

UNIT IV

MULTITHREADING

MULTITHREADING

- Java provides builtin support for multithreaded programming.
- A multithreaded program contains two or more parts that can run concurrently.
- Each part of such a program is called a thread, and each thread defines a separate path of execution.
- Thus, multithreading is a specialized form of multitasking.

TYPES OF MULTITASKING

Processbased Multitasking

- A process is, in essence, a program that is executing.
- Thus, processbased multitasking is the feature that allows your computer to run two or more programs concurrently.
- *Ex: Processbased multitasking enables you to run the Java compiler at the same time that you are using a text editor.*

Threadbased Multitasking

- In a threadbased multitasking environment, the thread is the smallest unit of dispatchable code.
- This means that a single program can perform two or more tasks simultaneously.
- *Ex: A text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.*

Multitasking Threads	Multitasking Processes
Light weight tasks	Heavyweight tasks
Context switching Not Costly	Context switching – Costly
Share the same address space	Share the different address space
Less Idle time in CPU	More Idle time in CPU

The Java Thread Model

- The Java runtime system depends on threads for many things, and all the class libraries are designed with multithreading in mind.
- Java uses threads to enable the entire environment to be asynchronous.
- This helps reduce inefficiency by preventing the waste of CPU cycles.

SingleThreaded system

- Singlethreaded systems use an approach called an event loop with polling.
- In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next.
- Once this polling mechanism returns with, say, a signal that a network file is ready to be read,

then the event loop dispatches control to the appropriate event handler.

- Until this event handler returns, nothing else can happen in the system.
- This wastes CPU time. It can also result in one part of a program dominating the system and preventing any other events from being processed.
- In general, in a singlethreaded environment, when a thread blocks (that is, suspends execution) because it is waiting for some resource, the entire program stops running.

Java's Multithreading System

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

THREADS

Threads exist in several states.

Running

Ready to

Run

Suspended

Resumed

Blocked

Terminated

- A thread can be running.
- It can be ready to run as soon as it gets CPU time.
- A running thread can be suspended, which temporarily suspends its activity.
- A suspended thread can then be resumed, allowing it to pick up where it left off.
- A thread can be blocked when waiting for a resource.
- At any time, a thread can be terminated, which halts its execution immediately.
- Once terminated, a thread cannot be resumed.

The Thread Class and the Runnable Interface

- Java's multithreading system is built upon the Thread class, its methods, and its companion interface, Runnable.
- To create a new thread, your program will either extend Thread or implement the Runnable interface.

The Thread class defines several methods that help manage threads. They are...

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
Join	Wait for a thread to terminate.
Run	Entry point for the thread.
Sleep	Suspend a thread for a period of time.
Start	Start a thread by calling its run method.

The Main Thread

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

The main thread is important for two reasons:

- It is the thread from which other “child” threads will be spawned.
- Often it must be the last thread to finish execution because it performs various shutdown actions.
 - Although the main thread is created automatically when your program is started, it can be controlled through a Thread object.
 - To do so, you must obtain a reference to it by calling the method `currentThread()`, which is a public static member of Thread.
 - Its general form is shown here:

`static Thread currentThread()`

Program 1

```
class CurrentThreadDemo
{
    public static void main(String args[])
    {
        Thread t = Thread.currentThread();
        System.out.println("Current thread: " + t);
        t.setName("My Thread");
        System.out.println("After name change: "
            + t); try
        {
            for(int n = 5; n > 0; n)
            {
```

```

System.out.println(n);
Thread.sleep(1000);
}
}
catch (InterruptedException e)
{
System.out.println("Main thread interrupted");
}
}
}

```

Sleep()

- The sleep() method causes the thread from which it is called to suspend execution for the specified period of milliseconds.

Its general form is shown here:

static void sleep(long *milliseconds*) throws *InterruptedException*

Creating a Thread

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

Implementing Runnable

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

public void run()

- run() establishes the entry point for another, concurrent thread of execution within your program.
- This thread will end when run() returns.

Thread defines several constructors. The one that we will use is:

Thread(Runnable threadOb, String threadName)

- In this constructor, threadOb is an instance of a class that implements the Runnable interface.
- This defines where execution of the thread will begin. The name of the new thread is specified by threadName.

