UNIT III I/O STREAMS, STRINGS AND COLLECTIONS 9
Character Streams - Stream I/O - Serialization – Files – String Class - String handling - java. util. Collections - Collections Frameworks - Generic Classes and Methods.

Character Streams

Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file –

Example

```java
import java.io.*;
public class CopyFile {

  public static void main(String args[]) throws IOException {
    FileReader in = null;
    FileWriter out = null;

    try {
      in = new FileReader("input.txt");
      out = new FileWriter("output.txt");

      int c;
      while ((c = in.read()) != -1) {
        out.write(c);
      }
    }finally {
      if (in != null) {
        in.close();
      }
      if (out != null) {
        out.close();
      }
    }
  }
}
```

Now let's have a file **input.txt** with the following content –

```
This is test for copy file.
```

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following –

```
$javac CopyFile.java
$java CopyFile
```

The Character Streams
While the byte stream classes provide sufficient functionality to handle any type of I/O operation, they cannot work directly with Unicode characters. Since one of the main purposes of Java is to support the "write once, run anywhere" philosophy, it was necessary to include direct I/O support for characters. At the top of the character stream hierarchies are the **Reader** and **Writer** abstract classes.

Reader
Reader is an abstract class that defines Java's model of streaming character input. It implements the **AutoCloseable**, **Closeable**, and **Readable** interfaces. All of the methods in this class (except for **markSupported( )**) will throw an **IOException** on error conditions. Table 3.2 provides a synopsis of the methods in **Reader**.

Writer
Writer is an abstract class that defines streaming character output. It implements the **AutoCloseable**, **Closeable**, **Flushable**, and **Appendable** interfaces. All of the methods in this class throw an **IOException** in the case of errors. Table 3.3 shows a synopsis of the methods in **Writer**.

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( )** and **println( )**
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

Table 3.1 Character Stream I/O classes in java.io

Method	Description
abstract void close( )	Closes the input source. Further read attempts will generate an **IOException**.
void mark(int *numChars*)	Places a mark at the current point in the input stream that will remain valid until *numChars* characters are read.
boolean markSupported( )	Returns **true** if **mark( )**/**reset( )** are supported on this stream.
int read( )	Returns an integer representation of the next available character from the invoking input stream. –1 is returned when the end of the file is encountered.
int read(char *buffer*[ ])	Attempts to read up to *buffer.length* characters into *buffer* and returns the actual number of characters that were successfully read. –1 is returned when the end of the file is encountered.
int read(CharBuffer *buffer*)	Attempts to read characters into *buffer* and returns the actual number of characters that were successfully read. –1 is returned when the end of the file is encountered.
abstract int read(char *buffer*[ ], int *offset*, int *numChars*)	Attempts to read up to *numChars* characters into *buffer* starting at *buffer*[*offset*], returning the number of characters successfully read. –1 is returned when the end of the file is encountered.
boolean ready( )	Returns **true** if the next input request will not wait. Otherwise, it returns **false**.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long *numChars*)	Skips over *numChars* characters of input, returning the number of characters actually skipped.

Table 3.2 Methods Defined by Reader

Method	Description
Writer append(char *ch*)	Appends *ch* to the end of the invoking output stream. Returns a reference to the invoking stream.
Writer append(CharSequence *chars*)	Appends *chars* to the end of the invoking output stream. Returns a reference to the invoking stream.
Writer append(CharSequence *chars*, int *begin*, int *end*)	Appends the subrange of *chars* specified by *begin* and *end*–1 to the end of the invoking output stream. Returns a reference to the invoking stream.
abstract void close( )	Closes the output stream. Further write attempts will generate an **IOException**.
abstract void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(int *ch*)	Writes a single character to the invoking output stream. Note that the parameter is an **int**, which allows you to call **write** with an expression without having to cast it back to **char**. However, only the low-order 16 bits are written.
void write(char *buffer*[ ])	Writes a complete array of characters to the invoking output stream.
abstract void write(char *buffer*[ ], int *offset*, int *numChars*)	Writes a subrange of *numChars* characters from the array *buffer*, beginning at *buffer*[*offset*] to the invoking output stream.
void write(String *str*)	Writes *str* to the invoking output stream.
void write(String *str*, int *offset*, int *numChars*)	Writes a subrange of *numChars* characters from the string *str*, beginning at the specified *offset*.

Table 3.3 Methods Defined by Writer

FileReader

The **FileReader** class creates a **Reader** that you can use to read the contents of a file. Two commonly used constructors are shown here:

- FileReader(String *filePath*)
- FileReader(File *fileObj*)

Either can throw a **FileNotFoundException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file. The following example shows how to read lines from a file and display them on the standard output device. It reads its own source file, which must be in the current directory.

```
// Demonstrate FileReader.
// This program uses try-with-resources. It requires JDK 7 or later.
import java.io.*;
class FileReaderDemo {
public static void main(String args[]) {
try ( FileReader fr = new FileReader("FileReaderDemo.java") )
{
int c;
// Read and display the file.
while((c = fr.read()) != -1) System.out.print((char) c);
} catch(IOException e) {
System.out.println("I/O Error: " + e);
}
}
}
```

FileWriter

FileWriter creates a **Writer** that you can use to write to a file. Four commonly used constructors are shown here:

- FileWriter(String *filePath*)
- FileWriter(String *filePath*, boolean *append*)
- FileWriter(File *fileObj*)
- FileWriter(File *fileObj*, boolean *append*)

They can all throw an **IOException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file. *If append* is **true**, then output is appended to the end of the file.

Creation of a **FileWriter** is not dependent on the file already existing. **FileWriter** will create the file before opening it for output when you create the object. In the case where you attempt to open a read-only file, an **IOException** will be thrown.
The following example is a character stream version of an example shown earlier when **FileOutputStream** was discussed. This version creates a sample buffer of characters by first making a **String** and then using the **getChars( )** method to extract the character array equivalent. It then creates three files. The first, **file1.txt**, will contain every other character from the sample. The second, **file2.txt**, will contain the entire set of characters. Finally, the third, **file3.txt**, will contain only the last quarter.

```
// Demonstrate FileWriter.
// This program uses try-with-resources. It requires JDK 7 or later.
import java.io.*;
class FileWriterDemo {
public static void main(String args[]) throws IOException {
String source = "Now is the time for all good men\n"
+ " to come to the aid of their country\n"
+ " and pay their due taxes.";
char buffer[] = new char[source.length()];
source.getChars(0, source.length(), buffer, 0);
try ( FileWriter f0 = new FileWriter("file1.txt");
FileWriter f1 = new FileWriter("file2.txt");
FileWriter f2 = new FileWriter("file3.txt") )
{
// write to first file
for (int i=0; i < buffer.length; i += 2) {
f0.write(buffer[i]);
}
// write to second file
f1.write(buffer);
// write to third file
f2.write(buffer,buffer.length-buffer.length/4,buffer.length/4);
} catch(IOException e) {
System.out.println("An I/O Error Occurred");
}
}
}
```

CharArrayReader
CharArrayReader is an implementation of an input stream that uses a character array as the source. This class has two constructors, each of which requires a character array to provide the data source:

- CharArrayReader(char *array* [ ])
- CharArrayReader(char *array* [ ], int *start*, int *numChars*)

Here, *array* is the input source. The second constructor creates a **Reader** from a subset of your character array that begins with the character at the index specified by *start* and is *numChars* long.
The **close( )** method implemented by **CharArrayReader** does not throw any exceptions. This is because it cannot fail.
The following example uses a pair of **CharArrayReader**s:

```
// Demonstrate CharArrayReader.
// This program uses try-with-resources. It requires JDK 7 or later.
import java.io.*;
public class CharArrayReaderDemo {
public static void main(String args[]) {
String tmp = "abcdefghijklmnopqrstuvwxyz";
int length = tmp.length();
char c[] = new char[length];
tmp.getChars(0, length, c, 0);
int i;
try (CharArrayReader input1 = new CharArrayReader(c) )
{
System.out.println("input1 is:");
while((i = input1.read()) != -1) {
System.out.print((char)i);
}
System.out.println();
} catch(IOException e) {
System.out.println("I/O Error: " + e);
}
try ( CharArrayReader input2 = new CharArrayReader(c, 0, 5) )
{
System.out.println("input2 is:");
while((i = input2.read()) != -1) {
System.out.print((char)i);
}
System.out.println();
} catch(IOException e) {
System.out.println("I/O Error: " + e);
}
}
}
```

