is a unreachable code, that is an error in Java.

```
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.");
}

catch(ArithmeticException e) // ERROR - unreachable
{
System.out.println("This is never reached.");
}

}
}
```

The above program generates error "Unreachable Code", because ArithmeticException is a subclass of Exception.

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 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
{
if(a==1)
a = a/(a-a);
```



```
if(a==2)
{
int c[] = { 1 };
c[10] = 99;
}
}catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Array index out-of-bounds: " + e);
}
}catch(ArithmeticException e)
{
System.out.println("Divide by 0: " + e);
}
}
}
```

When a method is enclosed within a try block, and a method itself contains a try block, it is considered to be a nested try block.

```
class MethNestTry
{static void nesttry(int a)
{try
{
if(a==1)
a = a/(a-a); if(a==2)
{
int c[] = { 1 }; c[42] = 99;
}
}catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Array index out-of-bounds: " + e);
}
}
```

```
public static void main(String args[])
 {
try
 {
int a = args.length;
int b = 42 / a;
System.out.println("a = " + a);
```



```
nesttry(a);
} catch(ArithmeticException e)
 {
System.out.println("Divide by 0: " + e);
 }
 }
 }
```

throw

Till now, we have seen catching the exceptions that are thrown by the Java run-time system. 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.

```
class ThrowDemo
 {
static void demoproc()
 {
try
 {
throw new NullPointerException("demo");
```

```
} catch(NullPointerException e)
{
System.out.println("Caught inside demoproc: " + e);
}
}
public static void main(String args[])
{
demoproc();
}
```



```
}
```

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.

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.

```
class ThrowsDemo
{
```

```
static void throwOne() throws IllegalAccessException {
System.out.println("Inside throwOne."); throw new
IllegalAccessException("demo"); }
public static void main(String args[])
{
try
{
throwOne();
} catch (IllegalAccessException e)
{
```



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

finally

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Sometimes 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 such situations.

The finally clause creates a block of code that will be executed after a try/catch block has completed and before the next code of 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. 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();
}
}
```

Output:

```
inside procA
procA's finally

Exception caught
inside procB
```



```
procB's finally

inside procC
procC's finally
```

Java Generics

Generics was first introduced in Java5. Now it is one of the most profound feature of java programming language. Generic programming enables the programmer to create classes,interfaces and methods in which type of data is specified as a parameter. It provides a facility to write an algorithm independent of any specific type of data. Generics also provide type safety. Type safety means ensuring that an operation is being performed on the right type of data before executing that operation.

Using Generics, it has become possible to create a single class ,interface or method that automatically works with all types of data(Integer, String, Float etc). It has expanded the ability to reuse the code safely and easily.

Before Generics was introduced, generalized classes,interfaces or methods were created using references of type Object because Object is the super class of all classes in Java, but this way of programming did not ensure type safety.

Syntax for creating an Object of a Generic type

```
Class_name <data type> reference_name = new Class_name<data type>
(); OR
Class_name <data type> reference_name = new Class_name<>();
```

This is also known as Diamond Notation of creating an object of Generic

type. Example of Generic class

```
//<> brackets indicates that the class is of generic type
class Gen <T>
{
T ob; //an object of type T is declared<
Gen(T o) //constructor
{
ob = o;
}
public T getOb()
{
return ob;
}
```



```
}

class Demo
{
public static void main (String[] args)
{
Gen < Integer> iob = new Gen<>(100); //instance of Integer type Gen Class
int x = iob.getOb();
System.out.println(x);
Gen < String> sob = new Gen<>("Hello"); //instance of String type Gen Class
String str = sob.getOb();
System.out.println(str);
}
}
```

```
100
Hello
```

In the above program, we first passed an Integer type parameter to the Generic class. Then, we passed a String type parameter to the same Generic class. Hence, we reused the same class for two different data types. Thus, Generics helps in code reusability with ease.

Generics Work Only with Objects

Generics work only with objects i.e the type argument must be a class type.You cannot use primitive datatypes such as int, char etc. with Generics type. It should always be an object. We can use all the Wrapper Class objects and String class objects as Generic type.

Gen<int> iOb = new Gen<int>(07); //Error, can't use primitive type

Generics Types of different Type Arguments are never same

Reference of one generic type is never compatible with other generic type unless their type argument is same. In the example above we created two objects of class Gen, one of type Integer, and other of type String, hence,

iob = sob; //Absolutely Wrong

An array of Generic type cannot be created

Creation of a generic type array is not allowed in Generic programming. We can make a reference of an array, but we cannot instantiate it.

For example, In the above program, in class Gen,

T a[]; //this is allowed
T a[] = new T[10]; //this is not allowed

Generic Type with more than one parameter

In Generic parameterized types, we can pass more than one data type as parameter. It works the same as with one parameter Generic type.

```
class Gen <T1,T2>
{
 T1 name;
 T2 value;
 Gen(T1 o1,T2 o2)
 {
 name = o1;
 value = o2;
 }
 public T1 getName()
 {
 return name;
 }
 public T2 getValue()
```

```
{
return value;
}
}
class Demo
{
public static void main (String[] args)
{
Gen < String,Integer> obj = new Gen<>("StudyTonight",1);
String s = obj.getName();
System.out.println(s);
Integer i = obj.getValue();
System.out.println(i);
}
}
```


Note: Since there are two parameters in Generic class - T1 and T2, therefore, while creating an instance of this Generic class, we need to mention two data types that needs to be passed as parameter to this class.

Generic Methods

You can also create generic methods that can be called with different types of arguments. Based on the type of arguments passed to generic method, the compiler handles each method.

The syntax for a generic method includes a type-parameter inside angle brackets, and should appear before the method's return type.

<type-parameter> return_type method_name (parameters) {...}

Example of Generic method

```
class Demo
{
static <V, T> void display (V v, T t)
{
System.out.println(v.getClass().getName()+" = " +v);
System.out.println(t.getClass().getName()+" = " +t);
```

```
}
public static void main(String[] args)
{
display(88,"This is string");
}
}
```

```
java lang.Integer = 88
java lang.String = This is string
```

Generic Constructors in Java

It is possible to create a generic constructor even if the class is not

generic. Example of Generic Constructor

```
class Gen
{
private double val;
<T extends Number> Gen(T ob)
{
```



```
val = ob.doubleValue();
}
void show()
{
System.out.println(val);
}
}
class Demo
{
public static void main(String[] args)
{
Gen g = new Gen(100);
Gen g1 = new Gen(121.5f);
g.show();
g1.show();
}
}
```

100.0
121.5

Generic Bounded type Parameter

You can also set restriction on the type that will be allowed to pass to a type-parameter. This is done with the help of extends keyword when specifying the type parameter.

