Module -5
Object Oriented Programming with Java

The Networking with Java.net
Networking fundamentals,
Networking classes and Interfaces, The InetAddress class, The Socket Class ,The URL class, The URL Connection Class,The Http URL Connection Class.

The collections Framework: Collections Overview, Recent Changes to Collections, The Collection Interfaces, The Collection Classes, Accessing a collection Via an Iterator, Storing User Defined Classes in Collections, The Random Access Interface, Working With Maps, Comparators, The Collection Algorithms, Why Generic Collections?, The legacy Classes and Interfaces, Parting Thoughts on Collections.

At the core of Java's networking support is the concept of a socket.
A socket identifies an endpoint in a network.
The socket paradigm was part of the 4:2BSD Berkeley UNIX release in the early 1980s

Sockets are at the foundation of modern networking because a socket allows a single computer to serve many different clients at once, as well as to serve many different types of information. This is accomplished through the use of a port, which is a numbered socket on a particular machine. A server process is said to "listen" to a port until a client connects to it.

A server is allowed to accept multiple clients connected to the same port number, although each session is unique.
To manage multiple client connections, a server process must be multithreaded or have some other means of multiplexing the simultaneous I/O.

Socket communication takes place via a protocol. Internet Protocol (IP) is a low-level routing protocol that breaks data into small packets and sends them to an address across a network, which does not guarantee to deliver said packets to the destination.

Transmission Control Protocol (TCP) is a higher-level protocol that manages to robustly string together these packets, sorting and retransmitting them as necessary to reliably transmit data.
A third protocol, User Datagram Protocol (UDP), sits next to TCP and can be used directly to support fast, connectionless, unreliable transport of packets.

Once a connection has been established, a higher-level protocol ensues, which is dependent on which port you are using. TCP/IP reserves the lower 1,024 ports for specific 599 protocols. Many of these will seem familiar to you if you have spent any time surfing the Internet. Port number 21 is for FTP: 23 is for Telnet; 25 is for e-mail; 43 is for who is; 79 is for finger; 80 is for HTTP; 119 is for net news- and the list goes on. It is up to each protocol to determine how a client should interact with the port.

A key component of the Internet is the address. Every computer on the Internet has one. An Internet address is a number that uniquely identifies each computer on the Net. Originally, all Internet addresses consisted of 32-bit values, organized as four 8-bit values. This address type was specified by IPv4 (Internet Protocol, version 4).

However, a new addressing scheme, called IPv6 (Internet Protocol, version 6) has come into play. IPv6 uses a 128-bit value to represent an address, organized into eight 16-bit chunks. Although there are several reasons for and advantages to IPv6, the main one is that it supports a much larger address space than does IPv4.

The Networking Classes and Interfaces
Java supports both the TCP and UDP protocol families. TCP is used for reliable stream-based I/O across the network. UDP supports a simpler, hence faster, point-to-point datagram-oriented model. The classes contained in the java.net package are shown here:

Authenticator	Inet6Address	ServerSocket
CacheRequest	InetAddress	Socket
CacheResponse	InetSocketAddress	SocketAddress
ContentHandler	InterfaceAddress (Added by Java SE 6.)	SocketImpl
CookieHandler	JarURLConnection	SocketPermission
CookieManager (Added by Java SE 6.)	MulticastSocket	URI
DatagramPacket	NetPermission	URL

The java.net package's interfaces are listed here:
ContentHandlerFactory DatagramSocketime
SocketOptions
Factory

ContentHandlerFactory	DatagramSocketImplFactory	SocketOptions
CookiePolicy (Added by Java SE 6.)	FileNameMap	URLStreamHandlerFactory
CookieStore (Added by Java SE 6.)	SocketImplFactory	

InetAddress

The InetAddress class is used to encapsulate both the numerical IP address and the domain name for that address, we interact with this class by using the name of an IP host, which is more convenient and understandable than its IP address. The InetAddress class hides the number inside. InetAddress can handle both IPv4 and IPv6 addresses.

Factory Methods

The InetAddress class has no visible constructors. To create an InetAddress object, you have to use one of the available factory methods. Factory methods are merely a convention whereby static methods in a class return an instance of that class. This is done in lieu of overloading a constructor with various parameter lists when having unique méthod names makes the results much clearer. Three commonly used InetAddress factory methods are shown here:

static InetAddress getLocalHost() throws UnknownHostException
static InetAddress getByName(String hostName) throws UnknownHostException
static InetAddress[] getAllByName(String hostName) throws UnknownHostException

The getLocalHost() method simply returns the InetAddress object that represents the local host.

