Java Advanced



Java - Data Structures

The data structures provided by the Java utility package are very powerful and perform a wide range of functions. These data structures consist of the following interface and classes −

* Enumeration
* BitSet
* Vector
* Stack
* Dictionary
* Hashtable
* Properties

All these classes are now legacy and Java-2 has introduced a new framework called Collections Framework, which is discussed in the next chapter. −

The Enumeration

The Enumeration interface isn't itself a data structure, but it is very important within the context of other data structures. The Enumeration interface defines a means to retrieve successive elements from a data structure.

For example, Enumeration defines a method called nextElement that is used to get the next element in a data structure that contains multiple elements.

To have more detail about this interface, check [The Enumeration](https://www.tutorialspoint.com/java/java_enumeration_interface.htm).

The BitSet

The BitSet class implements a group of bits or flags that can be set and cleared individually.

This class is very useful in cases where you need to keep up with a set of Boolean values; you just assign a bit to each value and set or clear it as appropriate.

For more details about this class, check [The BitSet](https://www.tutorialspoint.com/java/java_bitset_class.htm).

The Vector

The Vector class is similar to a traditional Java array, except that it can grow as necessary to accommodate new elements.

Like an array, elements of a Vector object can be accessed via an index into the vector.

The nice thing about using the Vector class is that you don't have to worry about setting it to a specific size upon creation; it shrinks and grows automatically when necessary.

For more details about this class, check [The Vector](https://www.tutorialspoint.com/java/java_vector_class.htm).

The Stack

The Stack class implements a last-in-first-out (LIFO) stack of elements.

You can think of a stack literally as a vertical stack of objects; when you add a new element, it gets stacked on top of the others.

When you pull an element off the stack, it comes off the top. In other words, the last element you added to the stack is the first one to come back off.

For more details about this class, check [The Stack](https://www.tutorialspoint.com/java/java_stack_class.htm).

The Dictionary

The Dictionary class is an abstract class that defines a data structure for mapping keys to values.

This is useful in cases where you want to be able to access data via a particular key rather than an integer index.

Since the Dictionary class is abstract, it provides only the framework for a key-mapped data structure rather than a specific implementation.

For more details about this class, check [The Dictionary](https://www.tutorialspoint.com/java/java_dictionary_class.htm).

The Hashtable

The Hashtable class provides a means of organizing data based on some user-defined key structure.

For example, in an address list hash table you could store and sort data based on a key such as ZIP code rather than on a person's name.

The specific meaning of keys with regard to hash tables is totally dependent on the usage of the hash table and the data it contains.

For more detail about this class, check [The Hashtable](https://www.tutorialspoint.com/java/java_hashtable_class.htm).

The Properties

Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String.

The Properties class is used by many other Java classes. For example, it is the type of object returned by System.getProperties( ) when obtaining environmental values.

For more detail about this class, check [The Properties](https://www.tutorialspoint.com/java/java_properties_class.htm).

Java - Collections Framework

Prior to Java 2, Java provided ad hoc classes such as **Dictionary, Vector, Stack,** and **Properties** to store and manipulate groups of objects. Although these classes were quite useful, they lacked a central, unifying theme. Thus, the way that you used Vector was different from the way that you used Properties.

The collections framework was designed to meet several goals, such as −

* The framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hashtables) were to be highly efficient.
* The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
* The framework had to extend and/or adapt a collection easily.

Towards this end, the entire collections framework is designed around a set of standard interfaces. Several standard implementations such as **LinkedList, HashSet,** and **TreeSet**, of these interfaces are provided that you may use as-is and you may also implement your own collection, if you choose.

A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following −

* **Interfaces** − These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
* **Implementations, i.e., Classes** − These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
* **Algorithms** − These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.

In addition to collections, the framework defines several map interfaces and classes. Maps store key/value pairs. Although maps are not *collections* in the proper use of the term, but they are fully integrated with collections.

The Collection Interfaces

The collections framework defines several interfaces. This section provides an overview of each interface −

|  |  |
| --- | --- |
| **Sr.No.** | **Interface & Description** |
| 1 | [**The Collection Interface**](https://www.tutorialspoint.com/java/java_collection_interface.htm)  This enables you to work with groups of objects; it is at the top of the collections hierarchy. |
| 2 | [**The List Interface**](https://www.tutorialspoint.com/java/java_list_interface.htm)  This extends **Collection** and an instance of List stores an ordered collection of elements. |
| 3 | [**The Set**](https://www.tutorialspoint.com/java/java_set_interface.htm)  This extends Collection to handle sets, which must contain unique elements. |
| 4 | [**The SortedSet**](https://www.tutorialspoint.com/java/java_sortedset_interface.htm)  This extends Set to handle sorted sets. |
| 5 | [**The Map**](https://www.tutorialspoint.com/java/java_map_interface.htm)  This maps unique keys to values. |
| 6 | [**The Map.Entry**](https://www.tutorialspoint.com/java/java_mapentry_interface.htm)  This describes an element (a key/value pair) in a map. This is an inner class of Map. |
| 7 | [**The SortedMap**](https://www.tutorialspoint.com/java/java_sortedmap_interface.htm)  This extends Map so that the keys are maintained in an ascending order. |
| 8 | [**The Enumeration**](https://www.tutorialspoint.com/java/java_enumeration_interface.htm)  This is legacy interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. This legacy interface has been superceded by Iterator. |

The Collection Classes

Java provides a set of standard collection classes that implement Collection interfaces. Some of the classes provide full implementations that can be used as-is and others are abstract class, providing skeletal implementations that are used as starting points for creating concrete collections.