< T extends Number >

Here we have taken Number class, it can be any wrapper class name. This specifies that T can be only be replaced by Number class data itself or any of its subclass.

Generic Method with bounded type Parameters

In this example, we created a display method that allows only number type or its subclass type.

```
class Demo
{
static < T, V extends Number> void display(T t, V v)
{
System.out.println(v.getClass().getName()+" = " +v);
System.out.println(t.getClass().getName()+" = " +t);
}
public static void main(String[] args)
```



```
{
display ("this is string",99);
}
}
```

java.lang.String = This is string
java.lang.Double = 99.O

Type V is bounded to Number type and its subclass only. If display(88,"This is string") is uncommented, it will give an error of type incompatibility, as String is not a subclass of Number class.

Java Collection Framework

Java collection framework represents a hierarchy of set of interfaces and classes that are used to manipulate group of objects.

Collections framework was added to Java 1.2 version. Prior to Java 2, Java provided adhoc classes such as Dictionary, Vector, Stack and Properties to store and manipulate groups of objects.

Java collections framework is contained in java.util package. It provides many important classes and interfaces to collect and organize group of objects.

What is collection

Collection in java can be referred to an object that collects multiple elements into a single unit. It is used to store, fetch and manipulate data. For example, list is used to collect elements and referred by a list object.

Components of Collection Framework

• The collections framework consists of:
  • Collection interfaces such as sets, lists, and maps. These are used to collect different types of objects.
  • Collection classes such as ArrayList, HashSet etc that are implementations of collection interfaces.
  • Concurrent implementation classes that are designed for highly concurrent use. • Algorithms that provides static methods to perform useful functions on collections, such as sorting a list.
• Advantage of Collection Framework

  • It reduces programming effort by providing built-in set of data structures and algorithms.
  • Increases performance by providing high-performance implementations of data structures and algorithms.
  • Provides interoperability between unrelated APIs by establishing a common language to pass collections back and forth.
  • Increase Productivity
  • Reduce operational time
  • versatility to work with current collection as well

Important Interfaces of Collection API

Interface Description

Collection Enables you to work with groups of object; it is at the top of Collection hierarchy

Deque Extends Queue to handle double ended queue.

List Extends Collection to handle sequences list of object.

Queue Extends Collection to handle special kind of list in which element are removed only from the head.

Set Extends Collection to handle sets, which must contain unique element. SortedSet

Extends Set to handle sorted set.

Java Collection Hierarchy

Collection Framework hierarchy is represented by using a diagram. You can get idea how interfaces and classes are linked with each other and in what hierarchy they are situated. Top of the framework, Collection framework is available and rest of interfaces are sub interface of it.

Collection Heirarchy

All these Interfaces give several methods which are defined by collections classes which implement these interfaces.

Commonly Used Methods of Collection Framework

Method Description

• public boolean add(E e) It inserts an element in this collection.

• public boolean addAll(Collection<? extends E> c) It inserts the specified collection elements in the invoking collection.
• public boolean remove(Object element) It deletes an element from the collection. • public boolean removeAll(Collection<?> c) It deletes all the elements of the specified collection from the invoking collection.
• default boolean removeIf(Predicate<? super E> filter) It deletes all the elements of the collection that satisfy the specified predicate.
• public boolean retainAll(Collection<?> c) It deletes all the elements of invoking collection except the specified collection.
• public int size() It returns the total number of elements in the collection. • public void clear()It removes the total number of elements from the collection. • public boolean contains(Object element) It searches for an element.
• public boolean containsAll(Collection<?> c) It is used to search the specified collection in the

collection.

Java Collection Framework ArrayList

This class provides implementation of an array based data structure that is used to store elements in linear order. This class implements List interface and an abstract AbstractList class. It creates a dynamic array that grows based on the elements strength.

Java ArrayList class Declaration

public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, Serializable

- ArrayList supports dynamic array that can grow as needed.
- It can contain Duplicate elements and it also maintains the insertion order. • Manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.
- ArrayLists are not synchronized.
- ArrayList allows random access because it works on the index basis.
-

ArrayList Constructors

ArrayList class has three constructors that can be used to create Arraylist either from empty or elements of other collection.

ArrayList() // It creates an empty ArrayList

ArrayList(Collection C) // It creates an ArrayList that is initialized with elements of the Collection C

ArrayList(int capacity) // It creates an ArrayList that has the specified initial

capacity *Example: Creating an ArrayList*

Lets create an ArrayList to store string elements. Here list is empty because we did not add elements to it.

```
import java.util.*;
class Demo
{
public static void main(String[] args)
{
```

```
// Creating an ArrayList
ArrayList< String> fruits = new ArrayList< String>();
// Displaying Arraylist
System.out.println(fruits);
}
}
```

ArrayList Methods

The below table contains methods of Arraylist. We can use them to manipulate its elements.

- void add(int index, E element) - It inserts the specified element at the specified position in a list.
- boolean add(E e) - It appends the specified element at the end of a list. • boolean addAll(Collection<? extends E> c) - It appends all of the elements in the specified collection to the end of this list.
- boolean addAll(int index, Collection<? extends E> c) - It appends all the elements in the specified collection, starting at the specified position of the list.
- void clear() - It removes all of the elements from this list.
- void ensureCapacity(int requiredCapacity) - It enhances the capacity of an ArrayList instance.
- E get(int index) - It fetches the element from the particular position of the list. • boolean isEmpty() - It returns true if the list is empty, otherwise false. • int lastIndexOf(Object o) - It returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.

- Object[] toArray() - It returns an array containing all of the elements in this list in the correct order.
- <T> T[] toArray(T[] a) - It returns an array containing all of the elements in this list in the correct order.
- Object clone() - It returns a shallow copy of an ArrayList.
- boolean contains(Object o) - It returns true if the list contains the specified element • int indexOf(Object o) - It returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
- E remove(int index) - It removes the element present at the specified position in the list. • boolean remove(Object o) - It removes the first occurrence of the specified element. • boolean removeAll(Collection<?> c) - It removes all the elements from the list. • boolean removeIf(Predicate<? super E> filter) - It removes all the elements from the list that satisfies

the given predicate.