- After the new thread is created, it will not start running until you call its start() method, which is declared within Thread.
- In essence, start() executes a call to run().
- The start() method is shown here:

```
void start( )
```

Extending Thread

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

Program 3

```
class NewThread extends Thread
{ NewThread() {
// Create a new, second thread
super("Demo Thread");
System.out.println("Child thread: " +
this); start(); // Start the thread
}
// This is the entry point for the second
thread. public void run() {
try {
for(int i = 5; i > 0; i) {
System.out.println("Child Thread: " +
i); Thread.sleep(500);
}
} catch (InterruptedException e) {
```

```

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

```

Creating Multiple Threads

- Till now we have seen a Main Thread & one Child Thread.
- It is also possible to create multiple Threads.
- The following program demonstrates creation of multiThreads. Program 4

```

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

```

```

class MultiThreadDemo {
public static void main(String args[]) {
new NewThread("One"); // start
threads new NewThread("Two");

new
NewThread("Three"); try
{
// wait for other threads to
end Thread.sleep(10000);
} catch (InterruptedException e) {
System.out.println("Main thread
Interrupted");
}
System.out.println("Main thread exiting.");
}
}

```

Using isAlive() and join()

- How can one thread know when another thread has ended?
- Fortunately, Thread provides a means by which you can answer this question.
- Two ways exist to determine whether a thread has finished.

Alive()

- First, you can call isAlive() on the thread.
- This method is defined by Thread, and its general form is shown here:

final boolean isAlive()
- The isAlive() method returns true if the thread upon which it is called is still running. It returns false otherwise.
- Alive is occasionally used.

Join()

- The method that you will more commonly use to wait for a thread to finish is called join(), shown here:

final void join() throws InterruptedException

- This method waits until the thread on which it is called terminates.
- Its name comes from the concept of the calling thread waiting until the specified thread *joins*

it. Program 5

```

class NewThread implements Runnable
{ String name; // name of thread
Thread t;

NewThread(String threadname)
{ name = threadname;
t = new Thread(this, name);

```

```

System.out.println("New thread: " +
t); t.start(); // Start the thread
}
// This is the entry point for
thread. public void run() {
try {
for(int i = 5; i > 0; i) {
System.out.println(name + ": " +
i);

Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println(name + "
interrupted.");
}
System.out.println(name + " exiting.");
}
}
class DemoJoin {
public static void main(String args[]) {
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
NewThread ob3 = new
NewThread("Three");
System.out.println("Thread One is alive: "
+ ob1.t.isAlive());
System.out.println("Thread Two is alive:
" + ob2.t.isAlive());
System.out.println("Thread Three is alive:
" + ob3.t.isAlive());
// wait for threads to
finish try {
System.out.println("Waiting for threads to
finish."); ob1.t.join();
ob2.t.join();
ob3.t.join();
} catch (InterruptedException e) {
System.out.println("Main thread
Interrupted");
}
System.out.println("Thread One is alive: "
+ ob1.t.isAlive());
System.out.println("Thread Two is alive:

```



```

" + ob2.t.isAlive());
System.out.println("Thread Three is alive:
" + ob3.t.isAlive());
System.out.println("Main thread exiting.");
}
}

```

Thread Priorities

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.
- A higherpriority thread can also preempt a lowerpriority one.
- For Example, when a lowerpriority thread is running and a higherpriority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lowerpriority thread.
- To set a thread's priority, use the `setPriority()` method, which is a member of `Thread`. This is its general form:

```
final void setPriority(int level)
```

- The value of `level` must be within the range `MIN_PRIORITY` and `MAX_PRIORITY`.
- Currently, these values are 1 and 10, respectively.
- To return a thread to default priority, specify `NORM_PRIORITY`, which is currently 5.
- These priorities are defined as final variables within `Thread`.

Program 6

```

class clicker implements Runnable
{ int click = 0;
  Thread t;
  private volatile boolean running =
  true; public clicker(int p) {
  t = new
  Thread(this);
  t.setPriority(p);
  }
  public void run()
  { while (running)
  { click++;
  }
  }
  public void stop()
  { running = false;
  }
  public void start()
  { t.start();
  }
}

```

```

}
class HiLoPri {
public static void main(String args[]) {
Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
lo.start();

hi.start()
; try {
Thread.sleep(10000);
} catch (InterruptedException e) {
System.out.println("Main thread
interrupted.");
}
lo.stop();
hi.stop();
// Wait for child threads to
terminate. try {
hi.t.join();
lo.t.join();
} catch (InterruptedException e) {
System.out.println("InterruptedException
caught");
}
System.out.println("Lowpriority thread: " +
lo.click); System.out.println("Highpriority thread: "
+ hi.click);
}
}

```

Synchronization

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.
- The process by which this is achieved is called synchronization.
- Key to synchronization is the concept of the monitor (also called a *semaphore*).

Monitor

- A monitor is an object that is used as a mutually exclusive lock, or mutex.
- Only one thread can own a monitor at a given time.