The standard collection classes are summarized in the following table −

|  |  |
| --- | --- |
| **Sr.No.** | **Class & Description** |
| 1 | **AbstractCollection**  Implements most of the Collection interface. |
| 2 | **AbstractList**  Extends AbstractCollection and implements most of the List interface. |
| 3 | **AbstractSequentialList**  Extends AbstractList for use by a collection that uses sequential rather than random access of its elements. |
| 4 | [**LinkedList**](https://www.tutorialspoint.com/java/java_linkedlist_class.htm)  Implements a linked list by extending AbstractSequentialList. |
| 5 | [**ArrayList**](https://www.tutorialspoint.com/java/java_arraylist_class.htm)  Implements a dynamic array by extending AbstractList. |
| 6 | **AbstractSet**  Extends AbstractCollection and implements most of the Set interface. |
| 7 | [**HashSet**](https://www.tutorialspoint.com/java/java_hashset_class.htm)  Extends AbstractSet for use with a hash table. |
| 8 | [**LinkedHashSet**](https://www.tutorialspoint.com/java/java_linkedhashset_class.htm)  Extends HashSet to allow insertion-order iterations. |
| 9 | [**TreeSet**](https://www.tutorialspoint.com/java/java_treeset_class.htm)  Implements a set stored in a tree. Extends AbstractSet. |
| 10 | **AbstractMap**  Implements most of the Map interface. |
| 11 | [**HashMap**](https://www.tutorialspoint.com/java/java_hashmap_class.htm)  Extends AbstractMap to use a hash table. |
| 12 | [**TreeMap**](https://www.tutorialspoint.com/java/java_treemap_class.htm)  Extends AbstractMap to use a tree. |
| 13 | [**WeakHashMap**](https://www.tutorialspoint.com/java/java_weakhashmap_class.htm)  Extends AbstractMap to use a hash table with weak keys. |
| 14 | [**LinkedHashMap**](https://www.tutorialspoint.com/java/java_linkedhashmap_class.htm)  Extends HashMap to allow insertion-order iterations. |
| 15 | [**IdentityHashMap**](https://www.tutorialspoint.com/java/java_identityhashmap_class.htm)  Extends AbstractMap and uses reference equality when comparing documents. |

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 following legacy classes defined by java.util have been discussed in the previous chapter −

|  |  |
| --- | --- |
| **Sr.No.** | **Class & Description** |
| 1 | [**Vector**](https://www.tutorialspoint.com/java/java_vector_class.htm)  This implements a dynamic array. It is similar to ArrayList, but with some differences. |
| 2 | [**Stack**](https://www.tutorialspoint.com/java/java_stack_class.htm)  Stack is a subclass of Vector that implements a standard last-in, first-out stack. |
| 3 | [**Dictionary**](https://www.tutorialspoint.com/java/java_dictionary_class.htm)  Dictionary is an abstract class that represents a key/value storage repository and operates much like Map. |
| 4 | [**Hashtable**](https://www.tutorialspoint.com/java/java_hashtable_class.htm)  Hashtable was part of the original java.util and is a concrete implementation of a Dictionary. |
| 5 | [**Properties**](https://www.tutorialspoint.com/java/java_properties_class.htm)  Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String. |
| 6 | [**BitSet**](https://www.tutorialspoint.com/java/java_bitset_class.htm)  A BitSet class creates a special type of array that holds bit values. This array can increase in size as needed. |

The Collection Algorithms

The collections framework defines several algorithms that can be applied to collections and maps. These algorithms are defined as static methods within the Collections class.

Several of the methods can throw a **ClassCastException**, which occurs when an attempt is made to compare incompatible types, or an **UnsupportedOperationException**, which occurs when an attempt is made to modify an unmodifiable collection.

Collections define three static variables: EMPTY\_SET, EMPTY\_LIST, and EMPTY\_MAP. All are immutable.

|  |  |
| --- | --- |
| **Sr.No.** | **Algorithm & Description** |
| 1 | [**The Collection Algorithms**](https://www.tutorialspoint.com/java/java_collection_algorithms.htm)  Here is a list of all the algorithm implementation. |

How to Use an Iterator ?

Often, you will want to cycle through the elements in a collection. For example, you might want to display each element.

The easiest way to do this is to employ an iterator, which is an object that implements either the Iterator or the ListIterator interface.

Iterator enables you to cycle through a collection, obtaining or removing elements. ListIterator extends Iterator to allow bidirectional traversal of a list and the modification of elements.

|  |  |
| --- | --- |
| **Sr.No.** | **Iterator Method & Description** |
| 1 | [**Using Java Iterator**](https://www.tutorialspoint.com/java/java_using_iterator.htm)  Here is a list of all the methods with examples provided by Iterator and ListIterator interfaces. |

How to Use a Comparator ?

Both TreeSet and TreeMap store elements in a sorted order. However, it is the comparator that defines precisely what *sorted order* means.

This interface lets us sort a given collection any number of different ways. Also this interface can be used to sort any instances of any class (even classes we cannot modify).

|  |  |
| --- | --- |
| **Sr.No.** | **Iterator Method & Description** |
| 1 | [**Using Java Comparator**](https://www.tutorialspoint.com/java/java_using_comparator.htm)  Here is a list of all the methods with examples provided by Comparator Interface. |

Summary

The Java collections framework gives the programmer access to prepackaged data structures as well as to algorithms for manipulating them.

A collection is an object that can hold references to other objects. The collection interfaces declare the operations that can be performed on each type of collection.

The classes and interfaces of the collections framework are in package java.util.

Java - Generics

It would be nice if we could write a single sort method that could sort the elements in an Integer array, a String array, or an array of any type that supports ordering.

Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.

Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

Generic Methods

You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define Generic Methods −

* All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type ( < E > in the next example).
* Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
* The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
* A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

Example

Following example illustrates how we can print an array of different type using a single Generic method −

public class GenericMethodTest {

// generic method printArray

public static < E > void printArray( E[] inputArray ) {

// Display array elements

for(E element : inputArray) {

System.out.printf("%s ", element);

}

System.out.println();

}

public static void main(String args[]) {

// Create arrays of Integer, Double and Character

Integer[] intArray = { 1, 2, 3, 4, 5 };

Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };

Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

System.out.println("Array integerArray contains:");

printArray(intArray); // pass an Integer array

System.out.println("\nArray doubleArray contains:");

printArray(doubleArray); // pass a Double array

System.out.println("\nArray characterArray contains:");

printArray(charArray); // pass a Character array

}

}

This will produce the following result −

Output

Array integerArray contains:

1 2 3 4 5

Array doubleArray contains:

1.1 2.2 3.3 4.4

Array characterArray contains:

H E L L O

Bounded Type Parameters

There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.

To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound.

Example

Following example illustrates how extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces). This example is Generic method to return the largest of three Comparable objects −

public class MaximumTest {

// determines the largest of three Comparable objects

public static <T extends Comparable<T>> T maximum(T x, T y, T z) {

T max = x; // assume x is initially the largest

if(y.compareTo(max) > 0) {

max = y; // y is the largest so far

}

if(z.compareTo(max) > 0) {

max = z; // z is the largest now

}

return max; // returns the largest object

}

public static void main(String args[]) {

System.out.printf("Max of %d, %d and %d is %d\n\n",

3, 4, 5, maximum( 3, 4, 5 ));

System.out.printf("Max of %.1f,%.1f and %.1f is %.1f\n\n",

6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ));

System.out.printf("Max of %s, %s and %s is %s\n","pear",

"apple", "orange", maximum("pear", "apple", "orange"));

}

}

This will produce the following result −

Output

Max of 3, 4 and 5 is 5

Max of 6.6,8.8 and 7.7 is 8.8

Max of pear, apple and orange is pear

Generic Classes

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas. These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

Example

Following example illustrates how we can define a generic class −

public class Box<T> {

private T t;

public void add(T t) {

this.t = t;

}

public T get() {

return t;

}

public static void main(String[] args) {

Box<Integer> integerBox = new Box<Integer>();

Box<String> stringBox = new Box<String>();

integerBox.add(new Integer(10));

stringBox.add(new String("Hello World"));

System.out.printf("Integer Value :%d\n\n", integerBox.get());

System.out.printf("String Value :%s\n", stringBox.get());

}

}

This will produce the following result −

Output

Integer Value :10

String Value :Hello World

Java - Serialization

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.

The ObjectOutputStream class contains many write methods for writing various data types, but one method in particular stands out −

public final void writeObject(Object x) throws IOException

The above method serializes an Object and sends it to the output stream. Similarly, the ObjectInputStream class contains the following method for deserializing an object −

public final Object readObject() throws IOException, ClassNotFoundException

This method retrieves the next Object out of the stream and deserializes it. The return value is Object, so you will need to cast it to its appropriate data type.

To demonstrate how serialization works in Java, I am going to use the Employee class that we discussed early on in the book. Suppose that we have the following Employee class, which implements the Serializable interface −

Example

public class Employee implements java.io.Serializable {

public String name;

public String address;

public transient int SSN;

public int number;

public void mailCheck() {

System.out.println("Mailing a check to " + name + " " + address);

}

}

Notice that for a class to be serialized successfully, two conditions must be met −

* The class must implement the java.io.Serializable interface.
* All of the fields in the class must be serializable. If a field is not serializable, it must be marked **transient**.

If you are curious to know if a Java Standard Class is serializable or not, check the documentation for the class. The test is simple: If the class implements java.io.Serializable, then it is serializable; otherwise, it's not.

Serializing an Object

The ObjectOutputStream class is used to serialize an Object. The following SerializeDemo program instantiates an Employee object and serializes it to a file.

When the program is done executing, a file named employee.ser is created. The program does not generate any output, but study the code and try to determine what the program is doing.

**Note** − When serializing an object to a file, the standard convention in Java is to give the file a **.ser** extension.

Example

import java.io.\*;

public class SerializeDemo {

public static void main(String [] args) {

Employee e = new Employee();

e.name = "Reyan Ali";

e.address = "Phokka Kuan, Ambehta Peer";

e.SSN = 11122333;

e.number = 101;

try {

FileOutputStream fileOut =

new FileOutputStream("/tmp/employee.ser");

ObjectOutputStream out = new ObjectOutputStream(fileOut);

out.writeObject(e);

out.close();

fileOut.close();

System.out.printf("Serialized data is saved in /tmp/employee.ser");

}catch(IOException i) {

i.printStackTrace();

}

}

}

Deserializing an Object

The following DeserializeDemo program deserializes the Employee object created in the SerializeDemo program. Study the program and try to determine its output −

Example

import java.io.\*;

public class DeserializeDemo {

public static void main(String [] args) {

Employee e = null;

try {

FileInputStream fileIn = new FileInputStream("/tmp/employee.ser");

ObjectInputStream in = new ObjectInputStream(fileIn);

e = (Employee) in.readObject();

in.close();

fileIn.close();

}catch(IOException i) {

i.printStackTrace();

return;

}catch(ClassNotFoundException c) {

System.out.println("Employee class not found");

c.printStackTrace();

return;

}

System.out.println("Deserialized Employee...");

System.out.println("Name: " + e.name);

System.out.println("Address: " + e.address);

System.out.println("SSN: " + e.SSN);

System.out.println("Number: " + e.number);

}

}

This will produce the following result −

Output

Deserialized Employee...

Name: Reyan Ali

Address:Phokka Kuan, Ambehta Peer

SSN: 0

Number:101

Here are following important points to be noted −

* The try/catch block tries to catch a ClassNotFoundException, which is declared by the readObject() method. For a JVM to be able to deserialize an object, it must be able to find the bytecode for the class. If the JVM can't find a class during the deserialization of an object, it throws a ClassNotFoundException.
* Notice that the return value of readObject() is cast to an Employee reference.
* The value of the SSN field was 11122333 when the object was serialized, but because the field is transient, this value was not sent to the output stream. The SSN field of the deserialized Employee object is 0.