- protected void removeRange(int fromIndex, int toIndex) - It removes all the elements lies within the given range.
- void replaceAll(UnaryOperator<E> operator) - It replaces all the elements from the list with the specified element.
- void trimToSize() - It trims the capacity of this ArrayList instance to be the list's current size.

Add Elements to ArrayList

To add elements into ArrayList, we are using add() method. It adds elements into the list in the insertion order.

```
import java.util.*;
class Demo
{
public static void main(String[] args)
{
// Creating an ArrayList
ArrayList< String> fruits = new ArrayList< String>();
// Adding elements to ArrayList
fruits.add("Mango");
fruits.add("Apple");
fruits.add("Berry");
// Displaying ArrayList
System.out.println(fruits);
}
}
```

[Mango, Apple, Berry]

Removing Elements

To remove elements from the list, we are using remove method that remove the specified elements. We can also pass index value to remove the elements of it.

```
import java.util.*;
class Demo
{
public static void main(String[] args)
{
// Creating an ArrayList
```

```
ArrayList< String> fruits = new ArrayList< String>();
// Adding elements to ArrayList
fruits.add("Mango");
fruits.add("Apple");
fruits.add("Berry");
fruits.add("Orange");
// Displaying ArrayList
System.out.println(fruits);
// Removing Elements
fruits.remove("Apple");
System.out.println("After Deleting Elements: \n"+fruits);
// Removing Second Element
fruits.remove(1);
System.out.println("After Deleting Elements: \n"+fruits);

}
}
```

```
[Mango, Apple, Berry, Orange]
After Deleting Elements:
[Mango, Berry, Orange]
After Deleting Elements:
[Mango, Orange]
```

Traversing Elements of ArrayList

Since ArrayList is a collection then we can use loop to iterate its elements. In this example we are traversing elements. See the below example.

```
import java.util.*;
class Demo
```



```
{
public static void main(String[] args)
{
// Creating an ArrayList
ArrayList< String> fruits = new ArrayList< String>();
// Adding elements to ArrayList
fruits.add("Mango");
fruits.add("Apple");
```

```
fruits.add("Berry");
fruits.add("Orange");
// Traversing ArrayList
for(String element : fruits) {
System.out.println(element);
}
}
}
```

Mango
Apple
Berry
Orange

Get size of ArrayList

Sometimes we want to know number of elements an ArrayList holds. In that case we use size() then returns size of ArrayList which is equal to number of elements present in the list. import java.util.*;

```
class Demo
{
public static void main(String[] args)
{
// Creating an ArrayList
ArrayList< String> fruits = new ArrayList< String>();
// Adding elements to ArrayList
fruits.add("Mango");
fruits.add("Apple");
fruits.add("Berry");
fruits.add("Orange");
// Traversing ArrayList
for(String element : fruits) {
System.out.println(element);
```



```
}
System.out.println("Total Elements: "+fruits.size());
}
}
```

Mango

```
Apple
Berry
Orange
Total Elements: 4
```

Sorting ArrayList Elements

To sort elements of an ArrayList, Java provides a class Collections that includes a static method sort(). In this example, we are using sort method to sort the elements.

```
import java.util.*;
class Demo
 {
 public static void main(String[] args)
 {
 // Creating an ArrayList
 ArrayList< String> fruits = new ArrayList< String>();
 // Adding elements to ArrayList
 fruits.add("Mango");
 fruits.add("Apple");
 fruits.add("Berry");
 fruits.add("Orange");
 // Sorting elements
 Collections.sort(fruits);
 // Traversing ArrayList
 for(String element : fruits) {
 System.out.println(element);
 }
 }
 }
```

```
Apple
Berry
Mango
Orange
```


Java Collection Framework Linked list

Java LinkedList class provides implementation of linked-list data structure. It used doubly linked list to store the elements. It implements List, Deque and Queue interface and extends

AbstractSequentialList class. below is the declaration of LinkedList class.

LinkedList class Declaration

public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>, Deque<E>, Cloneable, Serializable

- LinkedList class extends AbstractSequentialList and implements List,Deque and Queue inteface.
- It can be used as List, stack or Queue as it implements all the related interfaces. •

It allows null entry.

- It is dynamic in nature i.e it allocates memory when required. Therefore insertion and deletion operations can be easily implemented.
- It can contain duplicate elements and it is not synchronized.
- Reverse Traversing is difficult in linked list.
- In LinkedList, manipulation is fast because no shifting needs to be occurred.

LinkedList Constructors

LinkedList class has two constructors. First is used to create empty LinkedList and second for creating from existing collection.

LinkedList() // It creates an empty LinkedList
LinkedList(Collection c) // It creates a LinkedList that is initialized with elements of the Collection c

LinkedList class Example

Lets take an example to create a linked-list, no element is inserted so it creates an empty LinkedList. See the below example.

```
import java.util.*;
class Demo
{
public static void main(String[] args)
{
// Creating LinkedList
LinkedList< String> linkedList = new LinkedList< String>();
// Displaying LinkedLIst
```



```
System.out.println(linkedList);
}
```

}
[]

LinkedList Methods

The below table contains methods of LinkedList. We can use them to manipulate its elements.

• boolean add(E e) - It appends the specified element to the end of a list. • void add(int index, E element) - It inserts the specified element at the specified position index in a list.

• boolean addAll(Collection<? extends E> c) - It appends all of the elements in the specified collection to the end of this list.

• boolean addAll(Collection<? extends E> c) - It appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.

• boolean addAll(int index, Collection<? extends E> c) - It appends all the elements in the specified collection, starting at the specified position of the list.

• void addFirst(E e) - It inserts the given element at the beginning of a list. • void addLast(E e) - It appends the given element to the end of a list.

• void clear() - It removes all the elements from a list.

• Object clone() - It returns a shallow copy of an ArrayList.

• boolean contains(Object o) - It returns true if a list contains a specified element. • Iterator<E> descendingIterator() - It returns an iterator over the elements in a deque in reverse sequential order.

• E element() - It retrieves the first element of a list.

• E get(int index) - It returns the element at the specified position in a list. • E getFirst() - It returns the first element in a list.

• E getLast() - It returns the last element in a list.

• int indexOf(Object o) - It returns the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element.

• int lastIndexOf(Object o) - It is used to return the index in a list of the last occurrence of the specified element, or -1 if the list does not contain any element.