The **input1** object is constructed using the entire lowercase alphabet, whereas **input2** contains only the first five letters. Here is the output:

```
input1 is:
abcdefghijklmnopqrstuvwxyz
input2 is:
abcde
```

CharArrayWriter

CharArrayWriter is an implementation of an output stream that uses an array as the destination. **CharArrayWriter** has two constructors, shown here:

- CharArrayWriter( )
- CharArrayWriter(int *numChars*)

In the first form, a buffer with a default size is created. In the second, a buffer is created with a size equal to that specified by *numChars*. The buffer is held in the **buf** field of **CharArrayWriter**. The buffer size will be increased automatically, if needed. The number of characters held by the buffer is contained in the **count** field of **CharArrayWriter**. Both **buf**
and **count** are protected fields.The **close( )** method has no effect on a **CharArrayWriter**.
The following example demonstrates **CharArrayWriter** by reworking the sample program shown earlier for **ByteArrayOutputStream**. It produces the same output as the previous version.

```
// Demonstrate CharArrayWriter.
// This program uses try-with-resources. It requires JDK 7 or later.
import java.io.*;
class CharArrayWriterDemo {
public static void main(String args[]) throws IOException {
CharArrayWriter f = new CharArrayWriter();
String s = "This should end up in the array";
char buf[] = new char[s.length()];
s.getChars(0, s.length(), buf, 0);
try {
f.write(buf);
} catch(IOException e) {
System.out.println("Error Writing to Buffer");
return;
}
System.out.println("Buffer as a string");
System.out.println(f.toString());
System.out.println("Into array");
char c[] = f.toCharArray();
for (int i=0; i<c.length; i++) {
System.out.print(c[i]);
}
System.out.println("\nTo a FileWriter()");
// Use try-with-resources to manage the file stream.
try ( FileWriter f2 = new FileWriter("test.txt") )
{
f.writeTo(f2);
} catch(IOException e) {
System.out.println("I/O Error: " + e);
}
System.out.println("Doing a reset");
f.reset();
for (int i=0; i<3; i++) f.write('X');
System.out.println(f.toString());
}
}
```

BufferedReader

BufferedReader improves performance by buffering input. It has two constructors:

- BufferedReader(Reader *inputStream*)
- BufferedReader(Reader *inputStream*, int *bufSize*)

The first form creates a buffered character stream using a default buffer size. In the second, the size of the buffer is passed in *bufSize*.

Closing a **BufferedReader** also causes the underlying stream specified by *inputStream* to be closed. As is the case with the byte-oriented stream, buffering an input character stream also provides the foundation required to support moving backward in the stream within the available buffer. To support this, **BufferedReader** implements the **mark( )** and **reset( )**
methods, and **BufferedReader.markSupported( )** returns **true**. JDK 8 adds a new method to**BufferedReader** called **lines( )**. It returns a **Stream** reference to the sequence of lines read by the reader.

The following example reworks the **BufferedInputStream** example, shown earlier, so that it uses a **BufferedReader** character stream rather than a buffered byte stream. As before, it uses the **mark( )** and **reset( )** methods to parse a stream for the HTML entity reference for the copyright symbol. Such a reference begins with an ampersand (&) and ends with a semicolon (;) without any intervening whitespace. The sample input has two ampersands to show the case where the **reset( )** happens and where it does not. Output is the same as that shown earlier.

```
// Use buffered input.
// This program uses try-with-resources. It requires JDK 7 or later.
import java.io.*;
class BufferedReaderDemo {
public static void main(String args[]) throws IOException {
String s = "This is a &copy; copyright symbol " +
"but this is &copy not.\n";
char buf[] = new char[s.length()];
s.getChars(0, s.length(), buf, 0);
CharArrayReader in = new CharArrayReader(buf);
int c;
boolean marked = false;
try ( BufferedReader f = new BufferedReader(in) )
{
while ((c = f.read()) != -1) {
switch(c) {
case '&':
if (!marked) {
```

```
f.mark(32);
marked = true;
} else {
marked = false;
}
break;
case ';':
if (marked) {
marked = false;
System.out.print("(c)");
} else
System.out.print((char) c);
break;
case ' ':
if (marked) {
marked = false;
f.reset();
System.out.print("&");
} else
System.out.print((char) c);
break;
default:
if (!marked)
System.out.print((char) c);
break;
}
}
} catch(IOException e) {
System.out.println("I/O Error: " + e);
}
}
```

BufferedWriter

A **BufferedWriter** is a **Writer** that buffers output. Using a **BufferedWriter** can improve performance by reducing the number of times data is actually physically written to the output device.

A **BufferedWriter** has these two constructors:

- BufferedWriter(Writer *outputStream*)
- BufferedWriter(Writer *outputStream*, int *bufSize*)

The first form creates a buffered stream using a buffer with a default size. In the second, the size of the buffer is passed in *bufSize*.

PushbackReader

The **PushbackReader** class allows one or more characters to be returned to the input stream. This allows you to look ahead in the input stream. Here are its two constructors:

- PushbackReader(Reader *inputStream*)
- PushbackReader(Reader *inputStream*, int *bufSize*)

The first form creates a buffered stream that allows one character to be pushed back. In the second, the size of the pushback buffer is passed in *bufSize* Closing a **PushbackReader** also closes the underlying stream specified by *inputStream*.

PushbackReader provides **unread( )**, which returns one or more characters to the invoking input stream. It has the three forms shown here:

- void unread(int *ch*) throws IOException
- void unread(char *buffer* [ ]) throws IOException
- void unread(char *buffer* [ ], int *offset*, int *numChars*) throws IOException

The first form pushes back the character passed in *ch*. This will be the next character returned by a subsequent call to **read( )**. The second form returns the characters in *buffer*. The third form pushes back *numChars* characters beginning at *offset* from *buffer*. An **IOException** will be thrown if there is an attempt to return a character when the pushback buffer is full. The following program reworks the earlier **PushbackInputStream** example by replacing **PushbackInputStream** with **PushbackReader**. As before, it shows how a programming language parser can use a pushback stream to deal with the difference between the **==** operator for comparison and the **=** operator for assignment.

```
// Demonstrate unread().
import java.io.*;
class PushbackReaderDemo {
public static void main(String args[]) {
String s = "if (a == 4) a = 0;\n";
char buf[] = new char[s.length()];
s.getChars(0, s.length(), buf, 0);
CharArrayReader in = new CharArrayReader(buf);
int c;
try ( PushbackReader f = new PushbackReader(in) )
{
while ((c = f.read()) != -1) {
switch(c) {
case '=':
if ((c = f.read()) == '=')
System.out.print(".eq.");
else {
System.out.print("<-");
f.unread(c);
}
break;
default:
System.out.print((char) c);
break;
}
}
} catch(IOException e) {
System.out.println("I/O Error: " + e);
}
}
}
```

PrintWriter

PrintWriter is essentially a character-oriented version of **PrintStream**. It implements the **Appendable**, **AutoCloseable**, **Closeable**, and **Flushable** interfaces. **PrintWriter** has several constructors. The following have been supplied by **PrintWriter** from the start:

- PrintWriter(OutputStream *outputStream*)
- PrintWriter(OutputStream *outputStream*, boolean *autoFlushingOn*)
- PrintWriter(Writer *outputStream*)
- PrintWriter(Writer *outputStream*, boolean *autoFlushingOn*)

Here, *outputStream* specifies an open **OutputStream** that will receive output. The *autoFlushingOn* parameter controls whether the output buffer is automatically flushed every time **println( )**, **printf( )**, or **format( )** is called. If *autoFlushingOn* is **true**, flushing automatically takes place. If **false**, flushing is not automatic. Constructors that do not
specify the *autoFlushingOn* parameter do not automatically flush. The next set of constructors gives you an easy way to construct a **PrintWriter** that writes its output to a file.

- PrintWriter(File *outputFile*) throws FileNotFoundException
- PrintWriter(File *outputFile*, String *charSet*) throws FileNotFoundException, UnsupportedEncodingException
- PrintWriter(String *outputFileName*) throws FileNotFoundException
- PrintWriter(String *outputFileName*, String *charSet*) throws FileNotFoundException, UnsupportedEncodingException

These allow a **PrintWriter** to be created from a **File** object or by specifying the name of a file. In either case, the file is automatically created. Any preexisting file by the same name is destroyed. Once created, the **PrintWriter** object directs all output to the specified file. You can specify a character encoding by passing its name in *charSet*. **PrintWriter** supports the **print( )** and **println( )** methods for all types, including **Object**. If an argument is not a primitive type, the **PrintWriter** methods will call the object's **toString( )** method and then output the result.