- When a thread acquires a lock, it is said to have entered the monitor.
- All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.
- These other threads are said to be waiting for the monitor.
- A thread that owns a monitor can reenter the same monitor if it so desires.

Java implements synchronization through language elements in either one of the two ways.

1. Using Synchronized Methods

2. The synchronized Statement

Using Synchronized Methods

```

Program 7(unsynchronized
Thread) class Callme {
void call(String msg) {
System.out.print "[" +
msg); try {
Thread.sleep(1000);
} catch (InterruptedException e)
{
System.out.println("Interrupted"
);
}
System.out.println("]");
}
}
class Caller implements Runnable
{ String msg;
Callme
target;
Thread t;
public Caller(Callme targ, String
s) { target = targ;
msg = s;
t = new
Thread(this);
t.start();
}
public void run() {

target.call(msg);
}
}
class Synch {
public static void main(String
args[]) { Callme target = new
Callme();

```

```

Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target,
"Synchronized"); Caller ob3 = new
Caller(target, "World");
// wait for threads to
end try {
ob1.t.join();
ob2.t.join();
ob3.t.join();
} catch (InterruptedException e)
{
System.out.println("Interrupted"
);
}
}
}

```

The synchronized Statement

- Second solution is to call the methods defined by the class inside a synchronized block.
- This is the general form of the synchronized statement:

```

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

```

Program 8

```

class Callme
{
void call(String msg) {
System.out.print "[" +
msg); try {
Thread.sleep(1000);
} catch (InterruptedException e)
{
System.out.println("Interrupted"
);
}
System.out.println("]");
}
}
class Caller implements Runnable
{ String msg;
Callme
target;
Thread t;

```

```

public Caller(Callme targ, String
s) { target = targ;
msg = s;
t = new
Thread(this);
t.start();
}

```

```

// synchronize calls to
call() public void run() {
synchronized(target) { // synchronized
block target.call(msg);
}
}
}
class Synch1 {
public static void main(String
args[]) { Callme target = new
Callme();
Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target,
"Synchronized"); Caller ob3 = new
Caller(target, "World");
// wait for threads to end
try {
ob1.t.join()
;
ob2.t.join()
;
ob3.t.join()
;
} catch(InterruptedException e)
{
System.out.println("Interrupted"
);
}
}
}

```

Interthread Communication

- To avoid polling, Java includes an elegant interprocess communication mechanism via the wait(), notify(), and notifyAll() methods.
wait() tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify().

notify() wakes up the first thread that called wait() on the same object.

notifyAll() wakes up all the threads that called wait() on the same object. The highest priority thread will run first.

These methods are declared within Object, as shown
here: final void wait() throws
InterruptedException final void notify()
final void notifyAll()

Program 9 (Correct Implementation of Producer & Consumer Problem)

```
class Q {
int n;
boolean valueSet =
false; synchronized int
get() { if(!valueSet)
try {
wait(
);
} catch(InterruptedException e) {
System.out.println("InterruptedException
caught");
}
System.out.println("Got: " + n);

valueSet =
false; notify();
return n;
}
synchronized void put(int n)
{ if(valueSet)
try {
wait(
);
} catch(InterruptedException e) {
System.out.println("InterruptedException
caught");
}
this.n = n;
valueSet =
true;
System.out.println("Put: " +
n); notify();
}
}
class Producer implements Runnable {
```

```

Q q;
Producer(Q q)
{ this.q = q;
  new Thread(this, "Producer").start();
}
public void run()
{ int i = 0;
  while(true) {
    q.put(i++);
  }
}
}
class Consumer implements Runnable {
  Q q;
  Consumer(Q q)
  { this.q = q;
    new Thread(this, "Consumer").start();
  }
  public void run()
  { while(true) {
    q.get();
  }
}
}
class PCFixed {
  public static void main(String
  args[]) { Q q = new Q();
  new Producer(q);
  new
  Consumer(q);
  System.out.println("Press ControlC to stop.");
}
}

```

Deadlock

- A special type of error that you need to avoid that relates specifically to multitasking is “deadlock”.
- It occurs when two threads have a circular dependency on a pair of synchronized objects.

Deadlock is a difficult error to debug for two reasons:

- In general, it occurs only rarely, when the two threads timeslice in just the right way.
- It may involve more than two threads and two synchronized objects.