The getByName() method returns an InetAddress for a host name passed to it. If these methods are unable to resolve the host name, they throw an UnknownHostException.

On the Internet, it is common for a single name to be used to represent several machines.
In the world of web servers, this is one way to provide some degree of scaling.L

The getAllByName() factory method returns an array of InetAddresses that represent all of the addresses that a particular name resolves to. It will also throw an UnknownHostException

The following example prints the addresses and names of the local machine and two well-known Internet web sites:

```
// Demonstrate InetAddress. import java.net.*;
class InetAddress Test
public static void main(String args[]) throws
UnknownHostException { InetAddress Address =
InetAddress.getLocalHost(); System.out.println(Address);
Address = InetAddress.getByName("osborne.com"):
System.out.println(Address);
InetAddress SW] =””:
for (int i=0; i<SW.length; i++)
System.out.println(SW[i]);
```

Instance Methods

The InetAddress class has several other methods, which can be used on the objects returned by the methods just discussed. Here are some of the more commonly used methods:

boolean equals(Object *other*)	Returns **true** if this object has the same Internet address as *other*.
byte[ ] getAddress( )	Returns a byte array that represents the object's IP address in network byte order.
String getHostAddress( )	Returns a string that represents the host address associated with the **InetAddress** object.
String getHostName( )	Returns a string that represents the host name associated with the **InetAddress** object.
boolean isMulticastAddress( )	Returns **true** if this address is a multicast address. Otherwise, it returns **false**.
String toString( )	Returns a string that lists the host name and the IP address for convenience.

Inet4Address and Inet6Address

Beginning with version 1.4, Java has included support for IPv6 addresses. Because of this, two subclasses of InetAddress were created: Inet4Address and Inet6Address. Inet4Address represents a traditional-style IPv4 address. Inet6Address encapsulates a new-style IPv6 address. Because they are subclasses of InetAddress, an InetAddress reference can refer to either. This is one way that Java was able to add IPv6 functionality without breaking existing code or adding many more classes. For the most part, you can simply use InetAddress when working with IP addresses because it can accommodate both styles.

TCP/IP Client Sockets

TCP/IP sockets are used to implement reliable, bidirectional, persistent, point-to-point, stream-based connections between hosts on the Internet. A socket can be used to connect Java's I/O system to other programs that may reside either on the local machine or on any other machine on the Internet. There are two kinds of TCP sockets in Java. One is for servers, and the other is for clients. The ServerSocket class is designed to be a "listener," which waits for clients to connect before doing anything. Thus, ServerSocket is for servers. The Socket class is for clients. It is designed to connect to server sockets and initiate protocol exchanges. Because client sockets are the most commonly used by Java applications, they are examined here.

Internet is not about the older protocols such as whois, finger, and FTP. It is about WWW, the World Wide Web. The Web is a loose collection of higher-level protocols and file formats, all unified in a web browser. One of the most important aspects of the Web is that Tim Berners-Lee devised a scaleable way to locate all of the resources of the Net. Once you can reliably name anything and everything, it becomes a very powerful paradigm. The Uniform Resource Locator (URL)

URLConnection

URLConnection is a general-purpose class for accessing the attributes of a remote resource. Once you make a connection to a remote server, you can use URLConnection to inspect the properties of the remote object before actually transporting it locally. These attributes are exposed by the HTTP protocol specification and, as such, only make sense for URL objects that are using the HTTP protocol.

HttpURLConnection

Java provides a subclass of URLConnection that provides support for HTTP connections. This class is called HttpURLConnection. You obtain an HttpURLConnection in the same way just shown, by calling openConnection() on a URL object, but you must cast the result to HttpURLConnection. (Of course, you must make sure that you are actually opening an HTTP connection.) Once you have obtained a reference to an HttpURLConnection object, you can use any of the methods inherited from URLConnection. You can also use any of the several methods defined by HttpURL Connection. Here is a sampling:

The URI Class

A relatively recent addition to Java is the URI class, which encapsulates a Uniform Resource Identifier (URI). URIs are similar to URLs. In fact, URLs constitute a subset of URIS. A URI represents a standard way to identify a resource. A URL also describes how to access the resource

Cookies

The java.net package includes classes and interfaces that help manage cookies and can be used to create a stateful (as opposed to stateless) HTTP session. The classes are CookieHandler, CookieManager, and HttpCookie. The interfaces are Cookie Policy and CookieStore. All but CookieHandler was added by Java SE 6. (CookieHandler was added by JDK 5.)