Java - Multithreading

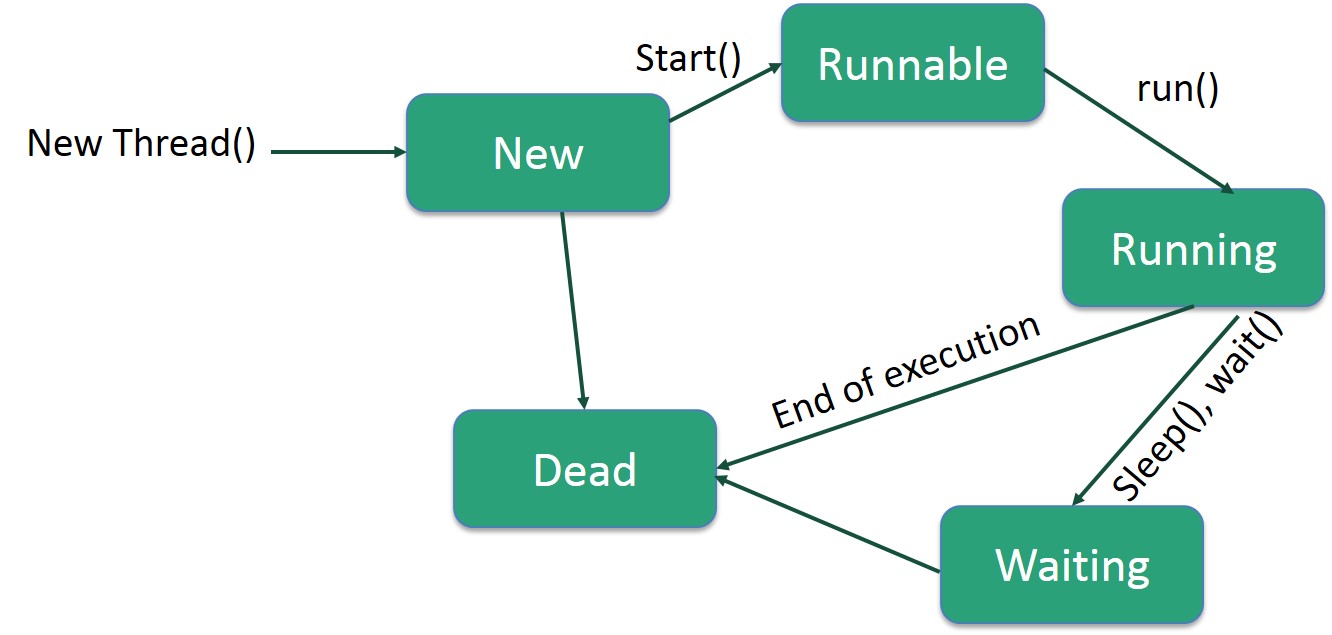
Java is a *multi-threaded programming language* which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

By definition, multitasking is when multiple processes share common processing resources such as a CPU. Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.

Life Cycle of a Thread

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.



Following are the stages of the life cycle −

* **New** − A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a **born thread**.
* **Runnable** − After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
* **Waiting** − Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
* **Timed Waiting** − A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
* **Terminated (Dead)** − A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread Priorities

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java thread priorities are in the range between MIN\_PRIORITY (a constant of 1) and MAX\_PRIORITY (a constant of 10). By default, every thread is given priority NORM\_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.

Create a Thread by Implementing a Runnable Interface

If your class is intended to be executed as a thread then you can achieve this by implementing a **Runnable** interface. You will need to follow three basic steps −

Step 1

As a first step, you need to implement a run() method provided by a **Runnable** interface. This method provides an entry point for the thread and you will put your complete business logic inside this method. Following is a simple syntax of the run() method −

public void run( )

Step 2

As a second step, you will instantiate a **Thread** object using the following constructor −

Thread(Runnable threadObj, String threadName);

Where, *threadObj* is an instance of a class that implements the **Runnable** interface and **threadName** is the name given to the new thread.

Step 3

Once a Thread object is created, you can start it by calling **start()** method, which executes a call to run( ) method. Following is a simple syntax of start() method −

void start();

Example

Here is an example that creates a new thread and starts running it −

class RunnableDemo implements Runnable {

private Thread t;

private String threadName;

RunnableDemo( String name) {

threadName = name;

System.out.println("Creating " + threadName );

}

public void run() {

System.out.println("Running " + threadName );

try {

for(int i = 4; i > 0; i--) {

System.out.println("Thread: " + threadName + ", " + i);

// Let the thread sleep for a while.

Thread.sleep(50);

}

}catch (InterruptedException e) {

System.out.println("Thread " + threadName + " interrupted.");

}

System.out.println("Thread " + threadName + " exiting.");

}

public void start () {

System.out.println("Starting " + threadName );

if (t == null) {

t = new Thread (this, threadName);

t.start ();

}

}

}

public class TestThread {

public static void main(String args[]) {

RunnableDemo R1 = new RunnableDemo( "Thread-1");

R1.start();

RunnableDemo R2 = new RunnableDemo( "Thread-2");

R2.start();

}

}

This will produce the following result −

Output

Creating Thread-1

Starting Thread-1

Creating Thread-2

Starting Thread-2

Running Thread-1

Thread: Thread-1, 4

Running Thread-2

Thread: Thread-2, 4

Thread: Thread-1, 3

Thread: Thread-2, 3

Thread: Thread-1, 2

Thread: Thread-2, 2

Thread: Thread-1, 1

Thread: Thread-2, 1

Thread Thread-1 exiting.

Thread Thread-2 exiting.