Suspending, Resuming, and Stopping Threads

- `suspend()` and `resume()`, which are methods defined by `Thread`, to pause and restart the execution of a thread. They have the form shown below:

```
final void  
suspend()  
final void  
resume()
```

- The `Thread` class also defines a method called `stop()` that stops a thread. Its signature is shown here:
`final void stop()`

The Collections Framework

Class Hierarchy

- `java.lang.Object`
 - `java.util.AbstractCollection<E>` (implements `java.util.Collection<E>`)
 - `java.util.AbstractList<E>` (implements `java.util.List<E>`)
 - `java.util.AbstractSequentialList<E>`
 - `java.util.LinkedList<E>` (implements `java.lang.Cloneable`, `java.util.List<E>`, `java.util.Queue<E>`, `java.io.Serializable`)
 - `java.util.ArrayList<E>` (implements `java.lang.Cloneable`, `java.util.List<E>`, `java.util.RandomAccess`, `java.io.Serializable`)
 - `java.util.Vector<E>` (implements `java.lang.Cloneable`, `java.util.List<E>`, `java.util.RandomAccess`, `java.io.Serializable`)
 - `java.util.Stack<E>`
 - `java.util.AbstractQueue<E>` (implements `java.util.Queue<E>`)
 - `java.util.PriorityQueue<E>` (implements `java.io.Serializable`)
 - `java.util.AbstractSet<E>` (implements `java.util.Set<E>`)
 - `java.util.EnumSet<E>` (implements `java.lang.Cloneable`, `java.io.Serializable`)

- java.util.[HashSet](#)<E> (implements java.lang.[Cloneable](#), java.io.[Serializable](#), java.util.[Set](#)<E>)
 - java.util.[LinkedHashSet](#)<E> (implements java.lang.[Cloneable](#), java.io.[Serializable](#), java.util.[Set](#)<E>)
 - java.util.[TreeSet](#)<E> (implements java.lang.[Cloneable](#), java.io.[Serializable](#), java.util.[SortedSet](#)<E>)
- java.util.[AbstractMap](#)<K,V> (implements java.util.[Map](#)<K,V>)
- java.util.[EnumMap](#)<K,V> (implements java.lang.[Cloneable](#), java.io.[Serializable](#))
 - java.util.[HashMap](#)<K,V> (implements java.lang.[Cloneable](#), java.util.[Map](#)<K,V>, java.io.[Serializable](#))
 - java.util.[LinkedHashMap](#)<K,V> (implements java.util.[Map](#)<K,V>)
 - java.util.[IdentityHashMap](#)<K,V> (implements java.lang.[Cloneable](#), java.util.[Map](#)<K,V>, java.io.[Serializable](#))
 - java.util.[TreeMap](#)<K,V> (implements java.lang.[Cloneable](#), java.io.[Serializable](#), java.util.[SortedMap](#)<K,V>)
 - java.util.[WeakHashMap](#)<K,V> (implements java.util.[Map](#)<K,V>)
- java.util.[Arrays](#)
- java.util.[BitSet](#) (implements java.lang.[Cloneable](#), java.io.[Serializable](#))
- java.util.[Calendar](#) (implements java.lang.[Cloneable](#), java.lang.[Comparable](#)<T>, java.io.[Serializable](#))
- java.util.[GregorianCalendar](#)
- java.util.[Collections](#)
- java.util.[Currency](#) (implements java.io.[Serializable](#))
- java.util.[Date](#) (implements java.lang.[Cloneable](#), java.lang.[Comparable](#)<T>, java.io.[Serializable](#))
- java.util.[Dictionary](#)<K,V>
 - java.util.[Hashtable](#)<K,V> (implements java.lang.[Cloneable](#), java.util.[Map](#)<K,V>, java.io.[Serializable](#))
 - java.util.[Properties](#)
- java.util.[EventListenerProxy](#) (implements java.util.[EventListener](#))
- java.util.[EventObject](#) (implements java.io.[Serializable](#))
- java.util.[FormattableFlags](#)
- java.util.[Formatter](#) (implements java.io.[Closeable](#), java.io.[Flushable](#))
- java.util.[Locale](#) (implements java.lang.[Cloneable](#), java.io.[Serializable](#))
- java.util.[Observable](#)
- java.security.[Permission](#) (implements java.security.[Guard](#), java.io.[Serializable](#))
 - java.security.[BasicPermission](#) (implements java.io.[Serializable](#))
 - java.util.[PropertyPermission](#)
- java.util.[Random](#) (implements java.io.[Serializable](#))
- java.util.[ResourceBundle](#)
 - java.util.[ListResourceBundle](#)

- java.util.[PropertyResourceBundle](#)

java.util.[Scanner](#) (implements java.util.[Iterator](#)<E>)

java.util.[StringTokenizer](#) (implements java.util.[Enumeration](#)<E>)

java.lang.[Throwable](#) (implements java.io.[Serializable](#))