TCP/IP Server Sockets

As mentioned earlier, Java has a different socket class that must be used for creating server applications. The ServerSocket class is used to create servers that listen for either local or remote client programs to connect to them on published ports. ServerSockets are quite different from normal Sockets. When you create a ServerSocket, it will register itself with the system as having an interest in client connections. The constructors for ServerSocket reflect the port number that you want to accept connections on and, optionally, how long you want the queue for said port to be. The queue length tells the system how many client connections it can leave pending before it should simply refuse connections. The default is 50. The constructors might throw an IOException under adverse conditions. Here are three of its constructors:

ServerSocket(int *port*) throws IOException	Creates server socket on the specified port with a queue length of 50.
ServerSocket(int *port*, int *maxQueue*) throws IOException	Creates a server socket on the specified port with a maximum queue length of *maxQueue*.
ServerSocket(int *port*, int *maxQueue*, InetAddress *localAddress*) throws IOException	Creates a server socket on the specified port with a maximum queue length of *maxQueue*. On a multihomed host, *localAddress* specifies the IP address to which this socket binds.

Datagrams

Datagrams are bundles of information passed between machines. They are somewhat like a hard throw from a well-trained but blindfolded catcher to the third baseman. Once the datagram has been released to its intended target, there is no assurance that it will arrive or even that someone will be there to catch it. Likewise, when the datagram is received, there is no assurance that it hasn't been damaged in transit or that whoever sent it is still there to receive a response.

Java implements datagrams on top of the UDP protocol by using two classes: the DatagramPacket object is the data container, while the DatagramSocket is the mechanism used to send or receive the DatagramPackets. Each is examined here.

DatagramPacket
DatagramPacket defines several constructors. Four are shown here:

DatagramPacket(byte data[ ], int size)
DatagramPacket(byte data[ ], int offset, int size)
DatagramPacket(byte data[ ], int size, InetAddress ipAddress, int port)
DatagramPacket(byte data[], int offset, int size,InetAddress ipAddress, int port

DatagramPacket defines several methods, including those shown here, that give access to the address and port number of a packet, as well as the raw data and its length. In general, the get methods are used on packets that are received and the set methods are used on packets that will be sent.

InetAddress getAddress( )	Returns the address of the source (for datagrams being received) or destination (for datagrams being sent).
byte[ ] getData( )	Returns the byte array of data contained in the datagram. Mostly used to retrieve data from the datagram after it has been received.
int getLength( )	Returns the length of the valid data contained in the byte array that would be returned from the getData( ) method. This may not equal the length of the whole byte array.

InetAddress getAddress( )	Returns the address of the source (for datagrams being received) or destination (for datagrams being sent).
byte[ ] getData( )	Returns the byte array of data contained in the datagram. Mostly used to retrieve data from the datagram after it has been received.
int getLength( )	Returns the length of the valid data contained in the byte array that would be returned from the getData( ) method. This may not equal the length of the whole byte array.
int getOffset( )	Returns the starting index of the data.
int getPort( )	Returns the port number.
void setAddress(InetAddress ipAddress)	Sets the address to which a packet will be sent. The address is specified by ipAddress.
void setData(byte[ ] data)	Sets the data to data, the offset to zero, and the length to number of bytes in data.
void setData(byte[ ] data, int idx, int size)	Sets the data to data, the offset to idx, and the length to size.
void setLength(int size)	Sets the length of the packet to size.
void setPort(int port)	Sets the port to port.

The Collections Framework

The java.util package also contains one of Java's most powerful subsystems: The Collections Framework. The Collections
Framework is a sophisticated hierarchy of interfaces and classes that provide state-of-the-art technology for managing groups of objects. It merits close attention by all programmers. Because java.util contains a wide array of functionality, it is quite large. Here is a list of its classes:

Collections Overview

Prior to the Collections Framework, 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. The way that you used Vector was different from the way that you used Properties. for example. Also, this early, ad hoc approach was not designed to be easily extended or adapted. Collections are an answer to these (and other) problems.

The Collections Framework was designed to meet several goals. First, the framework had to be high-performance.
The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) are highly efficient. You seldom, if ever, need to code one of these "data engines" manually. Second, the framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability. Third, extending and/or adapting a collection had to be easy. Toward this end, the entire Collections Framework is built upon a set of standard interfaces.