• ListIterator<E> listIterator(int index) - It returns a list-iterator of the elements in proper sequence, starting at the specified position in the list.

• boolean offer(E e) - It adds the specified element as the last element of a list. • boolean offerFirst(E e) - It inserts the specified element at the front of a list. • boolean offerLast(E e) - It inserts the specified element at the end of a list.

• E peek() - It retrieves the first element of a list

• E peekFirst() - It retrieves the first element of a list or returns null if a list is empty. • E peekLast() - It retrieves the last element of a list or returns null if a list is empty. • E poll() - It retrieves and removes the first element of a list.

• E pollFirst() - It retrieves and removes the first element of a list, or returns null if a list is empty.

• E pollLast() - It retrieves and removes the last element of a list, or returns null if a list is empty.

• E pop() - It pops an element from the stack represented by a list.

• void push(E e) - It pushes an element onto the stack represented by a list. • E remove() - It is used to retrieve and removes the first element of a list. • E remove(int index) - It is used to remove the element at the specified position in a list. • boolean remove(Object o)It is used to remove the first occurrence of the specified element in a list.

• E removeFirst() - It removes and returns the first element from a list.

• boolean removeFirstOccurrence(Object o) - It removes the first occurrence of the specified element in a list (when traversing the list from head to tail).

• E removeLast() - It removes and returns the last element from a list.

• boolean removeLastOccurrence(Object o) - It removes the last occurrence of the specified element in a list (when traversing the list from head to tail).

• E set(int index, E element) - It replaces the element at the specified position in a list with the specified element.

• int size() - It returns the number of elements in a list.

Add Elements to LinkedList

To add elements into LinkedList, we are using add() method. It adds elements into the list in the insertion order.

```
import java.util.*;
class Demo
{
public static void main(String[] args)
{
// Creating LinkedList
LinkedList< String> linkedList = new LinkedList< String>();
linkedList.add("Delhi");
linkedList.add("NewYork");
linkedList.add("Moscow");
linkedList.add("Dubai");
// Displaying LinkedList
```



```
System.out.println(linkedList);
```

```
}
}
```

[Delhi, NewYork, Moscow, Dubai]

Removing Elements

To remove elements from the list, we are using remove method that remove the specified elements. We can also pass index value to remove the elements of it.

```
import java.util.*;
class Demo
{
public static void main(String[] args)
{
// Creating LinkedList
LinkedList< String> linkedList = new LinkedList< String>();
linkedList.add("Delhi");
linkedList.add("NewYork");
linkedList.add("Moscow");
linkedList.add("Dubai");
// Displaying LinkedList
System.out.println(linkedList);
// Removing Elements
linkedList.remove("Moscow");
System.out.println("After Deleting Elements: \n"+linkedList);
// Removing Second Element
linkedList.remove(1);
System.out.println("After Deleting Elements: \n"+linkedList);
}
}
```

[Delhi, NewYork, Moscow, Dubai]
After Deleting Elements:
[Delhi, NewYork, Dubai]
After Deleting Elements:
[Delhi, Dubai]

Traversing Elements of LinkedList

Since LinkedList is a collection then we can use loop to iterate its elements. In this example we are traversing elements. See the below example.

```
import java.util.*;
class Demo
{
 public static void main(String[] args)
 {
 // Creating LinkedList
 LinkedList< String> linkedList = new LinkedList< String>();
 linkedList.add("Delhi");
 linkedList.add("NewYork");
 linkedList.add("Moscow");
 linkedList.add("Dubai");
 // Traversing ArrayList
 for(String element : linkedList) {
 System.out.println(element);
 }
 }
}
```

```
Delhi
NewYork
Moscow
Dubai
```

Get size of LinkedList

Sometimes we want to know number of elements an LinkedList holds. In that case we use size() then returns size of LinkedList which is equal to number of elements present in the list.

```
import java.util.*;
class Demo
{
 public static void main(String[] args)
 {
 // Creating LinkedList
 LinkedList< String> linkedList = new LinkedList< String>();
 linkedList.add("Delhi");
 linkedList.add("NewYork");
 linkedList.add("Moscow");
```

```
linkedList.add("Dubai");
 // Traversing ArrayList
for(String element : linkedList) {
System.out.println(element);
}
System.out.println("Total Elements: "+linkedList.size());
}
}
```

```
Delhi
NewYork
Moscow
Dubai
Total Elements: 4
```

Sorting LinkedList Elements

To sort elements of an LinkedList, Java provides a class Collections that includes a static method sort(). In this example, we are using sort method to sort the elements.

```
import java.util.*;
class Demo
{
public static void main(String[] args)
{
// Creating LinkedList
LinkedList< String> linkedList = new LinkedList< String>();
linkedList.add("Delhi");
linkedList.add("NewYork");
linkedList.add("Moscow");
linkedList.add("Dubai");
// Sorting elements
Collections.sort(linkedList);
// Traversing ArrayList
for(String element : linkedList) {
System.out.println(element);
}
}
}
```

Delhi

Dubai
Moscow
NewYork

Java Collection Framework Treemap

The TreeMap class is a implementation of Map interface based on red-black tree. It provides an efficient way of storing key-value pairs in sorted order.

It is similar to HashMap class except it is sorted in the ascending order of its keys. It is not suitable for thread-safe operations due to its unsynchronized nature.

It implements NavigableMap interface and extends AbstractMap class. Declaration of this class is given below.

TreeMap Declaration

public class TreeMap<K,V>extends AbstractMap<K,V>implements NavigableMap<K,V>, Cloneable, Serializable

Important Points:

- It contains only unique elements.
- It cannot have a null key but can have multiple null values.
- It is non synchronized.
- It maintains ascending order.

TreeMap Constructors

• TreeMap() - It creates a new, empty tree map, using the natural ordering of its keys. • TreeMap(Comparator<? super K> comparator) - It created a new, empty tree map, ordered according to the given comparator.
• TreeMap(Map<? extends K,? extends V> m) - It creates a new tree map containing the same mappings as the given map.
• TreeMap(SortedMap<K,? extends V> m) - It creates a new tree map containing sortedamp.

Example : Creating TreeMap

Lets take an example to understand the creation of treeMap which is initially empty.

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



```
// Creating TreeMap
TreeMap<String,Integer> treeMap = new TreeMap<String,Integer>();
// Displaying TreeMap
System.out.println(treeMap);
}
}

{}
```

TreeMap Methods

This table contains the methods of TreeMap that can be used to manipulate objects.