- java.lang.[Exception](#)
 - java.io.[IOException](#)
 - java.util.[InvalidPropertiesFormatException](#)
 - java.lang.[RuntimeException](#)
 - java.util.[ConcurrentModificationException](#)
 - java.util.[EmptyStackException](#)
 - java.lang.[IllegalArgumentException](#)
 - java.util.[IllegalFormatException](#)
 - java.util.[DuplicateFormatFlagsException](#)
 - java.util.[FormatFlagsConversionMismatchException](#)
 - java.util.[IllegalFormatCodePointException](#)
 - java.util.[IllegalFormatConversionException](#)
 - java.util.[IllegalFormatFlagsException](#)
 - java.util.[IllegalFormatPrecisionException](#)
 - java.util.[IllegalFormatWidthException](#)
 - java.util.[MissingFormatArgumentException](#)
 - java.util.[MissingFormatWidthException](#)
 - java.util.[UnknownFormatConversionException](#)
 - java.util.[UnknownFormatFlagsException](#)
 - java.lang.[IllegalStateException](#)
 - java.util.[FormatterClosedException](#)
 - java.util.[MissingResourceException](#)
 - java.util.[NoSuchElementException](#)
 - java.util.[InputMismatchException](#)
 - java.util.[TooManyListenersException](#)

java.util.[Timer](#)

java.util.[TimerTask](#) (implements java.lang.[Runnable](#))

java.util.[TimeZone](#) (implements java.lang.[Cloneable](#), java.io.[Serializable](#))

- java.util.[SimpleTimeZone](#)

java.util.[UUID](#) (implements java.lang.[Comparable](#)<T>, java.io.[Serializable](#))

Interface Hierarchy

- java.util.[Comparator](#)<T>
- java.util.[Enumeration](#)<E>

- java.util.[EventListener](#)
- java.util.[Formattable](#)
- java.lang.[Iterable](#)<T>
 - java.util.[Collection](#)<E>
 - java.util.[List](#)<E>
 - java.util.[Queue](#)<E>
 - java.util.[Set](#)<E>
 - java.util.[SortedSet](#)<E>
 - java.util.[Iterator](#)<E>
- java.util.[ListIterator](#)<E>
 - java.util.[Map](#)<K,V>
- java.util.[SortedMap](#)<K,V>
 - java.util.[Map.Entry](#)<K,V>
 - java.util.[Observer](#)
 - java.util.[RandomAccess](#)

Enum Hierarchy

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

Introduction

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

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

```
Employ ob [] = new Employ [10];
```

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

Collection Object:

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

The primary advantages of a collections framework are that it:

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

The collections framework consists of:

- **Collection Interfaces** - Represent different types of collections, such as sets, lists and maps. These interfaces form the basis of the framework.
- **General-purpose Implementations** - Primary implementations of the collection interfaces.
- **Legacy Implementations** - The collection classes from earlier releases, Vector and Hashtable, have been retrofitted to implement the collection interfaces.
- **Special-purpose Implementations** - Implementations designed for use in special situations. These implementations display nonstandard performance characteristics, usage restrictions, or behavior.
- **Concurrent Implementations** - Implementations designed for highly concurrent use.
- **Wrapper Implementations** - Add functionality, such as synchronization, to other implementations.
- **Convenience Implementations** - High-performance "mini-implementations" of the collection interfaces.

- **Abstract Implementations** - Partial implementations of the collection interfaces to facilitate custom implementations.
 - **Algorithms** - Static methods that perform useful functions on collections, such as sorting a list.
 - **Infrastructure** - Interfaces that provide essential support for the collection interfaces.
 - **Array Utilities** - Utility functions for arrays of primitives and reference objects. Not, strictly speaking, a part of the Collections Framework, this functionality was added to the Java platform at the same time and relies on some of the same infrastructure.
-

Collection Interfaces

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

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

- Collections that do not support any modification operations (such as `add`, `remove` and `clear`) are referred to as *unmodifiable*. Collections that are not unmodifiable are referred to *modifiable*.
- Collections that additionally guarantee that no change in the `Collection` object will ever be visible are referred to as *immutable*. Collections that are not immutable are referred to as *mutable*.
- Lists that guarantee that their size remains constant even though the elements may change are referred to as *fixed-size*. Lists that are not fixed-size are referred to as *variable-size*.
- Lists that support fast (generally constant time) indexed element access are known as *random access* lists. Lists that do not support fast indexed element access are known as *sequential access* lists. The [RandomAccess](#) marker interface is provided to allow lists to advertise the fact that they support random access. This allows generic algorithms to alter their behavior to provide good performance when applied to either random or sequential access lists.