Create a Thread by Extending a Thread Class

The second way to create a thread is to create a new class that extends **Thread** class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

Step 1

You will need to override **run( )** method available in Thread class. This method provides an entry point for the thread and you will put your complete business logic inside this method. Following is a simple syntax of run() method −

public void run( )

Step 2

Once Thread object is created, you can start it by calling **start()** method, which executes a call to run( ) method. Following is a simple syntax of start() method −

void start( );

Example

Here is the preceding program rewritten to extend the Thread −

class ThreadDemo extends Thread {

private Thread t;

private String threadName;

ThreadDemo( String name) {

threadName = name;

System.out.println("Creating " + threadName );

}

public void run() {

System.out.println("Running " + threadName );

try {

for(int i = 4; i > 0; i--) {

System.out.println("Thread: " + threadName + ", " + i);

// Let the thread sleep for a while.

Thread.sleep(50);

}

}catch (InterruptedException e) {

System.out.println("Thread " + threadName + " interrupted.");

}

System.out.println("Thread " + threadName + " exiting.");

}

public void start () {

System.out.println("Starting " + threadName );

if (t == null) {

t = new Thread (this, threadName);

t.start ();

}

}

}

public class TestThread {

public static void main(String args[]) {

ThreadDemo T1 = new ThreadDemo( "Thread-1");

T1.start();

ThreadDemo T2 = new ThreadDemo( "Thread-2");

T2.start();

}

}

This will produce the following result −

Output

Creating Thread-1

Starting Thread-1

Creating Thread-2

Starting Thread-2

Running Thread-1

Thread: Thread-1, 4

Running Thread-2

Thread: Thread-2, 4

Thread: Thread-1, 3

Thread: Thread-2, 3

Thread: Thread-1, 2

Thread: Thread-2, 2

Thread: Thread-1, 1

Thread: Thread-2, 1

Thread Thread-1 exiting.

Thread Thread-2 exiting.

Thread Methods

Following is the list of important methods available in the Thread class.

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **public void start()**  Starts the thread in a separate path of execution, then invokes the run() method on this Thread object. |
| 2 | **public void run()**  If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object. |
| 3 | **public final void setName(String name)**  Changes the name of the Thread object. There is also a getName() method for retrieving the name. |
| 4 | **public final void setPriority(int priority)**  Sets the priority of this Thread object. The possible values are between 1 and 10. |
| 5 | **public final void setDaemon(boolean on)**  A parameter of true denotes this Thread as a daemon thread. |
| 6 | **public final void join(long millisec)**  The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes. |
| 7 | **public void interrupt()**  Interrupts this thread, causing it to continue execution if it was blocked for any reason. |
| 8 | **public final boolean isAlive()**  Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion. |

The previous methods are invoked on a particular Thread object. The following methods in the Thread class are static. Invoking one of the static methods performs the operation on the currently running thread.

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **public static void yield()**  Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled. |
| 2 | **public static void sleep(long millisec)**  Causes the currently running thread to block for at least the specified number of milliseconds. |
| 3 | **public static boolean holdsLock(Object x)**  Returns true if the current thread holds the lock on the given Object. |
| 4 | **public static Thread currentThread()**  Returns a reference to the currently running thread, which is the thread that invokes this method. |
| 5 | **public static void dumpStack()**  Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application. |

Example

The following ThreadClassDemo program demonstrates some of these methods of the Thread class. Consider a class **DisplayMessage** which implements **Runnable** −

// File Name : DisplayMessage.java

// Create a thread to implement Runnable

public class DisplayMessage implements Runnable {

private String message;

public DisplayMessage(String message) {

this.message = message;

}

public void run() {

while(true) {

System.out.println(message);

}

}

}

Following is another class which extends the Thread class −

// File Name : GuessANumber.java

// Create a thread to extentd Thread

public class GuessANumber extends Thread {

private int number;

public GuessANumber(int number) {

this.number = number;

}

public void run() {

int counter = 0;

int guess = 0;

do {

guess = (int) (Math.random() \* 100 + 1);

System.out.println(this.getName() + " guesses " + guess);

counter++;

} while(guess != number);

System.out.println("\*\* Correct!" + this.getName() + "in" + counter + "guesses.\*\*");

}

}

Following is the main program, which makes use of the above-defined classes −

// File Name : ThreadClassDemo.java

public class ThreadClassDemo {

public static void main(String [] args) {

Runnable hello = new DisplayMessage("Hello");

Thread thread1 = new Thread(hello);

thread1.setDaemon(true);

thread1.setName("hello");

System.out.println("Starting hello thread...");

thread1.start();

Runnable bye = new DisplayMessage("Goodbye");

Thread thread2 = new Thread(bye);

thread2.setPriority(Thread.MIN\_PRIORITY);

thread2.setDaemon(true);

System.out.println("Starting goodbye thread...");

thread2.start();

System.out.println("Starting thread3...");

Thread thread3 = new GuessANumber(27);

thread3.start();

try {

thread3.join();

}catch(InterruptedException e) {

System.out.println("Thread interrupted.");

}

System.out.println("Starting thread4...");

Thread thread4 = new GuessANumber(75);

thread4.start();

System.out.println("main() is ending...");

}

}

This will produce the following result. You can try this example again and again and you will get a different result every time.

Output

Starting hello thread...

Starting goodbye thread...

Hello

Hello

Hello

Hello

Hello

Hello

Goodbye

Goodbye

Goodbye

Goodbye

Goodbye

.......

Major Java Multithreading Concepts

While doing Multithreading programming in Java, you would need to have the following concepts very handy −

* [What is thread synchronization?](https://www.tutorialspoint.com/java/java_thread_synchronization.htm)
* [Handling interthread communication](https://www.tutorialspoint.com/java/java_thread_communication.htm)
* [Handling thread deadlock](https://www.tutorialspoint.com/java/java_thread_deadlock.htm)
* [Major thread operations](https://www.tutorialspoint.com/java/java_thread_control.htm)

Java - Applet Basics

An **applet** is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.