- Map.Entry<K,V> ceilingEntry(K key) - It returns the key-value pair having the least key, greater than or equal to the specified key, or null if there is no such key.
- K ceilingKey(K key) - It returns the least key, greater than the specified key or null if there is no such key.
- void clear() - It removes all the key-value pairs from a map.
- Object clone() - It returns a shallow copy of TreeMap instance.
- Comparator<? super K> comparator() - It returns the comparator that arranges the key in order, or null if the map uses the natural ordering.
- NavigableSet<K> descendingKeySet() - It returns a reverse order NavigableSet view of the keys contained in the map.
- NavigableMap<K,V> descendingMap() - It returns the specified key-value pairs in descending order.
- Map.Entry firstEntry() - It returns the key-value pair having the least key. • Map.Entry<K,V> floorEntry(K key) - It returns the greatest key, less than or equal to the specified key, or null if there is no such key.
- void forEach(BiConsumer<? super K,? super V> action) - It performs the given action for each entry in the map until all entries have been processed or the action throws an exception. • SortedMap<K,V> headMap(K toKey) - It returns the key-value pairs whose keys are strictly less than toKey.
- NavigableMap<K,V> headMap(K toKey, boolean inclusive) - It returns the key-value pairs whose keys are less than (or equal to if inclusive is true) toKey.

• Map.Entry<K,V> higherEntry(K key) - It returns the least key strictly greater than the given key, or null if there is no such key.
• K higherKey(K key) - It is used to return true if this map contains a mapping for the specified key.
• Set keySet() - It returns the collection of keys exist in the map.

• Map.Entry<K,V> lastEntry() - It returns the key-value pair having the greatest key, or null if there is no such key.
• Map.Entry<K,V> lowerEntry(K key) - It returns a key-value mapping associated with the greatest key strictly less than the given key, or null if there is no such key. • K lowerKey(K key) - It returns the greatest key strictly less than the given key, or null if there is no such key.
• NavigableSet navigableKeySet() - It returns a NavigableSet view of the keys contained in this map.
• Map.Entry<K,V> pollFirstEntry() - It removes and returns a key-value mapping associated with the least key in this map, or null if the map is empty.
• Map.Entry<K,V> pollLastEntry() - It removes and returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.
• V put(K key, V value) - It inserts the specified value with the specified key in the map. • void putAll(Map<? extends K,? extends V> map) - It is used to copy all the key-value pair from one map to another map.
• V replace(K key, V value) - It replaces the specified value for a specified key. • boolean replace(K key, V oldValue, V newValue) - It replaces the old value with the new value for a specified key.
• void replaceAll(BiFunction<? super K,? super V,? extends V> function) - It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
• NavigableMap<K,V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive) - It returns key-value pairs whose keys range from fromKey to toKey. • SortedMap<K,V> subMap(K fromKey, K toKey) - It returns key-value pairs whose keys range from fromKey, inclusive, to toKey, exclusive.
• SortedMap<K,V> tailMap(K fromKey) - It returns key-value pairs whose keys are greater than or equal to fromKey.
• NavigableMap<K,V> tailMap(K fromKey, boolean inclusive) - It returns key-value pairs whose keys are greater than (or equal to, if inclusive is true) fromKey.
• boolean containsKey(Object key) - It returns true if the map contains a mapping for the specified key.
• boolean containsValue(Object value) - It returns true if the map maps one or more keys to the

specified value.

• K firstKey() - It is used to return the first (lowest) key currently in this sorted map. • V get(Object key) - It is used to return the value to which the map maps the specified key. • K lastKey() - It is used to return the last (highest) key currently in the sorted map. • V remove(Object key) - It removes the key-value pair of the specified key from the map. • int size() - It returns the number of key-value pairs exists in the hashtable. • Collection values() - It returns a collection view of the values contained in the map.

Example : Adding Elements to TreeMap

After creating TreeMap, lets add elements to it. We used put method to insert elements which takes two arguments: first is key and second is value.

```
class Demo
 {
 public static void main(String args[])
 {
 // Creating TreeMap
 TreeMap<String,Integer> treeMap = new TreeMap<String,Integer>();
// Adding elements
 treeMap.put("a",100);
 treeMap.put("b",200);
 treeMap.put("c",300);
 treeMap.put("d",400);
 // Displaying TreeMap
 System.out.println(treeMap);
 }
```

```
 {a=100, b=200, c=300, d=400}
```

Get first and last key in LinkedHashMap

We can get first and last element of the map using the firstEntry() and lastEntry() method. It returns a pair of key and value.

```
 import java.util.*;
class Demo
 {
 public static void main(String args[])
 {
```

```
 // Creating TreeMap
 TreeMap<String,Integer> treeMap = new TreeMap<String,Integer>();
// Adding elements
 treeMap.put("a",100);
 treeMap.put("b",200);
 treeMap.put("c",300);
 treeMap.put("d",400);
 // Displaying TreeMap
 System.out.println(treeMap);
```



```
 // First Element
 System.out.println("First Element: "+treeMap.firstEntry());
 // Last Element
 System.out.println("Last Element: "+treeMap.lastEntry());

 }
}
```

```
 {a=100, b=200, c=300, d=400}
First Element: a=100
Last Element: d=400
```

Example: Find HeadMap and TailMap of the TreeMap

We can get head elements and tail elements of the TreeMap by using the headMap() and tailMap() methods. These methods return a map containing the elements.

```
 import java.util.*;
class Demo
{
 public static void main(String args[])
 {
 // Creating TreeMap
 TreeMap<String,Integer> treeMap = new TreeMap<String,Integer>();
 // Adding elements
 treeMap.put("a",100);
 treeMap.put("b",200);
 treeMap.put("c",300);
 treeMap.put("d",400);
```

```
// Displaying TreeMap
System.out.println(treeMap);
// Head Map
System.out.println("Head Map: "+treeMap.headMap("c"));
// Tail Map
System.out.println("Tail Map: "+treeMap.tailMap("c"));

}
}
```

{a=100, b=200, c=300, d=400}

Head Map: {a=100, b=200}
Tail Map: {c=300, d=400}

Example: Iterating TreeMap

In this example, we are creating TreeMap to store data. It uses tree to store data and data is always in sorted order and iterating the map using loop. See the below example.

```
import java.util.*;
class Demo
{
public static void main(String args[])
{
TreeMap<String,Integer> tm = new TreeMap<String,Integer>();
tm.put("a",100);
tm.put("b",200);
tm.put("c",300);
tm.put("d",400);

Set<Map.Entry<String,Integer>> st = tm.entrySet();
for(Map.Entry<String,Integer> me:st)
{
System.out.print(me.getKey()+":");
System.out.println(me.getValue());
}
}
}
```

a:100
b:200
c:300
d:400

Java Collection Framework Hashtable

Java Hashtable is an implementation of hash table which stores elements in key-value pair. It does not allow null key and null values. It is synchronized version of HashMap.