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

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

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

Collection Implementations

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

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

The general-purpose implementations support all of the *optional operations* in the collection interfaces, and have no restrictions on the elements they may contain. They are unsynchronized, but the `Collections` class contains static factories called *synchronization wrappers* that may be used to add synchronization to any unsynchronized collection. All of the new implementations have *fail-fast iterators*, which detect illegal concurrent modification, and fail quickly and cleanly (rather than behaving erratically).

- The `AbstractCollection`, `AbstractSet`, `AbstractList`, `AbstractSequentialList` and `AbstractMap` classes provide skeletal implementations of the core collection interfaces, to minimize the effort required to implement them. The API documentation

for these classes describes precisely how each method is implemented so the implementer knows which methods should be overridden, given the performance of the "basic operations" of a specific implementation.

- **Set:** A Set represents a group of elements (objects) arranged just like an array. The set will grow dynamically when the elements are stored into it. A set will not allow duplicate elements.
- **List:** Lists are like sets but allow duplicate values to be stored.
- **Queue:** A Queue represents arrangement of elements in FIFO (First In First Out) order. This means that an element that is stored as a first element into the queue will be removed first from the queue.
- **Map:** Maps store elements in the form of key value pairs. If the key is provided its corresponding value can be obtained.

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

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

Iterator Interface: Iterator is an interface that contains methods to retrieve the elements one by one from a collection object. It retrieves 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.

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

```
//HashSet Demo
```

```
import java.util.*;
```

```
class HS
```



```

{ public static void main(String args[])
{ //create a HashSet to store Strings

    HashSet <String> hs = new HashSet<String> ();

    //Store some String elements

    hs.add ("India");
    hs.add ("America");
    hs.add ("Japan");
    hs.add ("China");
    hs.add ("America");

    //view the HashSet

    System.out.println ("HashSet = " + hs);

    //add an Iterator to hs

    Iterator it = hs.iterator ();

    //display element by element using Iterator

    System.out.println ("Elements Using Iterator: ");

    while (it.hasNext() )
    { String s = (String) it.next ();

        System.out.println(s);

    }

}
}

```

Output:

```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac HS.java
D:\JQR>java HS
HashSet = [America, China, Japan, India]
Elements Using Iterator:
America
China
Japan
India
D:\JQR>
```

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

```
class LinkedHashSet<T>
```

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

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

```
class Stack<E>
```

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

Stack Class Methods:

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

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

//pushing, popping, searching elements in a stack

```

import java.io.*;
import java.util.*;

class Stack1
{
    int top=-1,st[]=new int[5];

    void push(int el)
    {
        st[++top]=el;
    }

    int pop()
    {
        return(st[top--]);
    }

    void display()
    {
        System.out.println("\nStack elements from top to bottom\n");
        for(int i=top;i>=0;i--)
            System.out.println(st[i]);
    }

    boolean isFull()
    {
        return(top==5-1);
    }

    boolean isEmpty()
    {

```

```

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

```

```
case 2:if(s.isEmpty())
    System.out.println("\nstack is empty");
    else
    {
        el=s.pop();
        System.out.println("\nDeleted element = "+el);
        }break;
case 3:if(s.isEmpty())
    System.out.println("\nstack is empty");
    else
    s.display();
    break;
case 4:break;
default:System.out.println("\nEnter correct choice");
}
}
}
}
```


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

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

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

```
import java.util.*;
```

```
//Linked List
```

```
class LinkedDemo
```

```

{ public static void main(String args[])
{   LinkedList <String> ll = new LinkedList<String>();

    ll.add ("Asia");

    ll.add ("North America");

    ll.add ("South America");

    ll.add ("Africa");

    ll.addFirst ("Europe");

    ll.add (1,"Australia");

    ll.add (2,"Antarctica");

    System.out.println ("Elements in Linked List is : " + ll);

    System.out.println ("Size of the Linked List is : " + ll.size() );

}
}

```

Output:



```

C:\WINDOWS\system32\cmd.exe
D:\JQR>javac LinkedDemo.java
D:\JQR>java LinkedDemo
Elements in Linked List is : [Europe, Australia, Antarctica, Asia, North America, South America, Africa]
Size of the Linked List is : 7
D:\JQR>

```

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

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

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

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

ArrayList Class Methods:

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

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

```
import java.util.*;

//ArrayList Demo

class ArrayListDemo

{ public static void main(String args[])

{ ArrayList <String> al = new ArrayList<String>();

  al.add ("Asia");

  al.add ("North America");

  al.add ("South America");
```

```

al.add ("Africa");
al.add ("Europe");
al.add (1,"Australia");
al.add (2,"Antarctica");

System.out.print ("Size of the Array List is: " + al.size ());

System.out.print ("\nRetrieving elements in ArrayList using Iterator :");

Iterator it = al.iterator ();

while (it.hasNext () )

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

}

}