There are some important differences between an applet and a standalone Java application, including the following −

* An applet is a Java class that extends the java.applet.Applet class.
* A main() method is not invoked on an applet, and an applet class will not define main().
* Applets are designed to be embedded within an HTML page.
* When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.
* A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.
* The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.
* Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.
* Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file.

Life Cycle of an Applet

Four methods in the Applet class gives you the framework on which you build any serious applet −

* **init** − This method is intended for whatever initialization is needed for your applet. It is called after the param tags inside the applet tag have been processed.
* **start** − This method is automatically called after the browser calls the init method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.
* **stop** − This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.
* **destroy** − This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you should not normally leave resources behind after a user leaves the page that contains the applet.
* **paint** − Invoked immediately after the start() method, and also any time the applet needs to repaint itself in the browser. The paint() method is actually inherited from the java.awt.

A "Hello, World" Applet

Following is a simple applet named HelloWorldApplet.java −

import java.applet.\*;

import java.awt.\*;

public class HelloWorldApplet extends Applet {

public void paint (Graphics g) {

g.drawString ("Hello World", 25, 50);

}

}

These import statements bring the classes into the scope of our applet class −

* java.applet.Applet
* java.awt.Graphics

Without those import statements, the Java compiler would not recognize the classes Applet and Graphics, which the applet class refers to.

The Applet Class

Every applet is an extension of the *java.applet.Applet class*. The base Applet class provides methods that a derived Applet class may call to obtain information and services from the browser context.

These include methods that do the following −

* Get applet parameters
* Get the network location of the HTML file that contains the applet
* Get the network location of the applet class directory
* Print a status message in the browser
* Fetch an image
* Fetch an audio clip
* Play an audio clip
* Resize the applet

Additionally, the Applet class provides an interface by which the viewer or browser obtains information about the applet and controls the applet's execution. The viewer may −

* Request information about the author, version, and copyright of the applet
* Request a description of the parameters the applet recognizes
* Initialize the applet
* Destroy the applet
* Start the applet's execution
* Stop the applet's execution

The Applet class provides default implementations of each of these methods. Those implementations may be overridden as necessary.

The "Hello, World" applet is complete as it stands. The only method overridden is the paint method.

Invoking an Applet

An applet may be invoked by embedding directives in an HTML file and viewing the file through an applet viewer or Java-enabled browser.

The <applet> tag is the basis for embedding an applet in an HTML file. Following is an example that invokes the "Hello, World" applet −

<html>

<title>The Hello, World Applet</title>

<hr>

<applet code = "HelloWorldApplet.class" width = "320" height = "120">

If your browser was Java-enabled, a "Hello, World"

message would appear here.

</applet>

<hr>

</html>

**Note** − You can refer to [HTML Applet Tag](https://www.tutorialspoint.com/html/html_applet_tag.htm" \o "HTML Applet Tag" \t "_blank) to understand more about calling applet from HTML.

The code attribute of the <applet> tag is required. It specifies the Applet class to run. Width and height are also required to specify the initial size of the panel in which an applet runs. The applet directive must be closed with an </applet> tag.

If an applet takes parameters, values may be passed for the parameters by adding <param> tags between <applet> and </applet>. The browser ignores text and other tags between the applet tags.

Non-Java-enabled browsers do not process <applet> and </applet>. Therefore, anything that appears between the tags, not related to the applet, is visible in non-Java-enabled browsers.

The viewer or browser looks for the compiled Java code at the location of the document. To specify otherwise, use the codebase attribute of the <applet> tag as shown −

<applet codebase = "https://amrood.com/applets" code = "HelloWorldApplet.class"

width = "320" height = "120">

If an applet resides in a package other than the default, the holding package must be specified in the code attribute using the period character (.) to separate package/class components. For example −

<applet = "mypackage.subpackage.TestApplet.class"

width = "320" height = "120">

Getting Applet Parameters

The following example demonstrates how to make an applet respond to setup parameters specified in the document. This applet displays a checkerboard pattern of black and a second color.

The second color and the size of each square may be specified as parameters to the applet within the document.

CheckerApplet gets its parameters in the init() method. It may also get its parameters in the paint() method. However, getting the values and saving the settings once at the start of the applet, instead of at every refresh, is convenient and efficient.

The applet viewer or browser calls the init() method of each applet it runs. The viewer calls init() once, immediately after loading the applet. (Applet.init() is implemented to do nothing.) Override the default implementation to insert custom initialization code.

The Applet.getParameter() method fetches a parameter given the parameter's name (the value of a parameter is always a string). If the value is numeric or other non-character data, the string must be parsed.

The following is a skeleton of CheckerApplet.java −

import java.applet.\*;

import java.awt.\*;

public class CheckerApplet extends Applet {

int squareSize = 50; // initialized to default size

public void init() {}

private void parseSquareSize (String param) {}

private Color parseColor (String param) {}

public void paint (Graphics g) {}

}

Here are CheckerApplet's init() and private parseSquareSize() methods −

public void init () {

String squareSizeParam = getParameter ("squareSize");

parseSquareSize (squareSizeParam);

String colorParam = getParameter ("color");

Color fg = parseColor (colorParam);

setBackground (Color.black);

setForeground (fg);

}

private void parseSquareSize (String param) {

if (param == null) return;

try {

squareSize = Integer.parseInt (param);

}catch (Exception e) {

// Let default value remain

}

}

The applet calls parseSquareSize() to parse the squareSize parameter. parseSquareSize() calls the library method Integer.parseInt(), which parses a string and returns an integer. Integer.parseInt() throws an exception whenever its argument is invalid.

Therefore, parseSquareSize() catches exceptions, rather than allowing the applet to fail on bad input.

The applet calls parseColor() to parse the color parameter into a Color value. parseColor() does a series of string comparisons to match the parameter value to the name of a predefined color. You need to implement these methods to make this applet work.

Specifying Applet Parameters

The following is an example of an HTML file with a CheckerApplet embedded in it. The HTML file specifies both parameters to the applet by means of the <param> tag.