It extends Dictionary class and implements Map interface. Declaration of this class is given below.

Hashtable Declaration

public class Hashtable<K,V>extends Dictionary<K,V>implements Map<K,V>, Cloneable, Serializable

Important Points

- It contains values based on the key.
- It contains unique elements.
- It doesn't allow null key or value.
- It is synchronized.
- The initial default capacity of Hashtable is 11.

In this tutorial, we will learn to create a Hashtable, add new elements to it, traverse its elements etc.

Hashtable Constructors

• Hashtable() - It creates an empty hashtable having the initial default capacity and load factor. • Hashtable(int capacity) - It accepts an integer parameter and creates a hash table that contains a specified initial capacity.
• Hashtable(int capacity, float loadFactor) - It is used to create a hash table having the specified initial capacity and loadFactor.
• Hashtable(Map<? extends K,? extends V> t) - It creates a new hash table with the same mappings as the given Map.

Example: Creating Hashtable

Lets take an example to create hashtable that takes elements of string and integer type pair. Initially it is empty so if we print it, it shows empty braces. We will learn to add elements in next example.

```
 import java.util.*;
class Demo
 {
 public static void main(String args[])
 {
 // Creating Hashtable
 Hashtable<String,Integer> hashtable = new Hashtable<String,Integer>();

 // Displaying Hashtable
 System.out.println(hashtable);
```



```
 }
}

 {}
```

Hashtable Methods

- void clear() - It empty the hash table.
- Object clone() - It returns a shallow copy of the Hashtable.
- V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) - It computes a mapping for the specified key and its current mapped value.
- V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction) - It computes its value using the given mapping function.
- Enumeration elements() - It returns an enumeration of the values in the hash table. • Set<Map.Entry<K,V>> entrySet() - It returns a set view of the mappings contained in the map.
- boolean equals(Object o) - It compares the specified Object with the Map. • void forEach(BiConsumer<? super K,? super V> action) - It performs the given action for each entry in the map until all entries have been processed or the action throws an exception. • V getOrDefault(Object key, V defaultValue) - It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key.
- int hashCode() - It returns the hash code value for the Map
- Enumeration<K> keys() - It returns an enumeration of the keys in the hashtable. • Set<K>

keySet() - It returns a Set view of the keys contained in the map. • V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)
- If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.

• V put(K key, V value) - It inserts the specified value with the specified key in the hash table. • void putAll(Map<? extends K,? extends V> t)) - It is used to copy all the key-value pair from map to hashtable.

• V putIfAbsent(K key, V value) - If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.

• boolean remove(Object key, Object value) - It removes the specified values with the associated specified keys from the hashtable.

• V replace(K key, V value) - It replaces the specified value for a specified key. • String toString() - It returns a string representation of the Hashtable object. • Collection values() - It returns a collection view of the values contained in the map.

• boolean contains(Object value) - This method returns true if some value equal to the value exists within the hash table, else return false.

• boolean containsValue(Object value) - This method returns true if some value equal to the value exists within the hash table, else return false.

• boolean containsKey(Object key) - This method return true if some key equal to the key exists within the hash table, else return false.

• boolean isEmpty() - This method returns true if the hash table is empty; returns false if it contains at least one key.

• protected void rehash() - It is used to increase the size of the hash table and rehashes all of its keys.

• V get(Object key) - This method returns the object that contains the value associated with the key.

• V remove(Object key) - It is used to remove the key and its value. This method returns the value associated with the key.

• int size() - This method returns the number of entries in the hash table.

Adding Elements to Hashtable

To insert elements into the hashtable, we are using put() method that adds new elements. It takes two argument: first is key and second is value.

```
import java.util.*;
class Demo
```

```
{
public static void main(String args[])
{
// Creating Hashtable
Hashtable<String,Integer> hashtable = new Hashtable<String,Integer>();
// Adding elements
hashtable.put("a",100);
hashtable.put("b",200);
hashtable.put("c",300);
hashtable.put("d",400);
// Displaying Hashtable
System.out.println(hashtable);

}
}
```

```
{b=200, a=100, d=400, c=300}
```


Example: No Null Allowed

Since hashtable does not allow null key or value then forcing to insert the null will throw an error. See the below example.

```
import java.util.*;
class Demo
{
public static void main(String args[])
{
// Creating Hashtable
Hashtable<String,Integer> hashtable = new Hashtable<String,Integer>();
// Adding elements
hashtable.put("a",100);
hashtable.put("b",200);
hashtable.put("c",300);
hashtable.put("d",400);
hashtable.put(null, 0); // error: no null allowed
// Displaying Hashtable
System.out.println(hashtable);
```

```
}
}
```

Exception in thread "main" java.lang.NullPointerException

Example: Search for a key or value

Hashtable provides various methods such as contains(), containsKey() etc to search for an element in the hashtable. Contains() method search for specified value while containsKey() method search for specified key.

```
 import java.util.*;
class Demo
 {
 public static void main(String args[])
 {
 // Creating Hashtable
 Hashtable<String,Integer> hashtable = new Hashtable<String,Integer>();
// Adding elements
 hashtable.put("a",100);
```



```
 hashtable.put("b",200);
 hashtable.put("c",300);
 hashtable.put("d",400);
 // Displaying Hashtable
 System.out.println(hashtable);
 // Search for a value
 boolean val = hashtable.contains(400);
 System.out.println("is 400 present: "+val);
 // Search for a key
 val = hashtable.containsKey("d");
 System.out.println("is d present: "+val);
 }
 }
```

```
 {b=200, a=100, d=400, c=300}
is 400 present: true
is d present: true
```

Example: Traverse Hashtable

We can traverse the hashtable elements using loop to access its elements and to get list of all the available elements.

```
import java.util.*;
class Demo
 {
 public static void main(String args[])
 {
 // Creating Hashtable
 Hashtable<String,Integer> hashtable = new Hashtable<String,Integer>();
// Adding elements
 hashtable.put("a",100);
 hashtable.put("b",200);
 hashtable.put("c",300);
 hashtable.put("d",400);
 // Traversing Hashtable
 for(Map.Entry<String, Integer> m : hashtable.entrySet()) {
 System.out.println(m.getKey()+" "+m.getValue());
 }
 }
```



```
}
```

```
b 200
a 100
d 400
c 300
```

Java Vector

Vector is like the dynamic array which can grow or shrink its size. Unlike array, we can store n number of elements in it as there is no size limit. It is a part of Java Collection framework since Java 1.2. It is found in the java.util package and implements the List interface, so we can use all the methods of List interface here.