```

Output:



```

C:\WINDOWS\system32\cmd.exe
D:\JQR>javac ArrayListDemo.java
D:\JQR>java ArrayListDemo
Size of the Array List is: 7
Retrieving elements in ArrayList using Iterator :Asia Australia Antarctica
North America South America Africa Europe
D:\JQR>

```

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

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

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

Vector Class Methods:

Method	Description
boolean add(element obj)	This method appends the specified element to the end of the Vector. If the element is added successfully then the method returns true.
void add (int position, element obj)	This method inserts the specified element at the specified position in the Vector.
element remove (int position)	This method removes the element at the specified position in the Vector and returns it.
boolean remove (Object obj)	This method removes the first occurrence of the specified element obj from the Vector, if it is present.
void clear ()	This method removes all the elements from the Vector.
element set (int position, element obj)	This method replaces an element at the specified position in the Vector with the specified element obj.
boolean contains (Object obj)	This method returns true if the Vector contains the specified element obj.
element get (int position)	This method returns the element available at the specified position in the Vector.
int size ()	Returns number of elements in the Vector.
int indexOf (Object obj)	This method returns the index of the first occurrence of the specified element in the Vector, or -1 if the Vector does not contain the element.
int lastIndexOf (Object obj)	This method returns the index of the last occurrence of the specified element in the Vector, or -1 if the Vector does not contain the element.
Object[] toArray ()	This method converts the Vector into an array of Object class type. All the elements of the Vector will be stored into the array in the same sequence.
int capacity ()	This method returns the current capacity of the Vector.

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

```
import java.util.*;
```

```
//Vector Demo
```

```
class VectorDemo
```

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

```
{ Vector <Integer> v = new Vector<Integer> ();
```

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

```
//When x[i] is stored into v below, x[i] values are converted into Integer Objects
```

```
//and stored into v. This is auto boxing.
```

```

for (int i = 0; i<x.length; i++)
    v.add(x[i]);

System.out.println ("Getting Vector elements using get () method: ");
for (int i = 0; i<v.size(); i++)
    System.out.print (v.get (i) + "\t");

System.out.println ("\nRetrieving elements in Vector using ListIterator :");

ListIterator lit = v.listIterator ();

while (lit.hasNext () )

    System.out.print (lit.next () + "\t");

System.out.println ("\nRetrieving elements in reverse order using ListIterator :");

while (lit.hasPrevious () )

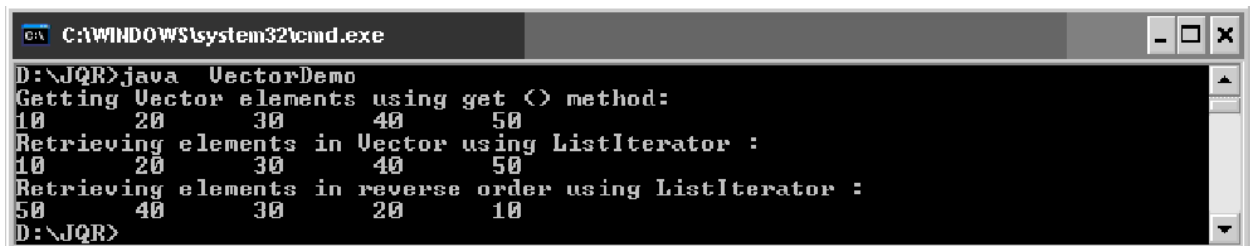
    System.out.print (lit.previous () + "\t");

}

}

```

Output:



```

C:\WINDOWS\system32\cmd.exe
D:\JQR>java VectorDemo
Getting Vector elements using get (<) method:
10 20 30 40 50
Retrieving elements in Vector using ListIterator :
10 20 30 40 50
Retrieving elements in reverse order using ListIterator :
50 40 30 20 10
D:\JQR>

```

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

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

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

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

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

HashMap Class Methods:

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

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

```
//HashMap Demo
```

```
import java.util.*;
```

```
class HashMapDemo
```

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

```
{ HashMap<Integer, String> hm = new HashMap<Integer, String> ();
```

```
hm.put (new Integer (101),"Naresh");
```

```
hm.put (new Integer (102),"Rajesh");
```

```
hm.put (new Integer (103),"Suresh");
```

```
hm.put (new Integer (104),"Mahesh");
```

```
hm.put (new Integer (105),"Ramesh");
```

```
Set<Integer> set = new HashSet<Integer>();
```

```
set = hm.keySet();
```

```

    System.out.println (set);
}
}