- **PrintWriter** also supports the **printf( )** method. It works the same way it does in the
- **PrintStream** class described earlier: It allows you to specify the precise format of the data.

Here is how **printf( )** is declared in **PrintWriter**:

- PrintWriter printf(String *fmtString*, Object … *args*)
- PrintWriter printf(Locale *loc*, String *fmtString*, Object …*args*)

The first version writes *args* to standard output in the format specified by *fmtString*, using the default locale. The second lets you specify a locale. Both return the invoking **PrintWriter**.
The **format( )** method is also supported. It has these general forms:

- PrintWriter format(String *fmtString*, Object … *args*)
- PrintWriter format(Locale *loc*, String *fmtString*, Object … *args*)

It works exactly like **printf( )**.

3.3 Serialization

Serialization is the process of writing the state of an object to a byte stream. This is useful when you want to save the state of your program to a persistent storage area, such as a file. At a later time, you may restore these objects by using the process of deserialization. Serialization is also needed to implement *Remote Method Invocation (RMI)*. RMI allows a Java object on one machine to invoke a method of a Java object on a different machine. An object may be supplied as an argument to that remote method. The sending machine serializes the object and transmits it. The receiving machine deserializes it.
Assume that an object to be serialized has references to other objects, which, in turn, have references to still more objects. This set of objects and the relationships among them form a directed graph. There may also be circular references within this object graph. That is, object X may contain a reference to object Y, and object Y may contain a reference back to object X. Objects may also contain references to themselves. The object serialization and deserialization facilities have been designed to work correctly in these scenarios. If you attempt to serialize an object at the top of an object graph, all of the other referenced objects are recursively located and serialized. Similarly, during the process of deserialization, all of these objects and their references are correctly restored. An overview of the interfaces and classes that support serialization follows.

Serializable

Only an object that implements the **Serializable** interface can be saved and restored by the serialization facilities. The **Serializable** interface defines no members. It is simply used to indicate that a class may be serialized. If a class is serializable, all of its subclasses are also serializable.
Variables that are declared as **transient** are not saved by the serialization facilities. Also, **static** variables are not saved.

Externalizable

The Java facilities for serialization and deserialization have been designed so that much of the work to save and restore the state of an object occurs automatically. However, there are cases in which the programmer may need to have control over these processes. For example, it may be desirable to use compression or encryption techniques. The **Externalizable** interface is designed for these situations.
The **Externalizable** interface defines these two methods:

- void readExternal(ObjectInput *inStream*) throws IOException, ClassNotFoundException
- void writeExternal(ObjectOutput *outStream*) throws IOException

In these methods, *inStream* is the byte stream from which the object is to be read, and *outStream* is the byte stream to which the object is to be written.

ObjectOutput

The **ObjectOutput** interface extends the **DataOutput** and **AutoCloseable** interfaces and supports object serialization. It defines the methods shown in Table Note especially the **writeObject( )** method. This is called to serialize an object. All of these methods will throw an **IOException** on error conditions.

Method	Description
void close( )	Closes the invoking stream. Further write attempts will generate an **IOException**.
void flush( )	Finalizes the output state so any buffers are cleared. That is, it flushes the output buffers.
void write(byte *buffer*[ ])	Writes an array of bytes to the invoking stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].
void write(int *b*)	Writes a single byte to the invoking stream. The byte written is the low-order byte of *b*.
void writeObject(Object *obj*)	Writes object *obj* to the invoking stream.

Table 3.4 Methods Defined by Object Output

ObjectOutputStream
The **ObjectOutputStream** class extends the **OutputStream** class and implements the **ObjectOutput** interface. It is responsible for writing objects to a stream. One constructor of this class is shown here:
ObjectOutputStream(OutputStream *outStream*) throws IOException
The argument *outStream* is the output stream to which serialized objects will be written.
Closing an **ObjectOutputStream** automatically closes the underlying stream specified by *outStream*.
Several commonly used methods in this class are shown in Table 3.5 They will throw an **IOException** on error conditions. There is also an inner class to **ObjectOuputStream** called **PutField**.

Method	Description
void close( )	Closes the invoking stream. Further write attempts will generate an **IOException**. The underlying stream is also closed.
void flush( )	Finalizes the output state so any buffers are cleared. That is, it flushes the output buffers.
void write(byte *buffer*[ ])	Writes an array of bytes to the invoking stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].
void write(int *b*)	Writes a single **byte** to the invoking stream. The byte written is the low-order byte of *b*.
void writeBoolean(boolean *b*)	Writes a **boolean** to the invoking stream.
void writeByte(int *b*)	Writes a **byte** to the invoking stream. The byte written is the low-order byte of *b*.
void writeBytes(String *str*)	Writes the bytes representing *str* to the invoking stream.
void writeChar(int *c*)	Writes a **char** to the invoking stream.
void writeChars(String *str*)	Writes the characters in *str* to the invoking stream.
void writeDouble(double *d*)	Writes a **double** to the invoking stream.
void writeFloat(float *f*)	Writes a **float** to the invoking stream.
void writeInt(int *i*)	Writes an **int** to the invoking stream.
void writeLong(long *l*)	Writes a **long** to the invoking stream.
final void writeObject(Object *obj*)	Writes *obj* to the invoking stream.
void writeShort(int *i*)	Writes a **short** to the invoking stream.

Table 3.5 sampling of Commonly used methods defined by ObjectOutput Stream

ObjectInput
The **ObjectInput** interface extends the **DataInput** and **AutoCloseable** interfaces and defines the methods shown in Table 20-8. It supports object serialization. Note especially the **readObject( )** method. This is called to deserialize an object. All of these methods will throw an **IOException** on error conditions. The **readObject( )** method can also throw **ClassNotFoundException**.

Method	Description
int available( )	Returns the number of bytes that are now available in the input buffer.
void close( )	Closes the invoking stream. Further read attempts will generate an **IOException**.
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ])	Attempts to read up to *buffer.length* bytes into *buffer*, returning the number of bytes that were successfully read. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes that were successfully read. –1 is returned when the end of the file is encountered.
Object readObject( )	Reads an object from the invoking stream.
long skip(long *numBytes*)	Ignores (that is, skips) *numBytes* bytes in the invoking stream, returning the number of bytes actually ignored.

Table 3.6 Methods Defined by Object Input

ObjectInputStream
The **ObjectInputStream** class extends the **InputStream** class and implements the **ObjectInput** interface. **ObjectInputStream** is responsible for reading objects from a stream. One constructor of this class is shown here:
ObjectInputStream(InputStream *inStream*) throws IOException
The argument *inStream* is the input stream from which serialized objects should be read. Closing an **ObjectInputStream** automatically closes the underlying stream specified by *inStream*.
Several commonly used methods in this class are shown in Table 3.7 . They will throw an **IOException** on error conditions. The **readObject( )** method can also throw **ClassNotFoundException**. There is also an inner class to **ObjectInputStream** called **GetField**.

Method	Description
int available( )	Returns the number of bytes that are now available in the input buffer.
void close( )	Closes the invoking stream. Further read attempts will generate an **IOException**. The underlying stream is also closed.
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered.
Boolean readBoolean( )	Reads and returns a **boolean** from the invoking stream.
byte readByte( )	Reads and returns a **byte** from the invoking stream.
char readChar( )	Reads and returns a **char** from the invoking stream.
double readDouble( )	Reads and returns a **double** from the invoking stream.
float readFloat( )	Reads and returns a **float** from the invoking stream.
void readFully(byte *buffer*[ ])	Reads *buffer.length* bytes into *buffer*. Returns only when all bytes have been read.
void readFully(byte *buffer*[ ], int *offset*, int *numBytes*)	Reads *numBytes* bytes into *buffer* starting at *buffer*[*offset*]. Returns only when *numBytes* have been read.
int readInt( )	Reads and returns an **int** from the invoking stream.
long readLong( )	Reads and returns a **long** from the invoking stream.
final Object readObject( )	Reads and returns an object from the invoking stream.
short readShort( )	Reads and returns a **short** from the invoking stream.
int readUnsignedByte( )	Reads and returns an unsigned **byte** from the invoking stream.
int readUnsignedShort( )	Reads and returns an unsigned **short** from the invoking stream.