Recent Changes to Collections

Recently, the Collections Framework underwent a fundamental change that significantly increased its power and streamlined its use. The changes were caused by the addition of generics, autoboxing/unboxing, and the for-each style for loop, by JDK 5. Although we will be revisiting these topics throughout the course of this chapter, a brief overview is warranted now.

Generics Fundamentally Change the Collections Framework

The addition of generics caused a significant change to the
Collections Framework because the entire Collections Framework has been reengineered for it. All collections are now generic, and many of the methods that operate on collections take generic type parameters. Simply put, the addition of generics has affected every part of the Collections Framework

In addition to the collection interfaces, collections also use the Comparator, RandomAccess, Iterator, and Listiterator interfaces, which are described in depth later in this chapter.
Briefly, Comparator defines how two objects are compared;
Iterator and Listiterator enumerate the objects within a collection.
By implementing Random Access, a list indicates that it supports efficient, random access to its elements.

The Collection Interface
The Collection interface is the foundation upon which the Collections Framework is built because it must be implemented by any class that defines a collection. Collection is a generic interface that has this declaration:

interface Collection<E>

Here, E specifies the type of objects that the collection will hold. Collection extends the Iterable interface.

Class	Description
AbstractCollection	Implements most of the Collection interface.
AbstractList	Extends AbstractCollection and implements most of the List interface.
AbstractQueue	Extends AbstractCollection and implements parts of the Queue interface.
AbstractSequentialList	Extends AbstractList for use by a collection that uses sequential rather than random access of its elements.
LinkedList	Implements a linked list by extending AbstractSequentialList.
ArrayList	Implements a dynamic array by extending AbstractList.
ArrayDeque	Implements a dynamic double-ended queue by extending AbstractCollection and implementing the Deque interface. (Added by Java SE 6.)
AbstractSet	Extends AbstractCollection and implements most of the Set interface.
EnumSet	Extends AbstractSet for use with enum elements.
HashSet	Extends AbstractSet for use with a hash table.
LinkedHashSet	Extends HashSet to allow insertion-order iterations.
PriorityQueue	Extends AbstractQueue to support a priority-based queue.
TreeSet	Implements a set stored in a tree. Extends AbstractSet.

The ArrayList Class

The ArrayList class extends AbstractList and implements the List interface. ArrayList is a generic class that has this declaration:

class ArrayList<E>

Here, E specifies the type of objects that the list will hold. ArrayList supports dynamic arrays that can grow as needed. In Java, standard arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold. But, sometimes, you may not know until run time precisely how large an array you need. To handle this situation, the Collections Framework defines ArrayList. In essênce, an ArrayList is a variable-length array of object references. That is, an ArrayList can dynamically increase or decrease in size. Array

The Linked List Class

The Linked List class extends AbstractSequentialList and implements the List, Deque, and Queue interfaces. It provides a linked-list data structure. LinkedList is a generic class that has this declaration:

class LinkedList<E>

Here, E specifies the type of objects that the list will hold. LinkedList has the two constructors shown here:

LinkedList()

LinkedList(Collection<? extends E> c)

Accessing a Collection via an Iterator
Often, you will want to cycle through the elements in a collection.
For example, you might want to display each element. One 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

Method	Description
boolean hasNext( )	Returns true if there are more elements. Otherwise, returns false.
E next( )	Returns the next element. Throws NoSuchElementException if there is not a next element.
void remove( )	Removes the current element. Throws IllegalStateException if an attempt is made to call remove( ) that is not preceded by a call to next( ).

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. They are summarized in Table 17-14. As explained earlier, beginning with JDK 5 all of the algorithms have been retrofitted for generics. Although the generic syntax might seem a bit intimidating at first, the algorithms are as simple to use as they were before generics. It's just that now, they are type safe.

Comparators
Both TreeSet and TreeMap store elements in sorted order. However, it is the comparator that defines precisely what "sorted order" means. By default, these classes store their elements by using what Java refers to as "natural ordering," which is usually the ordering that you would expect (A before B, 1 before 2, and so forth). If you want to order elements a different way, then specify a Comparator when you construct the set or map. Doing so gives you the ability to govern precisely how elements are stored within sorted collections and maps.
Comparator is a generic interface that has this declaration:
interface Comparator<T>

Class
The IdentityHashMap
IdentityHashMap extends AbstractMap and implements the Map interface. It is similar to HashMap except that it uses reference equality when comparing elements. IdentityHashMap is a generic class that has this declaration:
class IdentityHashMap<K, V>
Here, K specifies the type of key, and V specifies the type of value. The API documentation explicitly states that IdentityHashMap is not for general use.