It is recommended to use the Vector class in the thread-safe implementation only. If you don't need to use the thread-safe implementation, you should use the ArrayList, the ArrayList will perform better in such case.

The Iterators returned by the Vector class are fail-fast. In case of concurrent modification, it fails and throws the ConcurrentModificationException.

It is similar to the ArrayList, but with two differences-

- Vector is synchronized.
- Java Vector contains many legacy methods that are not the part of a collections framework.

Java Vector class Declaration

- public class Vector<E>
- extends Object<E>
- implements List<E>, Cloneable, Serializable

Java Vector Constructors

Vector class supports four types of constructors. These are given below:

- vector() - It constructs an empty vector with the default size as 10.
- vector(int initialCapacity) - It constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.
- vector(int initialCapacity, int capacityIncrement) - It constructs an empty vector with the specified initial capacity and capacity increment.

- Vector(Collection<? extends E> c) - It constructs a vector that contains the elements of a collection c.

Java Vector Methods

The following are the list of Vector class methods:

1. add() - It is used to append the specified element in the given vector.
2. addAll() - It is used to append all of the elements in the specified collection to the end of this Vector.
3. addElement() - It is used to append the specified component to the end of this vector. It increases the vector size by one.
4. capacity() - It is used to get the current capacity of this vector.
5. clear() - It is used to delete all of the elements from this vector.

6. clone() - It returns a clone of this vector.
7. contains() - It returns true if the vector contains the specified element.
8. containsAll() - It returns true if the vector contains all of the elements in the specified collection.
9. copyInto() - It is used to copy the components of the vector into the specified array.
10. elementAt() - It is used to get the component at the specified index.
11. elements() - It returns an enumeration of the components of a vector.
12. ensureCapacity() - It is used to increase the capacity of the vector which is in use, if necessary. It ensures that the vector can hold at least the number of components specified by the minimum capacity argument.
13. equals() - It is used to compare the specified object with the vector for equality. 14. firstElement() - It is used to get the first component of the vector.
15. forEach() - It is used to perform the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
16. get() - It is used to get an element at the specified position in the vector. 17. hashCode() - It is used to get the hash code value of a vector.
18. indexOf() - It is used to get the index of the first occurrence of the specified element in the vector. It returns -1 if the vector does not contain the element.
19. insertElementAt() - It is used to insert the specified object as a component in the given vector at the specified index.
20. isEmpty() - It is used to check if this vector has no components.
21. iterator() - It is used to get an iterator over the elements in the list in proper sequence. 22. lastElement() - It is used to get the last component of the vector.
23. lastIndexOf() - It is used to get the index of the last occurrence of the specified element in the vector. It returns -1 if the vector does not contain the element.
24. listIterator() - It is used to get a list iterator over the elements in the list in proper sequence.

25. remove() - It is used to remove the specified element from the vector. If the vector does not contain the element, it is unchanged.
26. removeAll() - It is used to delete all the elements from the vector that are present in the specified collection.
27. removeAllElements() - It is used to remove all elements from the vector and set the size of the vector to zero.
28. removeElement() - It is used to remove the first (lowest-indexed) occurrence of the argument from the vector.
29. removeElementAt() - It is used to delete the component at the specified index. 30. removeIf() - It is used to remove all of the elements of the collection that satisfy the given predicate.
31. removeRange() - It is used to delete all of the elements from the vector whose index is

between fromIndex, inclusive and toIndex, exclusive.

32. replaceAll() - It is used to replace each element of the list with the result of applying the operator to that element.
33. retainAll() - It is used to retain only that element in the vector which is contained in the specified collection.
34. set() - It is used to replace the element at the specified position in the vector with the specified element.
35. setElementAt() - It is used to set the component at the specified index of the vector to the specified object.
36. setSize() - It is used to set the size of the given vector.
37. size() - It is used to get the number of components in the given vector. 38. sort() - It is used to sort the list according to the order induced by the specified Comparator. 39. spliterator() - It is used to create a late-binding and fail-fast Spliterator over the elements in the list.
40. subList() - It is used to get a view of the portion of the list between fromIndex, inclusive, and toIndex, exclusive.
41. toArray() - It is used to get an array containing all of the elements in this vector in correct order.
42. toString() - It is used to get a string representation of the vector.
43. trimToSize() - It is used to trim the capacity of the vector to the vector's current size.

Java Vector Example

```
import java.util.*;
public class VectorExample {
 public static void main(String args[]) {
 //Create a vector
 Vector<String> vec = new Vector<String>();
 //Adding elements using add() method of List
```



```
 vec.add("Tiger");
 vec.add("Lion");
 vec.add("Dog");
 vec.add("Elephant");
 //Adding elements using addElement() method of Vector
vec.addElement("Rat");
 vec.addElement("Cat");
 vec.addElement("Deer");

 System.out.println("Elements are: "+vec);
 }
```

```
}
```
Output:
Elements are: [Tiger, Lion, Dog, Elephant, Rat, Cat, Deer]

Java Vector Example 2

```
import java.util.*;
public class VectorExample1 {
public static void main(String args[]) {
 //Create an empty vector with initial capacity 4
 Vector<String> vec = new Vector<String>(4);
 //Adding elements to a vector
 vec.add("Tiger");
 vec.add("Lion");
 vec.add("Dog");
 vec.add("Elephant");
 //Check size and capacity
 System.out.println("Size is: "+vec.size());
 System.out.println("Default capacity is: "+vec.capacity());
//Display Vector elements
 System.out.println("Vector element is: "+vec);
 vec.addElement("Rat");
 vec.addElement("Cat");
 vec.addElement("Deer");
 //Again check size and capacity after two insertions
System.out.println("Size after addition: "+vec.size());
System.out.println("Capacity after addition is: "+vec.capacity());
//Display Vector elements again
 System.out.println("Elements are: "+vec);
```



```
 //Checking if Tiger is present or not in this vector
 if(vec.contains("Tiger"))
 {
 System.out.println("Tiger is present at the index " +vec.indexOf("Tiger"));   }
 else
 {
 System.out.println("Tiger is not present in the list.");
 }
 //Get the first element
```

```
 System.out.println("The first animal of the vector is = "+vec.firstElement());
//Get the last element
 System.out.println("The last animal of the vector is = "+vec.lastElement());   }
}
```