Table 3.7Commonly used Methods Defined by ObjectInputStream

A Serialization Example
The following program illustrates how to use object serialization and deserialization. It begins by instantiating an object of class **MyClass**. This object has three instance variables that are of types **String**, **int**,

and **double**. This is the information we want to save and restore. A **FileOutputStream** is created that refers to a file named "serial", and an **ObjectOutputStream** is created for that file stream. The **writeObject( )** method of **ObjectOutputStream** is then used to serialize our object. The object output stream is flushed and closed. A **FileInputStream** is then created that refers to the file named "serial", and an **ObjectInputStream** is created for that file stream. The **readObject( )** method of **ObjectInputStream** is then used to deserialize our object. The object input stream is then closed.
Note that **MyClass** is defined to implement the **Serializable** interface.If if is not done, a N**otSerializableException** is thrown. Try experimenting with this program by declaring some of the **MyClass** instance variables to be **transient**. That data is then not saved during serialization.

```
// A serialization demo.import java.io.*;
public class SerializationDemo {
public static void main(String args[]) {
// Object serialization
try ( ObjectOutputStream objOStrm =
new ObjectOutputStream(new FileOutputStream("serial")) )
{
MyClass object1 = new MyClass("Hello", -7, 2.7e10);
System.out.println("object1: " + object1);
objOStrm.writeObject(object1);
}
catch(IOException e) {
System.out.println("Exception during serialization: " + e);
}
// Object deserialization
try ( ObjectInputStream objIStrm =
new ObjectInputStream(new FileInputStream("serial")) )
{
MyClass object2 = (MyClass)objIStrm.readObject();
System.out.println("object2: " + object2);
}
catch(Exception e) {
System.out.println("Exception during deserialization: " + e);
}
}
}
class MyClass implements Serializable {
String s;
int i;
double d;
public MyClass(String s, int i, double d) {
this.s = s;
this.i = i;
this.d = d;
}
public String toString() {
return "s=" + s + "; i=" + i + "; d=" + d;
}
}
```

This program demonstrates that the instance variables of **object1** and **object2** are identical.
The output is shown here:

```
object1: s=Hello; i=-7; d=2.7E10
object2: s=Hello; i=-7; d=2.7E10
```

Stream Benefits
The streaming interface to I/O in Java provides a clean abstraction for a complex and often cumbersome task. The composition of the filtered stream classes allows you to dynamically build the custom streaming interface to suit your data transfer requirements.

Reading and Writing Files
Java provides a number of classes and methods that allow you to read and write files. Although bytes streams are used, these techniques can be adapted to the character-based streams.
Two of the most often-used stream classes are **FileInputStream** and **FileOutputStream**, which create byte streams linked to files. To open a file, you simply create an object of one of these classes, specifying the name of the file as an argument to the constructor. Although both classes support additional constructors, the following are the forms that we will be using:

- FileInputStream(String *fileName*) throws FileNotFoundException
- FileOutputStream(String *fileName*) throws FileNotFoundException

Here, *fileName* specifies the name of the file that you want to open. When you create an input stream, if the file does not exist, then **FileNotFoundException** is thrown. For output streams, if the file cannot be opened or created, then **FileNotFoundException** is thrown.
FileNotFoundException is a subclass of **IOException**.
When an output file is opened, any preexisting file by the same name is destroyed. When you are done with a file, you must close it. This is done by calling the **close( )** method, which is implemented by both **FileInputStream** and **FileOutputStream**. It is shown here:
void close( ) throws IOException
Closing a file releases the system resources allocated to the file, allowing them to be used by another file. Failure to close a file can result in “memory leaks” because of unused resources remaining allocated.
To read from a file, you can use a version of **read( )** that is defined within **FileInputStream**.
The one that we will use is shown here:
int read( ) throws IOException
Each time that it is called, it reads a single byte from the file and returns the byte as an integer value. **read( )** returns –1 when the end of the file is encountered. It can throw an
IOException.
The following program uses **read( )** to input and display the contents of a file that contains ASCII text. The name of the file is specified as a command-line argument.

```
/* Display a text file.
To use this program, specify the name
of the file that you want to see.
For example, to see a file called TEST.TXT,
use the following command line.
java ShowFile TEST.TXT
*/
import java.io.*;
class ShowFile {
public static void main(String args[])
{
int i;
FileInputStream fin;
// First, confirm that a filename has been specified.
if(args.length != 1) {
System.out.println("Usage: ShowFile filename");
return;
}
// Attempt to open the file.
try {
fin = new FileInputStream(args[0]);
} catch(FileNotFoundException e) {
System.out.println("Cannot Open File");
return;
}
// At this point, the file is open and can be read.
// The following reads characters until EOF is encountered.
try {
do {
```

```
i = fin.read();
if(i != -1) System.out.print((char) i);
} while(i != -1);
} catch(IOException e) {
System.out.println("Error Reading File");
}
// Close the file.
try {
fin.close();
} catch(IOException e) {
System.out.println("Error Closing File");
}
}
}
```

In the program, notice the **try/catch** blocks that handle the I/O errors that might occur. Each I/O operation is monitored for exceptions, and if an exception occurs, it is handled. Be aware that in simple programs or example code, it is common to see I/O exceptions simply thrown out of **main( )**, as was done in the earlier console I/O examples.
Also, in some real-world code, it can be helpful to let an exception propagate to a calling routine to let the caller know that an I/O operation failed. However, most of the file I/O examples in this book handle all I/O exceptions explicitly, as shown, for the sake of illustration.
Although the preceding example closes the file stream after the file is read, there is a variation that is often useful. The variation is to call **close( )** within a **finally** block. In this approach, all of the methods that access the file are contained within a **try** block, and the **finally** block is used to close the file. This way, no matter how the **try** block terminates, the file is closed. Assuming the preceding example, here is how the **try** block that reads the file can be recoded:

```
try {
do {
i = fin.read();
if(i != -1) System.out.print((char) i);
} while(i != -1);
} catch(IOException e) {
System.out.println("Error Reading File");
} finally {
// Close file on the way out of the try block.
try {
fin.close();
} catch(IOException e) {
System.out.println("Error Closing File");
}
}
```

Although not an issue in this case, one advantage to this approach in general is that if the code that accesses a file terminates because of some non-I/O related exception, the file is still closed by the **finally** block.
Sometimes it's easier to wrap the portions of a program that open the file and access the file within a single **try** block (rather than separating the two) and then use a **finally** block to close the file. For example, here is another way to write the **ShowFile** program:

```
/* Display a text file.
To use this program, specify the name
of the file that you want to see.
For example, to see a file called TEST.TXT,
use the following command line.
java ShowFile TEST.TXT
This variation wraps the code that opens and
accesses the file within a single try block.
The file is closed by the finally block.
```

```
*/
import java.io.*;
class ShowFile {
public static void main(String args[])
{
int i;
FileInputStream fin = null;
// First, confirm that a filename has been specified.
if(args.length != 1) {
System.out.println("Usage: ShowFile filename");
return;
}
// The following code opens a file, reads characters until EOF
// is encountered, and then closes the file via a finally block.
try {
fin = new FileInputStream(args[0]);
do {
i = fin.read();
if(i != -1) System.out.print((char) i);
} while(i != -1);
} catch(FileNotFoundException e) {
System.out.println("File Not Found.");
} catch(IOException e) {
System.out.println("An I/O Error Occurred");
} finally {
// Close file in all cases.
try {
if(fin != null) fin.close();
} catch(IOException e) {
System.out.println("Error Closing File");
}
}
}
}
```

In this approach, notice that **fin** is initialized to **null**. Then, in the **finally** block, the file is closed only if **fin** is not **null**. This works because **fin** will be non-**null** only if the file is successfully opened. Thus, **close( )** is not called if an exception occurs while opening the file. It is possible to make the **try/catch** sequence in the preceding example a bit more compact. Because **FileNotFoundException** is a subclass of **IOException**, it need not be caught separately. For example, here is the sequence recoded to eliminate catching **FileNotFoundException**. In this case, the standard exception message, which describes the error, is displayed.

```
try {
fin = new FileInputStream(args[0]);
do {
i = fin.read();
if(i != -1) System.out.print((char) i);
} while(i != -1);
} catch(IOException e) {
System.out.println("I/O Error: " + e);
} finally {
// Close file in all cases.
try {
if(fin != null) fin.close();
} catch(IOException e) {
System.out.println("Error Closing File");
}
}
```