<html>

<title>Checkerboard Applet</title>

<hr>

<applet code = "CheckerApplet.class" width = "480" height = "320">

<param name = "color" value = "blue">

<param name = "squaresize" value = "30">

</applet>

<hr>

</html>

**Note** − Parameter names are not case sensitive.

Application Conversion to Applets

It is easy to convert a graphical Java application (that is, an application that uses the AWT and that you can start with the Java program launcher) into an applet that you can embed in a web page.

Following are the specific steps for converting an application to an applet.

* Make an HTML page with the appropriate tag to load the applet code.
* Supply a subclass of the JApplet class. Make this class public. Otherwise, the applet cannot be loaded.
* Eliminate the main method in the application. Do not construct a frame window for the application. Your application will be displayed inside the browser.
* Move any initialization code from the frame window constructor to the init method of the applet. You don't need to explicitly construct the applet object. The browser instantiates it for you and calls the init method.
* Remove the call to setSize; for applets, sizing is done with the width and height parameters in the HTML file.
* Remove the call to setDefaultCloseOperation. An applet cannot be closed; it terminates when the browser exits.
* If the application calls setTitle, eliminate the call to the method. Applets cannot have title bars. (You can, of course, title the web page itself, using the HTML title tag.)
* Don't call setVisible(true). The applet is displayed automatically.

Event Handling

Applets inherit a group of event-handling methods from the Container class. The Container class defines several methods, such as processKeyEvent and processMouseEvent, for handling particular types of events, and then one catch-all method called processEvent.

In order to react to an event, an applet must override the appropriate event-specific method.

import java.awt.event.MouseListener;

import java.awt.event.MouseEvent;

import java.applet.Applet;

import java.awt.Graphics;

public class ExampleEventHandling extends Applet implements MouseListener {

StringBuffer strBuffer;

public void init() {

addMouseListener(this);

strBuffer = new StringBuffer();

addItem("initializing the apple ");

}

public void start() {

addItem("starting the applet ");

}

public void stop() {

addItem("stopping the applet ");

}

public void destroy() {

addItem("unloading the applet");

}

void addItem(String word) {

System.out.println(word);

strBuffer.append(word);

repaint();

}

public void paint(Graphics g) {

// Draw a Rectangle around the applet's display area.

g.drawRect(0, 0,

getWidth() - 1,

getHeight() - 1);

// display the string inside the rectangle.

g.drawString(strBuffer.toString(), 10, 20);

}

public void mouseEntered(MouseEvent event) {

}

public void mouseExited(MouseEvent event) {

}

public void mousePressed(MouseEvent event) {

}

public void mouseReleased(MouseEvent event) {

}

public void mouseClicked(MouseEvent event) {

addItem("mouse clicked! ");

}

}

Now, let us call this applet as follows −

<html>

<title>Event Handling</title>

<hr>

<applet code = "ExampleEventHandling.class"

width = "300" height = "300">

</applet>

<hr>

</html>

Initially, the applet will display "initializing the applet. Starting the applet." Then once you click inside the rectangle, "mouse clicked" will be displayed as well.

Displaying Images

An applet can display images of the format GIF, JPEG, BMP, and others. To display an image within the applet, you use the drawImage() method found in the java.awt.Graphics class.

Following is an example illustrating all the steps to show images −

import java.applet.\*;

import java.awt.\*;

import java.net.\*;

public class ImageDemo extends Applet {

private Image image;

private AppletContext context;

public void init() {

context = this.getAppletContext();

String imageURL = this.getParameter("image");

if(imageURL == null) {

imageURL = "java.jpg";

}

try {

URL url = new URL(this.getDocumentBase(), imageURL);

image = context.getImage(url);

}catch(MalformedURLException e) {

e.printStackTrace();

// Display in browser status bar

context.showStatus("Could not load image!");

}

}

public void paint(Graphics g) {

context.showStatus("Displaying image");

g.drawImage(image, 0, 0, 200, 84, null);

g.drawString("www.javalicense.com", 35, 100);

}

}

Now, let us call this applet as follows −

<html>

<title>The ImageDemo applet</title>

<hr>

<applet code = "ImageDemo.class" width = "300" height = "200">

<param name = "image" value = "java.jpg">

</applet>

<hr>

</html>

Playing Audio

An applet can play an audio file represented by the AudioClip interface in the java.applet package. The AudioClip interface has three methods, including −

* **public void play()** − Plays the audio clip one time, from the beginning.
* **public void loop()** − Causes the audio clip to replay continually.
* **public void stop()** − Stops playing the audio clip.

To obtain an AudioClip object, you must invoke the getAudioClip() method of the Applet class. The getAudioClip() method returns immediately, whether or not the URL resolves to an actual audio file. The audio file is not downloaded until an attempt is made to play the audio clip.

Following is an example illustrating all the steps to play an audio −

import java.applet.\*;

import java.awt.\*;

import java.net.\*;

public class AudioDemo extends Applet {

private AudioClip clip;

private AppletContext context;

public void init() {

context = this.getAppletContext();

String audioURL = this.getParameter("audio");

if(audioURL == null) {

audioURL = "default.au";

}

try {

URL url = new URL(this.getDocumentBase(), audioURL);

clip = context.getAudioClip(url);

}catch(MalformedURLException e) {

e.printStackTrace();

context.showStatus("Could not load audio file!");

}

}

public void start() {

if(clip != null) {

clip.loop();

}

}

public void stop() {

if(clip != null) {

clip.stop();

}

}

}

Now, let us call this applet as follows −

<html>

<title>The ImageDemo applet</title>

<hr>

<applet code = "ImageDemo.class" width = "0" height = "0">

<param name = "audio" value = "test.wav">

</applet>

<hr>

</html>

You can use test.wav on your PC to test the above example.