Output:

```
Size is: 4
Default capacity is: 4
Vector element is: [Tiger, Lion, Dog, Elephant]
Size after addition: 7
Capacity after addition is: 8
Elements are: [Tiger, Lion, Dog, Elephant, Rat, Cat, Deer]
Tiger is present at the index 0
The first animal of the vector is = Tiger
The last animal of the vector is = Deer
```

Java Vector Example 3

```
import java.util.*;
public class VectorExample2 {
 public static void main(String args[]) {
 //Create an empty Vector
 Vector<Integer> in = new Vector<>();
 //Add elements in the vector
 in.add(100);
 in.add(200);
 in.add(300);
 in.add(200);
 in.add(400);
```



```
 in.add(500);
 in.add(600);
 in.add(700);
 //Display the vector elements
 System.out.println("Values in vector: " +in);
 //use remove() method to delete the first occurence of an element
 System.out.println("Remove first occourence of element 200: "+in.remove((Integer)200));
//Display the vector elements afre remove() method
 System.out.println("Values in vector: " +in);
```

```
//Remove the element at index 4
System.out.println("Remove element at index 4: " +in.remove(4));
System.out.println("New Value list in vector: " +in);
//Remove an element
in.removeElementAt(5);
//Checking vector and displays the element
System.out.println("Vector element after removal: " +in);
//Get the hashcode for this vector
System.out.println("Hash code of this vector = "+in.hashCode());
//Get the element at specified index
System.out.println("Element at index 1 is = "+in.get(1));
}
}
```

Output:
Values in vector: [100, 200, 300, 200, 400, 500, 600, 700]
Remove first occourence of element 200: true
Values in vector: [100, 300, 200, 400, 500, 600, 700]
Remove element at index 4: 500
New Value list in vector: [100, 300, 200, 400, 600, 700]
Vector element after removal: [100, 300, 200, 400, 600]
Hash code of this vector = 130123751
Element at index 1 is = 300

Java Dictionary Class

Java dictionary is an inbuilt class that helps us to form different key-value relationships. It is an abstract class which extends the Object class and is present in the java.util package. The Dictionary class is an abstract parent of any class, such as the Hashtable, which maps the keys to values. Every key and every value is the object in anyone.

Java Dictionary Class

Java dictionary can also be termed as a list of pair values, where the pair consists of the key and the value. The relationship of the key to the values is somewhat similar to mapping. First, we map the key with the value, and then afterward, we can again retrieve the value with the help of the key.

Declaration

If we want to declare a Dictionary in java, we have to write the below method from java.util.Dictionary.keys().

public abstract Enumeration<K> keys()

Various Methods in Dictionary Packages

The following are the various methods that are predefined inside the java.util.Dictionary package.

1. put(key, value) - The put() method takes two arguments as shown, and is used to map the key with the value and store it in the dictionary.
   Syntax: public abstract V put(K key, V value)
   Parameters:
   key
   value
   Return: key-value pair mapped in the dictionary.
2. get() - The get() method takes the key as the argument and returns the value that is mapped to it. If no value is mapped with the given key, it simply returns null.
   Syntax: public abstract V get(Object key)
   Parameters:
   key – key whose mapped value we want
   Return: value mapped with the argument key.
3. elements() - The elements() method is used to represent all the values present inside the Dictionary. It is usually used with loop statements as they can then represent one value at a time.
   Syntax: public abstract Enumeration elements()
   Return: value enumeration in the dictionary.
4. keys() - As the elements() method returns the enumerated values present inside the dictionary; similarly, the keys() method returns the enumerated keys present inside the dictionary.
   Syntax: public abstract Enumeration keys()
   Return: The key enumeration in the dictionary.

5. isEmpty() - The isEmpty() method returns a boolean value, which is true if there are no key value pairs present inside the Dictionary. If even any single key-value pair resides inside the dictionary, it returns false.
   Syntax: public abstract boolean isEmpty()
   Return: It returns true if there is no key-value relation in the dictionary; else false. 6. remove(key) - The remove() method takes the key as its argument, and it simply removes both the key and the value mapped with it from the dictionary.

Syntax: public abstract V remove(Object key)
Parameters:
key: a key to be removed
Return: The key enumeration in the dictionary.

7. size() - The size() method returns the total number of key-value pairs present inside the Dictionary.
Syntax: public abstract int size()
Return: It returns the no. of key-value pairs in the Dictionary.

The following program code is an example of using Dictionaries in Java.

```
import java.util.Dictionary;
import java.util.Enumeration;
import java.util.Hashtable;

public class Dict {
 public static void main(String[] args) {

 Dictionary dictionary = new Hashtable();

 // put method
 dictionary.put("Apple", "A fruit");
 dictionary.put("Ball", "A round shaped toy");
 dictionary.put("Car", "A four wheeler vehicle designed to accomodate usually four people");
dictionary.put("Dog", "An animal with four legs and one tail");

 // get method
 System.out.println("\nApple: " + dictionary.get("Apple"));
 System.out.println("Dog: " + dictionary.get("Dog"));
 System.out.println("Elephant: " + dictionary.get("Elephant"));
 System.out.println();
```



```
 // elements method
 for (Enumeration i = dictionary.elements(); i.hasMoreElements();) {
System.out.println("Values contained in Dictionary : " + i.nextElement());  }
 System.out.println();
```

```
 // keys method :
 for (Enumeration k = dictionary.keys(); k.hasMoreElements();) {
System.out.println("Keys contianed in Dictionary : " + k.nextElement());  }

 // isEmpty method
 System.out.println("\nThe dictionary is empty? " + dictionary.isEmpty());

 // remove method :
 dictionary.remove("Dog");

 // Checking if the value is removed or not
 System.out.println("\nDog: " + dictionary.get("Dog"));

 // size method
 System.out.println("\nSize of Dictionary : " + dictionary.size());
}
}
```