In this approach, any error, including an error opening the file, is simply handled by the single **catch** statement. Because of its compactness, this approach is used by many of the I/O examples in this book. Be aware, however, that this approach is not appropriate in cases in which you want to deal separately with a failure to open a file, such as might be caused if a user mistypes a filename. In such a situation, you might want to prompt for the correct name, for example, before entering a **try** block that accesses the file. To write to a file, you can use the **write( )** method defined by **FileOutputStream**. Its simplest form is shown here:

void write(int *byteval*) throws IOException

This method writes the byte specified by *byteval* to the file. Although *byteval* is declared as an integer, only the low-order eight bits are written to the file. If an error occurs during writing, an **IOException** is thrown. The next example uses **write( )** to copy a file:

```
/* Copy a file.
To use this program, specify the name
of the source file and the destination file.
For example, to copy a file called FIRST.TXT
to a file called SECOND.TXT, use the following
command line.
java CopyFile FIRST.TXT SECOND.TXT
*/
import java.io.*;
class CopyFile {
public static void main(String args[]) throws IOException
{
int i;
FileInputStream fin = null;
FileOutputStream fout = null;
// First, confirm that both files have been specified.
if(args.length != 2) {
System.out.println("Usage: CopyFile from to");
return;
}
// Copy a File.
try {
// Attempt to open the files.
fin = new FileInputStream(args[0]);
fout = new FileOutputStream(args[1]);
do {
i = fin.read();
if(i != -1) fout.write(i);
} while(i != -1);
} catch(IOException e) {
System.out.println("I/O Error: " + e);
} finally {
try {
if(fin != null) fin.close();
} catch(IOException e2) {
System.out.println("Error Closing Input File");
}
try {
if(fout != null) fout.close();
} catch(IOException e2) {
System.out.println("Error Closing Output File");
}
}
}
}
```

In the program, notice that two separate **try** blocks are used when closing the files. This ensures that both files are closed, even if the call to **fin.close( )** throws an exception. In general, notice that all potential I/O errors are handled in the preceding two programs by the use of exceptions. This differs from some computer languages that use error codes to report file errors. Not only do exceptions make file handling cleaner, but they also enable Java to easily differentiate the end-of-file condition from file errors when input is being performed.

Automatically Closing a File

In the preceding section, the example programs have made explicit calls to **close( )** to close a file once it is no longer needed. As mentioned, this is the way files were closed when using versions of Java prior to JDK 7. Although this approach is still valid and useful, JDK 7 added a new feature that offers another way to manage resources, such as file streams, by automating the closing process. This feature, sometimes referred to as *automatic resource management*, or *ARM* for short, is based on an expanded version of the **try** statement. The principal advantage
of automatic resource management is that it prevents situations in which a file (or other resource) is inadvertently not released after it is no longer needed. As explained, forgetting to close a file can result in memory leaks, and could lead to other problems.

3.4 String Handling in java

In Java a string is a sequence of characters. But, unlike some other languages that implement strings as character
arrays, Java implements strings as objects of type **String**. Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient. For example, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string. Also, **String** objects can be constructed a number of ways, making it easy to obtain a string when needed. Somewhat unexpectedly, when you create a **String** object, you are creating a string that cannot be changed. That is, once a **String** object has been created, you cannot change the characters that comprise that string. At first, this may seem to be a serious restriction. However, such is not the case. You can still perform all types of string operations. The
difference is that each time you need an altered version of an existing string, a new **String**
object is created that contains the modifications. The original string is left unchanged. This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones. For those cases in which a modifiable string is desired, Java provides two options: **StringBuffer** and **StringBuilder**. Both hold strings that can be modified after they are created.
The **String**, **StringBuffer**, and **StringBuilder** classes are defined in **java.lang**. Thus, they are available to all programs automatically. All are declared **final**, which means that none of these classes may be subclassed. This allows certain optimizations that increase performance to take place on common string operations. All three implement the **CharSequence** interface. One last point: To say that the strings within objects of type **String** are unchangeable means that the contents of the **String** instance cannot be changed after it has been created. However, a variable declared as a **String** reference can be changed to point at some other **String** object at any time.

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

No.	Method	Description
1	char charAt(int index)	returns char value for the particular index
2	int length()	returns string length
3	static String format(String format, Object... args)	returns a formatted string.
4	static String format(Locale l, String format, Object... args)	returns formatted string with given locale.

5	String substring(int beginIndex)	returns substring for given begin index.
6	String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index.
7	boolean contains(CharSequence s)	returns true or false after matching the sequence of char value.
8	static String join(CharSequence delimiter, CharSequence... elements)	returns a joined string.
9	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	returns a joined string.
10	boolean equals(Object another)	checks the equality of string with the given object.
11	boolean isEmpty()	checks if string is empty.
12	String concat(String str)	concatenates the specified string.
13	String replace(char old, char new)	replaces all occurrences of the specified char value.
14	String replace(CharSequence old, CharSequence new)	replaces all occurrences of the specified CharSequence.
15	static String equalsIgnoreCase(String another)	compares another string. It doesn't check case.
16	String[] split(String regex)	returns a split string matching regex.
17	String[] split(String regex, int limit)	returns a split string matching regex and limit.
18	String intern()	returns an interned string.
19	int indexOf(int ch)	returns the specified char value index.
20	int indexOf(int ch, int fromIndex)	returns the specified char value index starting with given index.
21	int indexOf(String substring)	returns the specified substring index.

22	int indexOf(String substring, int fromIndex)	returns the specified substring index starting with given index.
23	String toLowerCase()	returns a string in lowercase.
24	String toLowerCase(Locale l)	returns a string in lowercase using specified locale.
25	String toUpperCase()	returns a string in uppercase.
26	String toUpperCase(Locale l)	returns a string in uppercase using specified locale.
27	String trim()	removes beginning and ending spaces of this string.
28	static String valueOf(int value)	converts given type into string. It is an overloaded method.

The String Constructors
The **String** class supports several constructors. To create an empty **String**, call the default constructor. For example,
String s = new String();
will create an instance of **String** with no characters in it.
Frequently, you will want to create strings that have initial values. The **String** class provides a variety of Constructors to handle this. To create a **String** initialized by an array of characters, use the constructor shown here:
String(char *chars*[ ])
Here is an example:
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
This constructor initializes **s** with the string "abc".

You can specify a subrange of a character array as an initializer using the following constructor:
String(char *chars*[ ], int *startIndex*, int *numChars*)
Here, *startIndex* specifies the index at which the subrange begins, and *numChars* specifies the number of characters to use. Here is an example:
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars, 2, 3);
This initializes **s** with the characters **cde.**
You can construct a **String** object that contains the same character sequence as another **String** object using this constructor:
String(String *strObj*)
Here, *strObj* is a **String** object. Consider this example:
// Construct one String from another.
class MakeString {
public static void main(String args[]) {
char c[] = {'J', 'a', 'v', 'a'};
String s1 = new String(c);
String s2 = new String(s1);
System.out.println(s1);

```
System.out.println(s2);
}
}
```

The output from this program is as follows:

```
Java
Java
```

As you can see, s1 and s2 contain the same string.
Even though Java’s char type uses 16 bits to represent the basic Unicode character set, the typical format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set. Because 8-bit ASCII strings are common, the String class provide constructors that initialize a string when given a byte array. Two forms are shown here:

- String(byte chrs[ ])
- String(byte chrs[ ], int startIndex, int numChars)

Here, chrs specifies the array of bytes. The second form allows you to specify a subrange. In each of these constructors, the byte-to-character conversion is done by using the default character encoding of the platform. The following program illustrates these constructors:

```
// Construct string from subset of char array.
class SubStringCons {
public static void main(String args[]) {
byte ascii[] = {65, 66, 67, 68, 69, 70 };
String s1 = new String(ascii);
System.out.println(s1);
String s2 = new String(ascii, 2, 3);
System.out.println(s2);
}
}
```

This program generates the following output:

```
ABCDEF
CDE
```

Extended versions of the byte-to-string constructors are also defined in which you can specify the character encoding that determines how bytes are converted to characters. However, you will often want to use the default encoding provided by the platform. NOTE The contents of the array are copied whenever you create a String object from an array. If you modify the contents of the array after you have created the string, the String will be unchanged.
You can construct a String from a StringBuffer by using the constructor shown here:
String(StringBuffer strBufObj)
You can construct a String from a StringBuilder by using this constructor:
String(StringBuilder strBuildObj)

The following constructor supports the extended Unicode character set: String(int codePoints[ ], int startIndex, int numChars)
Here, codePoints is an array that contains Unicode code points. The resulting string is constructed from the range that begins at startIndex and runs for numChars. There are also constructors that let you specify a Charset.
The length of a string is the number of characters that it contains. To obtain this value, call the length( ) method, shown here:
int length( )
The following fragment prints "3", since there are three characters in the string s:

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

Special String Operations

Because strings are a common and important part of programming, Java has added special support for several string operations within the syntax of the language. These operations include the automatic creation of new String instances from string literals, concatenation of multiple String objects by use of the + operator, and the

conversion of other data types to a string representation. There are explicit methods available to perform all of these functions,
but Java does them atomatically as a convenience for the programmer and to add clarity. String Literals The earlier examples showed how to explicitly create a String instance from an array of characters by using the new operator. However, there is an easier way to do this using a string literal. For each string literal in your program, Java automatically constructs a String object. Thus, you can use a string literal to initialize a String object. For example, the following code fragment creates two equivalent strings:

```
char chars[] = { 'a', 'b', 'c' };
String s1 = new String(chars);
String s2 = "abc"; // use string literal
```

Because a String object is created for every string literal, you can use a string literal any place you can use a String object. For example, you can call methods directly on a quoted string as if it were an object reference, as the following statement shows. It calls the length( ) method on the string "abc". As expected, it prints "3".

```
System.out.println("abc".length());
```

String Concatenation

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

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

This displays the string "He is 9 years old."