Class
The LinkedHashMap
LinkedHashMap extends HashMap. It maintains a linked list of the entries in the map, in the order in which they were inserted. This allows insertion-order iteration over the map. That is, when iterating through a collection-view of a Linked HashMap, the elements will be returned in the order in which they were inserted. You can also create a LinkedHashMap that returns its elements in the order in which they were last accessed. LinkedHashMap is a generic class that has this declaration:
class LinkedHashMap<K, V>
Here, K specifies the type of keys, and V specifies the type of values.
LinkedHashMap defines the following constructors:
LinkedHashMap()
LinkedHashMap(Map<? extends K, ? extends V> m)
LinkedHashMap(int capacity)
LinkedHashMap(int capacity, float fillRatio) LinkedHashMap(int capacity, float fillRatio, boolean Order)

Using an Iterator
Before you can access a collection through an iterator, you must obtain one. Each of the collection classes provides an iterator() method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time. In general, to use an iterator to cycle through the contents of a collection, follow these steps:
Obtain an iterator to the start of the collection by calling the collection's iterator()
method.
Set up a loop that makes a call to hasNext(). Have the loop iterate as long as hasNext()
returns true.
Within the loop, obtain each element by calling next().

Storing User-Defined Classes in Collections

For the sake of simplicity, the foregoing examples have stored built-in objects, such as String or Integer, in a collection. Of course, collections are not limited to the storage of built-in objects. Quite the contrary. The power of collections is that they can store any type of object, including objects of classes that you create. For example, consider the following example that uses a LinkedList to store mailing addresses:

The RandomAccess Interface

The RandomAccess interface contains no members. However, by implementing this interface, a collection signals that it supports efficient random access to its elements. Although a collection might support random access, it might not do so efficiently. By checking for the RandomAccess interface, client code can determine at run time whether a collection is suitable for certain types of random access operations-especially as they apply to large collections. (You can use instanceof to determine if a class implements an interface.) Random Access is implemented by ArrayList and by the legacy Vector class, among others.

Working with Maps

A map is an object that stores associations between keys and values, or key/value pairs. Given a key, you can find its value. Both keys and values are objects. The keys must be unique, but the values may be duplicated. Some maps can accept a null key and null values, others cannot.

There is one key point about maps that is important to mention at the outset: they don't implement the Iterable interface. This means that you cannot cycle through a map using a for-each style for loop. Furthermore, you can't obtain an iterator to a map. However, as you will soon see, you can obtain a collection-view of a map, which does allow the use of either the for loop or an iterator.

The Map Interfaces
Because the map interfaces define the character and nature of maps, this discussion of maps begins with them. The following interfaces support maps:

Interface	Description
Map	Maps unique keys to values.
Map.Entry	Describes an element (a key/value pair) in a map. This is an inner class of Map.
NavigableMap	Extends SortedMap to handle the retrieval of entries based on closest-match searches. (Added by Java SE 6.)
SortedMap	Extends Map so that the keys are maintained in ascending order.

The Map Interface

The Map interface maps unique keys to values. A key is an object that you use to retrieve a value at a later date. Given a key and a value, you can store the value in a Map object. After the value is stored, you can retrieve it by using its key. Map is generic and is declared as shown here:

interface Map<K, V>

Here, K specifies the type of keys, and V specifies the type of values.

The methods declared by Map are summarized in Table 17-10. Several methods throw a ClassCastException when an object is incompatible with the elements in a map

Method	Description
void clear( )	Removes all key/value pairs from the invoking map.
boolean containsKey(Object k)	Returns true if the invoking map contains k as a key. Otherwise, returns false.
boolean containsValue(Object v)	Returns true if the map contains v as a value. Otherwise, returns false.
Set<Map.Entry<K, V>> entrySet( )	Returns a Set that contains the entries in the map. The set contains objects of type Map.Entry. Thus, this method provides a set-view of the invoking map.
boolean equals(Object obj)	Returns true if obj is a Map and contains the same entries. Otherwise, returns false.
V get(Object k)	Returns the value associated with the key k. Returns null if the key is not found.
int hashCode( )	Returns the hash code for the invoking map.
boolean isEmpty( )	Returns true if the invoking map is empty. Otherwise, returns false.
Set<K> keySet( )	Returns a Set that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.
V put(K k, V v)	Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are k and v, respectively. Returns null if the key did not already exist. Otherwise, the previous value linked to the key is returned.
void putAll(Map<? extends K, ? extends V> m)	Puts all the entries from m into this map.
V remove(Object k)	Removes the entry whose key equals k.
int size( )	Returns the number of key/value pairs in the map.
Collection<V> values( )	Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.