```

Output:



```

C:\WINDOWS\system32\cmd.exe
D:\JQR>javac HashMapDemo.java
D:\JQR>java HashMapDemo
[102, 103, 101, 104, 105]
D:\JQR>

```

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

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

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

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

The default initial capacity of this Hashtable will be taken as 11 and the load factor as 0.75. Load factor represents at what level the Hashtable capacity should be doubled. For example, the product of capacity and load factor = $11 * 0.75 = 8.25$. This represents that after storing 8th key-value pair into the Hashtable, its capacity will become 22.

Hashtable Class Methods:

Method	Description
value put(key, value)	This method stores key-value pair into the Hashtable.
value get(Object key)	This method returns the corresponding value when key is given. If the key does not have a value associated with it, then it returns null.
Set<K> keyset()	This method, when applied on a Hashtable converts it into a set where only keys will be stored.
Collection <V> values()	This method, when applied on a Hashtable object returns all the values of the Hashtable into a Collection object.
value remove(Object key)	This method removes the key and corresponding value from the Hashtable.

void clear()	This method removes all the key-value pairs from the Hashtable.
boolean isEmpty()	This method returns true if there are no key-value pairs in the Hashtable.
int size()	This method returns number of key-value pairs in the Hashtable.

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

//Hashtable Demo

import java.util.*;

class HashtableDemo

{ public static void main(String args[])

{

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

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

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

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

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

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

 Enumeration e = ht.keys ();

 while (e.hasMoreElements ())

 {

 Integer i1 = (Integer) e.nextElement ();

 System.out.println (i1 + "\t" + ht.get (i1));

 }

}

}

Output:

```

C:\WINDOWS\system32\cmd.exe
D:\JQR>javac HashtableDemo.java
D:\JQR>java HashtableDemo
105    Ramesh
104    Mahesh
103    Suresh
102    Rajesh
101    Naresh
D:\JQR>

```

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

Arrays Class Methods:

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

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

```

import java.util.*;

//Arrays Demo

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

```



```

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

    Arrays.sort( x );

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

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

}
}

```

Output:

```

C:\WINDOWS\system32\cmd.exe
D:\JQR>javac ArraysDemo.java
D:\JQR>java ArraysDemo
10 20 30 40 50
D:\JQR>

```

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, "delimiter");
```

StringTokenizer Class Methods:

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

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

```
//cutting the String into tokens
```

```
import java.util.*;
```

```
class STDemo
```

```

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

```

Output:



```

C:\WINDOWS\system32\cmd.exe
D:\JQR>javac STDemo.java
D:\JQR>java STDemo
The tokens are:
Java
is
an
OOP
Language
D:\JQR>_

```

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

Calendar Class Methods:

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

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

```
//To display system time and date
```

```
import java.util.*;
```

```
class Cal
```

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

```
{ Calendar cl = Calendar.getInstance ();
```

```
    //Retrieve Date
```

```
    int dd = cl.get (Calendar.DATE);
```

```
    int mm = cl.get (Calendar.MONTH);
```

```
    ++mm;
```

```
    int yy = cl.get (Calendar.YEAR);
```

```
    System.out.println ("Current Date is : " + dd + "-" + mm + "-" + yy );
```

```
    //Retrieve Time
```

```
    int hh = cl.get (Calendar.HOUR);
```

```
    int mi = cl.get (Calendar.MINUTE);
```

```
    int ss = cl.get (Calendar.SECOND);
```

```
    System.out.println ("Current Time is : " + hh + ":" + mi + ":" +ss);
```

```
}
```

```
}
```

Output:



```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac Cal.java
D:\JQR>java Cal
Current Date is : 9-10-2011
Current Time is : 4:31:42
D:\JQR>
```

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

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

DateFormat class Methods:

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

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

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

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

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

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

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

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

//Display System date and time using Date class

```
import java.util.*;
```

```
import java.text.*;
```

```
class MyDate
```

```

{
    public static void main(String args[])
    {
        Date d = new Date ();

        DateFormat fmt = DateFormat.getDateTimeInstance (DateFormat.MEDIUM,
        DateFormat.SHORT, Locale.UK);

        String str = fmt.format (d);

        System.out.println (str);
    }
}

```

Output:



```

C:\WINDOWS\system32\cmd.exe
D:\JQR>javac MyDate.java
D:\JQR>java MyDate
09-Oct-2011 16:45
D:\JQR>

```

Design Goals

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

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

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

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

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