One practical use of string concatenation is found when you are creating very long strings. Instead of letting long strings wrap around within your source code, you can break them into smaller pieces, using the + to concatenate them. Here is an example:

```
// Using concatenation to prevent long lines.
class ConCat {
public static void main(String args[]) {
String longStr = "This could have been " +
"a very long line that would have " +
"wrapped around. But string concatenation " +
"prevents this.";
System.out.println(longStr);
}
}
```

String Concatenation with Other Data Types

You can concatenate strings with other types of data. For example, consider this slightly different version of the earlier example:

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

In this case, age is an int rather than another String, but the output produced is the same as before. This is because the int value in age is automatically converted into its string representation within a String object. This string is then concatenated as before. The compiler will convert an operand to its string equivalent whenever the other operand of
the + is an instance of String.

Be careful when you mix other types of operations with string concatenation expressions, however. You might get surprising results. Consider the following:

```
String s = "four: " + 2 + 2;
System.out.println(s);
```

This fragment displays

```
four: 22
```

rather than the

```
four: 4
```

that you probably expected. Here’s why. Operator precedence causes the concatenation of "four" with the string equivalent of 2 to take place first. This result is then concatenated with the string equivalent of 2 a second time. To complete the integer addition first, you must use parentheses, like this: String s = "four: " + (2 + 2); Now s contains the string "four: 4".

String Conversion and toString( )

When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method valueOf( ) defined by String. valueOf( ) is overloaded for all the primitive types and for type Object. For the primitive types, valueOf( ) returns a string that contains the human-readable equivalent of the value with which it is called. For objects, valueOf( ) calls the toString( ) method on the
object. We will look more closely at valueOf( ) later in this chapter. Here, let's examine the toString( ) method, because it is the means by which you can determine the string representation for objects of classes that you create.
Every class implements toString( ) because it is defined by Object. However, the default implementation of toString( ) is seldom sufficient. For most important classes that you create, you will want to override toString( ) and provide your own string representations.
Fortunately, this is easy to do. The toString( ) method has this general form:

```
String toString( )
```

To implement toString( ), simply return a String object that contains the human-readable string that appropriately describes an object of your class.
By overriding toString( ) for classes that you create, you allow them to be fully integrated into Java's programming environment. For example, they can be used in print( ) and println( ) statements and in concatenation expressions. The following program
demonstrates this by overriding toString( ) for the Box class:

```
// Override toString() for Box class.
class Box {
double width;
double height;
double depth;
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
public String toString() {
return "Dimensions are " + width + " by " +

}
}
class toStringDemo {
public static void main(String args[]) {
Box b = new Box(10, 12, 14);
String s = "Box b: " + b; // concatenate Box object
System.out.println(b); // convert Box to string
System.out.println(s);
}
}
```

The output of this program is shown here:

```
Dimensions are 10.0 by 14.0 by 12.0
Box b: Dimensions are 10.0 by 14.0 by 12.0
```

As you can see, Box's toString( ) method is automatically invoked when a Box object is used in a concatenation expression or in a call to println( ).

Character Extraction

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

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

```
char charAt(int where)
```

Here, where is the index of the character that you want to obtain. The value of where must be nonnegative and specify a location within the string. charAt( ) returns the character at the specified location. For example,

```
char ch;
ch = "abc".charAt(1);
```

assigns the value b to ch.

getChars( )

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

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

Here, sourceStart specifies the index of the beginning of the substring, and sourceEnd specifies an index that is one past the end of the desired substring. Thus, the substring contains the characters from sourceStart through sourceEnd–1. The array that will receive the characters is specified by target. The index within target at which the substring will be copied is passed in targetStart. Care must be taken to assure that the target array is large enough to
hold the number of characters in the specified substring.

The following program demonstrates getChars( ):

```
class getCharsDemo {
public static void main(String args[]) {
String s = "This is a demo of the getChars method.";
int start = 10;
int end = 14;
char buf[] = new char[end - start];
s.getChars(start, end, buf, 0);
System.out.println(buf);
}
}
```

Here is the output of this program:

```
demo
```

getBytes( )

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

```
byte[ ] getBytes( )
```

Other forms of getBytes( ) are also available. getBytes( ) is most useful when you are exporting a String value into an environment that does not support 16-bit Unicode characters. For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

toCharArray( )

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

```
char[ ] toCharArray( )
```

This function is provided as a convenience, since it is possible to use getChars( ) to achieve the same result.

String Comparison

The String class includes a number of methods that compare strings or substrings within strings. Several are examined here.

equals( ) and equalsIgnoreCase( )

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

```
boolean equals(Object str)
```

Here, str is the String object being compared with the invoking String object. It returns true if the strings contain the same characters in the same order, and false otherwise. The comparison is case-sensitive. To perform a comparison that ignores case differences, call equalsIgnoreCase( ). When it compares two strings, it considers A-Z to be the same as a-z. It has this general form:

```
boolean equalsIgnoreCase(String str)
```

Here, str is the String object being compared with the invoking String object. It, too, returns true if the strings contain the same characters in the same order, and false otherwise.
Here is an example that demonstrates equals( ) and equalsIgnoreCase( ):

```
// Demonstrate equals() and equalsIgnoreCase().
class equalsDemo {
public static void main(String args[]) {
String s1 = "Hello";
String s2 = "Hello";
String s3 = "Good-bye";
String s4 = "HELLO";
System.out.println(s1 + " equals " + s2 + " -> " +
s1.equals(s2));
System.out.println(s1 + " equals " + s3 + " -> " +
s1.equals(s3));
System.out.println(s1 + " equals " + s4 + " -> " +
s1.equals(s4));
System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +
s1.equalsIgnoreCase(s4));
}
}
```

The output from the program is shown here:

```
Hello equals Hello -> true
Hello equals Good-bye -> false
Hello equals HELLO -> false
Hello equalsIgnoreCase HELLO -> true
```

regionMatches( )

The regionMatches( ) method compares a specific region inside a string with another specific region in another string. There is an overloaded form that allows you to ignore case in such comparisons. Here are the general forms for these two methods:

```
boolean regionMatches(int startIndex, String str2,
int str2StartIndex, int numChars)
boolean regionMatches(boolean ignoreCase,
int startIndex, String str2,
int str2StartIndex, int numChars)
```

For both versions, startIndex specifies the index at which the region begins within the invoking String object. The String being compared is specified by str2. The index at which the comparison will start within str2 is specified by str2StartIndex. The length of the substring being compared is passed in numChars. In the second version, if ignoreCase is true, the case of the characters is ignored. Otherwise, case is significant.
startsWith( ) and endsWith( )
String defines two methods that are, more or less, specialized forms of regionMatches( ). The startsWith( ) method determines whether a given String begins with a specified string. Conversely, endsWith( ) determines whether the String in question ends with a specified
string. They have the following general forms:

```
boolean startsWith(String str)
boolean endsWith(String str)
```

Here, str is the String being tested. If the string matches, true is returned. Otherwise, false is returned. For example,
"Foobar".endsWith("bar")
and
"Foobar".startsWith("Foo")
are both true.
A second form of startsWith( ), shown here, lets you specify a starting point:

```
boolean startsWith(String str, int startIndex)
```

Here, startIndex specifies the index into the invoking string at which point the search will begin. For example, "Foobar".startsWith("bar", 3) returns true.
equals( ) Versus ==

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

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

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

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

compareTo( )

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

int compareTo(String str)

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

Here is a sample program that sorts an array of strings. The program uses compareTo( ) to determine sort ordering for a bubble sort:

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

```
// A bubble sort for Strings.
class SortString {
static String arr[] = {
"Now", "is", "the", "time", "for", "all", "good", "men",
"to", "come", "to", "the", "aid", "of", "their", "country"
};
public static void main(String args[]) {
for(int j = 0; j < arr.length; j++) {
for(int i = j + 1; i < arr.length; i++) {
if(arr[i].compareTo(arr[j]) < 0) {
String t = arr[j];
arr[j] = arr[i];
arr[i] = t;
}
}
System.out.println(arr[j]);
}
}
}
```

The output of this program is the list of words:

```
Now
aid
all
come
country
for
good
is
men
of
the
the
their
time
to
to
```

As you can see from the output of this example, compareTo( ) takes into account uppercase and lowercase letters. The word "Now" came out before all the others because it begins with an uppercase letter, which means it has a lower value in the ASCII character set.