Java - Documentation Comments

The Java language supports three types of comments −

|  |  |
| --- | --- |
| **Sr.No.** | **Comment & Description** |
| 1 | **/\* text \*/**  The compiler ignores everything from /\* to \*/. |
| 2 | **//text**  The compiler ignores everything from // to the end of the line. |
| 3 | **/\*\* documentation \*/**  This is a documentation comment and in general its called **doc comment**. The **JDK javadoc** tool uses *doc comments* when preparing automatically generated documentation. |

This chapter is all about explaining Javadoc. We will see how we can make use of Javadoc to generate useful documentation for Java code.

What is Javadoc?

Javadoc is a tool which comes with JDK and it is used for generating Java code documentation in HTML format from Java source code, which requires documentation in a predefined format.

Following is a simple example where the lines inside /\*….\*/ are Java multi-line comments. Similarly, the line which preceeds // is Java single-line comment.

Example

/\*\*

\* The HelloWorld program implements an application that

\* simply displays "Hello World!" to the standard output.

\*

\* @author Zara Ali

\* @version 1.0

\* @since 2014-03-31

\*/

public class HelloWorld {

public static void main(String[] args) {

/\* Prints Hello, World! on standard output.

System.out.println("Hello World!");

}

}

You can include required HTML tags inside the description part. For instance, the following example makes use of <h1>....</h1> for heading and <p> has been used for creating paragraph break −

Example

/\*\*

\* <h1>Hello, World!</h1>

\* The HelloWorld program implements an application that

\* simply displays "Hello World!" to the standard output.

\* <p>

\* Giving proper comments in your program makes it more

\* user friendly and it is assumed as a high quality code.

\*

\*

\* @author Zara Ali

\* @version 1.0

\* @since 2014-03-31

\*/

public class HelloWorld {

public static void main(String[] args) {

/\* Prints Hello, World! on standard output.

System.out.println("Hello World!");

}

}

The javadoc Tags

The javadoc tool recognizes the following tags −

|  |  |  |
| --- | --- | --- |
| **Tag** | **Description** | **Syntax** |
| @author | Adds the author of a class. | @author name-text |
| {@code} | Displays text in code font without interpreting the text as HTML markup or nested javadoc tags. | {@code text} |
| {@docRoot} | Represents the relative path to the generated document's root directory from any generated page. | {@docRoot} |
| @deprecated | Adds a comment indicating that this API should no longer be used. | @deprecated deprecatedtext |
| @exception | Adds a **Throws** subheading to the generated documentation, with the classname and description text. | @exception class-name description |
| {@inheritDoc} | Inherits a comment from the **nearest** inheritable class or implementable interface. | Inherits a comment from the immediate surperclass. |
| {@link} | Inserts an in-line link with the visible text label that points to the documentation for the specified package, class, or member name of a referenced class. | {@link package.class#member label} |
| {@linkplain} | Identical to {@link}, except the link's label is displayed in plain text than code font. | {@linkplain package.class#member label} |
| @param | Adds a parameter with the specified parameter-name followed by the specified description to the "Parameters" section. | @param parameter-name description |
| @return | Adds a "Returns" section with the description text. | @return description |
| @see | Adds a "See Also" heading with a link or text entry that points to reference. | @see reference |
| @serial | Used in the doc comment for a default serializable field. | @serial field-description | include | exclude |
| @serialData | Documents the data written by the writeObject( ) or writeExternal( ) methods. | @serialData data-description |
| @serialField | Documents an ObjectStreamField component. | @serialField field-name field-type field-description |
| @since | Adds a "Since" heading with the specified since-text to the generated documentation. | @since release |
| @throws | The @throws and @exception tags are synonyms. | @throws class-name description |
| {@value} | When {@value} is used in the doc comment of a static field, it displays the value of that constant. | {@value package.class#field} |
| @version | Adds a "Version" subheading with the specified version-text to the generated docs when the -version option is used. | @version version-text |

Example

Following program uses few of the important tags available for documentation comments. You can make use of other tags based on your requirements.

The documentation about the AddNum class will be produced in HTML file AddNum.html but at the same time a master file with a name index.html will also be created.

import java.io.\*;

/\*\*

\* <h1>Add Two Numbers!</h1>

\* The AddNum program implements an application that

\* simply adds two given integer numbers and Prints

\* the output on the screen.

\* <p>

\* <b>Note:</b> Giving proper comments in your program makes it more

\* user friendly and it is assumed as a high quality code.

\*

\* @author Zara Ali

\* @version 1.0

\* @since 2014-03-31

\*/

public class AddNum {

/\*\*

\* This method is used to add two integers. This is

\* a the simplest form of a class method, just to

\* show the usage of various javadoc Tags.

\* @param numA This is the first paramter to addNum method

\* @param numB This is the second parameter to addNum method

\* @return int This returns sum of numA and numB.

\*/

public int addNum(int numA, int numB) {

return numA + numB;

}

/\*\*

\* This is the main method which makes use of addNum method.

\* @param args Unused.

\* @return Nothing.

\* @exception IOException On input error.

\* @see IOException

\*/

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

AddNum obj = new AddNum();

int sum = obj.addNum(10, 20);

System.out.println("Sum of 10 and 20 is :" + sum);

}

}

Now, process the above AddNum.java file using javadoc utility as follows −

$ javadoc AddNum.java

Loading source file AddNum.java...

Constructing Javadoc information...

Standard Doclet version 1.7.0\_51

Building tree for all the packages and classes...

Generating /AddNum.html...

AddNum.java:36: warning - @return tag cannot be used in method with void return type.

Generating /package-frame.html...

Generating /package-summary.html...

Generating /package-tree.html...

Generating /constant-values.html...

Building index for all the packages and classes...

Generating /overview-tree.html...

Generating /index-all.html...

Generating /deprecated-list.html...

Building index for all classes...

Generating /allclasses-frame.html...

Generating /allclasses-noframe.html...

Generating /index.html...

Generating /help-doc.html...

1 warning