The SortedMap Interface

The SortedMap interface extends Map. It ensures that the entries are maintained in ascending order based on the keys. Sorted Map is generic and is declared as shown here:

interface SortedMap<K, V>

Here, K specifies the type of keys, and V specifies the type of values.

The methods declared by Sorted Map are summarized in Table 17-11. Several methods throw a NoSuch ElementException when no items are in the invoking map. A ClassCastException is thrown when an object is incompatible with the elements in a map. A NullPointerException is thrown if an attempt is made to use a null object when null is not allowed in the map.

Sorted maps allow very efficient manipulations of submaps (in other words, subsets of a map). To obtain a submap, use headMap(), tailMap(), or subMap(). To get the first key in the set, call firstKey().
To get the last key, use lastKey().

Method	Description
Comparator<? super K> comparator( )	Returns the invoking sorted map's comparator. If natural ordering is used for the invoking map, null is returned.
K firstKey( )	Returns the first key in the invoking map.
SortedMap<K, V> headMap(K end)	Returns a sorted map for those map entries with keys that are less than end.
K lastKey( )	Returns the last key in the invoking map.
SortedMap<K, V> subMap(K start, K end)	Returns a map containing those entries with keys that are greater than or equal to start and less than end.
SortedMap<K, V> tailMap(K start)	Returns a map containing those entries with keys that are greater than or equal to start.

The NavigableMap Interface

The NavigableMap interface was added by Java SE 6. It extends SortedMap and declares the behavior of a map that supports the retrieval of entries based on the closest match to a given key or keys. NavigableMap is a generic interface that has this declaration:

interface NavigableMap<K,V>

Here, K specifies the type of the keys, and V specifies the type of the values associated with the keys. In addition to the methods that it inherits from Sorted Map, NavigableMap

The Map Classes

Several classes provide implementations of the map interfaces. The classes that can be used for maps are summarized here:

Class	Description
AbstractMap	Implements most of the Map interface.
EnumMap	Extends AbstractMap for use with enum keys.
HashMap	Extends AbstractMap to use a hash table.
TreeMap	Extends AbstractMap to use a tree.
WeakHashMap	Extends AbstractMap to use a hash table with weak keys.
LinkedHashMap	Extends HashMap to allow insertion-order iterations.
IdentityHashMap	Extends AbstractMap and uses reference equality when comparing documents.

The HashMap Class
The HashMap class extends AbstractMap and implements the Map interface. It uses a hash table to store the map. This allows the execution time of get() and put() to remain constant even for large sets. HashMap is a generic class that has this declaration:
class HashMap<K, V>
Here, K specifies the type of keys, and V specifies the type of values.
The following constructors are defined:
HashMap()
HashMap(Map<? extends K, ? extends V> m) HashMap(int capacity)
HashMap(int capacity, float fillRatio)

The first form constructs an empty tree map that will be sorted by using the natural order of its keys. The second form constructs an empty tree-based map that will be sorted by using the Comparator comp. (Comparators are discussed later in this chapter.) The third form initializes a tree map with the entries from m, which will be sorted by using the natural order of the keys. The fourth form initializes a tree map with the entries from sm, which will be sorted in the same order as sm.

The TreeMap Class
The TreeMap class extends AbstractMap and implements the NavigableMap interface. It creates maps stored in a tree structure. A TreeMap provides an efficient means of storing key/value pairs in sorted order and allows rapid retrieval. You should note that, unlike a hash map, a tree map guarantees that its elements will be sorted in ascending key order.
TreeMap is a generic class that has this declaration: class TreeMap<K, V>
Here, K specifies the type of keys, and V specifies the type of values.
The following TreeMap constructors are defined:
TreeMap()
TreeMap(Comparator<? super K> comp) TreeMap(Map<? extends K, ? extends V> m) TreeMap(SortedMap<K, ? extends V> sm)