If you want to ignore case differences when comparing two strings, use compareToIgnoreCase( ), as shown here:

```
int compareToIgnoreCase(String str)
```

This method returns the same results as compareTo( ), except that case differences are ignored. You might want to try substituting it into the previous program. After doing so, "Now" will no longer be first.

Searching Strings

The String class provides two methods that allow you to search a string for a specified character or substring:

- indexOf( ) Searches for the first occurrence of a character or substring.
- lastIndexOf( ) Searches for the last occurrence of a character or substring.

These two methods are overloaded in several different ways. In all cases, the methods return the index at which the character or substring was found, or –1 on failure. To search for the first occurrence of a character, use

```
int indexOf(int ch)
```

To search for the last occurrence of a character, use

```
int lastIndexOf(int ch)
```

Here, ch is the character being sought.

To search for the first or last occurrence of a substring, use

```
int indexOf(String str)
int lastIndexOf(String str)
```

Here, str specifies the substring.

You can specify a starting point for the search using these forms:

```
int indexOf(int ch, int startIndex)
int lastIndexOf(int ch, int startIndex)
int indexOf(String str, int startIndex)
int lastIndexOf(String str, int startIndex)
```

Here, startIndex specifies the index at which point the search begins. For indexOf( ), the search runs from startIndex to the end of the string. For lastIndexOf( ), the search runs from startIndex to zero.

The following example shows how to use the various index methods to search inside of a String:

```
// Demonstrate indexOf() and lastIndexOf().
class indexOfDemo {
public static void main(String args[]) {
String s = "Now is the time for all good men " +
"to come to the aid of their country.";
System.out.println(s);
System.out.println("indexOf(t) = " +
s.indexOf('t'));
System.out.println("lastIndexOf(t) = " +
s.lastIndexOf('t'));
System.out.println("indexOf(the) = " +
s.indexOf("the"));
System.out.println("lastIndexOf(the) = " +
s.lastIndexOf("the"));
System.out.println("indexOf(t, 10) = " +
s.indexOf('t', 10));
System.out.println("lastIndexOf(t, 60) = " +
s.lastIndexOf('t', 60));
System.out.println("indexOf(the, 10) = " +
s.indexOf("the", 10));
System.out.println("lastIndexOf(the, 60) = " +
s.lastIndexOf("the", 60));
}
```

Here is the output of this program:

```
Now is the time for all good men to come to the aid of their country.
indexOf(t) = 7
lastIndexOf(t) = 65
indexOf(the) = 7
lastIndexOf(the) = 55
indexOf(t, 10) = 11
lastIndexOf(t, 60) = 55
indexOf(the, 10) = 44
lastIndexOf(the, 60) = 55
```

Modifying a String

Because String objects are immutable, whenever you want to modify a String, you must either copy it into a StringBuffer or StringBuilder, or use a String method that constructs a new copy of the string with your modifications complete. A sampling of these methods are described here.

substring( )

You can extract a substring using substring( ). It has two forms. The first is String substring(int startIndex). Here, startIndex specifies the index at which the substring will begin. This form returns a copy of the substring that begins at startIndex and runs to the end of the invoking string. The second form of substring( ) allows you to specify both the beginning and ending index of the substring:

String substring(int startIndex, int endIndex)

Here, startIndex specifies the beginning index, and endIndex specifies the stopping point. The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

The following program uses substring( ) to replace all instances of one substring wit another within a string:

```
// Substring replacement.
class StringReplace {
public static void main(String args[]) {
String org = "This is a test. This is, too.";
String search = "is";
String sub = "was";
String result = "";
int i;
do { // replace all matching substrings
System.out.println(org);
i = org.indexOf(search);
if(i != -1) {
result = org.substring(0, i);
result = result + sub;
result = result + org.substring(i + search.length());
org = result;
}
} while(i != -1);
}
}
```
The output from this program is shown here:
```
This is a test. This is, too.
Thwas is a test. This is, too.
Thwas was a test. This is, too.
Thwas was a test. Thwas is, too.
Thwas was a test. Thwas was, too.
```
concat( )
You can concatenate two strings using concat( ), shown here:
String concat(String str)
This method creates a new object that contains the invoking string with the contents of str appended to the end. concat( ) performs the same function as +. For example,
```
String s1 = "one";
String s2 = s1.concat("two");
```
puts the string "onetwo" into s2. It generates the same result as the following sequence:
```
String s1 = "one";
String s2 = s1 + "two";
```
replace( )
The replace( ) method has two forms. The first replaces all occurrences of one character in the invoking string with another character. It has the following general form:
String replace(char original, char replacement)
Here, original specifies the character to be replaced by the character specified by replacement. The resulting string is returned. For example,
```
String s = "Hello".replace('l', 'w');
```
puts the string "Hewwo" into s.
The second form of replace( ) replaces one character sequence with another. It has this general form:
String replace(CharSequence original, CharSequence replacement)
trim( )
The trim( ) method returns a copy of the invoking string from which any leading an trailing whitespace has been removed. It has this general form:
String trim( )
Here is an example:
```
String s = " Hello World ".trim();
```
This puts the string "Hello World" into s.
The trim( ) method is quite useful when you process user commands. For example, the following program prompts the user for the name of a state and then displays that state's capital. It uses trim( ) to remove any leading or trailing whitespace that may have inadvertently been entered by the user.
```
// Using trim() to process commands.
import java.io.*;
class UseTrim {
public static void main(String args[])
throws IOException
{
// create a BufferedReader using System.in
BufferedReader br = new
BufferedReader(new InputStreamReader(System.in));
String str;
System.out.println("Enter 'stop' to quit.");
System.out.println("Enter State: ");
do {
str = br.readLine();
str = str.trim(); // remove whitespace
if(str.equals("Illinois"))
System.out.println("Capital is Springfield.");
else if(str.equals("Missouri"))
System.out.println("Capital is Jefferson City.");
else if(str.equals("California"))
System.out.println("Capital is Sacramento.");
else if(str.equals("Washington"))
System.out.println("Capital is Olympia.");
// ...
} while(!str.equals("stop"));
}
}
```

Data Conversion Using valueOf( )

The valueOf( ) method converts data from its internal format into a human-readable form. It is a static method that is overloaded within String for all of Java's built-in types so that each type can be converted properly into a string. valueOf( ) is also overloaded for type Object, so an object of any class type you create can also be used as an argument. (Recall that Object is a superclass for all classes.) Here are a few of its forms:

- static String valueOf(double num)
- static String valueOf(long num)
- static String valueOf(Object ob)
- static String valueOf(char chars[ ])

As discussed earlier, valueOf( ) is called when a string representation of some other type of data is needed—for example, during concatenation operations. You can call this method directly with any data type and get a reasonable String representation. All of the simple types are converted to their common String representation. Any object that you pass to valueOf( ) will return the result of a call to the object's toString( ) method. In fact, you could just call toString( ) directly and get the same result.

For most arrays, valueOf( ) returns a rather cryptic string, which indicates that it is an array of some type. For arrays of char, however, a String object is created that contains the characters in the char array. There is a special version of valueOf( ) that allows you to specify a subset of a char array. It has this general form:

static String valueOf(char chars[ ], int startIndex, int numChars)

Here, chars is the array that holds the characters, startIndex is the index into the array of characters at which the desired substring begins, and numChars specifies the length of the substring.

Changing the Case of Characters Within a String

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

String toLowerCase( )
String toUpperCase( )

Both methods return a String object that contains the uppercase or lowercase equivalent of the invoking String. The default locale governs the conversion in both cases.

Here is an example that uses toLowerCase( ) and toUpperCase( ):

```
// Demonstrate toUpperCase() and toLowerCase().
class ChangeCase {
public static void main(String args[])
{
String s = "This is a test.";
System.out.println("Original: " + s);
String upper = s.toUpperCase();
String lower = s.toLowerCase();
System.out.println("Uppercase: " + upper);
System.out.println("Lowercase: " + lower);
}
}
```

The output produced by the program is shown here:

```
Original: This is a test.
Uppercase: THIS IS A TEST.
Lowercase: this is a test.
```

One other point: Overloaded versions of toLowerCase( ) and toUpperCase( ) that let
you specify a Locale object to govern the conversion are also supplied. Specifying the locale can be quite important in some cases and can help internationalize your application.

Joining Strings

JDK 8 adds a new method to String called join( ). It is used to concatenate two or more strings, separating each string with a delimiter, such as a space or a comma. It has two forms. Its first is shown here:

static String join(CharSequence delim, CharSequence . . . strs)
Here, delim specifies the delimiter used to separate the character sequences specified by strs.

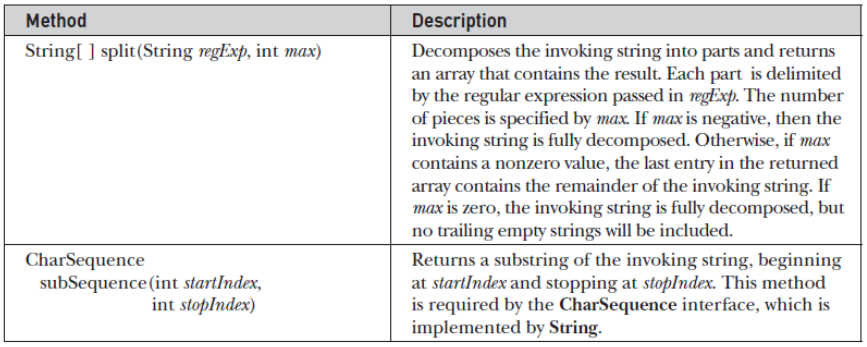
```
// Demonstrate the join() method defined by String.
class StringJoinDemo {
public static void main(String args[]) {
String result = String.join(" ", "Alpha", "Beta", "Gamma");
System.out.println(result);
result = String.join(", ", "John", "ID#: 569",
"E-mail: John@HerbSchildt.com");
System.out.println(result);
}
}
```

The output is shown here:

```
Alpha Beta Gamma
John, ID#: 569, E-mail: John@HerbSchildt.com
```

In the first call to join( ), a space is inserted between each string. In the second call, the delimiter is a comma followed by a space. This illustrates that the delimiter need not be just a single character. The second form of join( ) lets you join a list of strings obtained from an object that implements the Iterable interface. Iterable is implemented by the Collections Framework classes
Additional String Methods
In addition to those methods discussed earlier, String has many other methods, including those summarized in the following table:

Method	**Description**
int codePointAt(int *i*)	Returns the Unicode code point at the location specified by *i*.
int codePointBefore(int *i*)	Returns the Unicode code point at the location that precedes that specified by *i*.
int codePointCount(int *start*, int *end*)	Returns the number of code points in the portion of the invoking **String** that are between *start* and *end*–1.
boolean contains(CharSequence *str*)	Returns **true** if the invoking object contains the string specified by *str*. Returns **false** otherwise.
boolean contentEquals(CharSequence *str*)	Returns **true** if the invoking string contains the same string as *str*. Otherwise, returns **false**.
boolean contentEquals(StringBuffer *str*)	Returns **true** if the invoking string contains the same string as *str*. Otherwise, returns **false**.
static String format(String *fmtstr*, Object ... *args*)	Returns a string formatted as specified by *fmtstr*. (See Chapter 19 for details on formatting.)
static String format(Locale *loc*, String *fmtstr*, Object ... *args*)	Returns a string formatted as specified by *fmtstr*. Formatting is governed by the locale specified by *loc*. (See Chapter 19 for details on formatting.)
boolean isEmpty( )	Returns **true** if the invoking string contains no characters and has a length of zero.
boolean matches(string *regExp*)	Returns **true** if the invoking string matches the regular expression passed in *regExp*. Otherwise, returns **false**.
int offsetByCodePoints(int *start*, int *num*)	Returns the index within the invoking string that is *num* code points beyond the starting index specified by *start*.
String replaceFirst(String *regExp*, String *newStr*)	Returns a string in which the first substring that matches the regular expression specified by *regExp* is replaced by *newStr*.
String replaceAll(String *regExp*, String *newStr*)	Returns a string in which all substrings that match the regular expression specified by *regExp* are replaced by *newStr*.
String[ ] split(String *regExp*)	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in *regExp*.



Method	Description
String[ ] split(String *regExp*, int *max*)	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in *regExp*. The number of pieces is specified by *max*. If *max* is negative, then the invoking string is fully decomposed. Otherwise, if *max* contains a nonzero value, the last entry in the returned array contains the remainder of the invoking string. If *max* is zero, the invoking string is fully decomposed, but no trailing empty strings will be included.
CharSequence subSequence(int *startIndex*, int *stopIndex*)	Returns a substring of the invoking string, beginning at *startIndex* and stopping at *stopIndex*. This method is required by the **CharSequence** interface, which is implemented by **String**.

StringBuffer
StringBuffer supports a modifiable string. As you know, String represents fixed-length, immutable character sequences. In contrast, StringBuffer represents growable and writable character sequences. StringBuffer may have characters and substrings inserted in the middle or appended to the end. StringBuffer will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to
allow room for growth.

- StringBuffer Constructors
- StringBuffer defines these four constructors:
- StringBuffer( )
- StringBuffer(int size)
- StringBuffer(String str)
- StringBuffer(CharSequence chars)

The default constructor (the one with no parameters) reserves room for 16 characters without reallocation. The second version accepts an integer argument that explicitly sets the size of the buffer. The third version accepts a String argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation. StringBuffer allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time. Also, frequent reallocations can fragment memory. By allocating room for a few extra characters, StringBuffer reduces the number of reallocations that take place. The fourth constructor creates an object that contains the character sequence contained in chars and reserves room for 16 more characters.

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

```
int length( )
int capacity( )
```

Here is an example:

```
// StringBuffer length vs. capacity.
class StringBufferDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("Hello");
System.out.println("buffer = " + sb);
System.out.println("length = " + sb.length());
System.out.println("capacity = " + sb.capacity());
}
}
```

Here is the output of this program, which shows how StringBuffer reserves extra space for additional manipulations:

```
buffer = Hello
length = 5
capacity = 21
```

Since sb is initialized with the string "Hello" when it is created, its length is 5. Its capacity is 21 because room for 16 additional characters is automatically added.

ensureCapacity( )

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

```
void ensureCapacity(int minCapacity)
```

Here, minCapacity specifies the minimum size of the buffer. (A buffer larger than minCapacity may be allocated for reasons of efficiency.)

setLength( )

To set the length of the string within a StringBuffer object, use setLength( ). Its general form is shown here:

```
void setLength(int len)
```

Here, len specifies the length of the string. This value must be nonnegative When you increase the size of the string, null characters are added to the end. If yo call setLength( ) with a value less than the current value returned by length( ), then the characters stored beyond the new length will be lost. The setCharAtDemo sample program
in the following section uses setLength( ) to shorten a StringBuffer.

charAt( ) and setCharAt( )

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

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

For charAt( ), where specifies the index of the character being obtained. For setCharAt( ), where specifies the index of the character being set, and ch specifies the new value of that character. For both methods, where must be nonnegative and must not specify a location beyond the end of the string.

The following example demonstrates charAt( ) and setCharAt( ):

```
// Demonstrate charAt() and setCharAt().
class setCharAtDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("Hello");
System.out.println("buffer before = " + sb);
System.out.println("charAt(1) before = " + sb.charAt(1));
sb.setCharAt(1, 'i');
sb.setLength(2);
System.out.println("buffer after = " + sb);
System.out.println("charAt(1) after = " + sb.charAt(1));
}
}
```

Here is the output generated by this program:

```
buffer before = Hello
charAt(1) before = e
buffer after = Hi
charAt(1) after = i
```

getChars( )

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

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

Here, sourceStart specifies the index of the beginning of the substring, and sourceEnd specifies an index that is one past the end of the desired substring. This means that the substring contains the characters from sourceStart through sourceEnd–1. The array that will receive the characters is specified by target. The index within target at which the substring will be copied is passed in targetStart. Care must be taken to assure that the target array is large enough to hold the number of characters in the specified substring.

append( )

The append( ) method concatenates the string representation of any other type of data to the end of the invoking StringBuffer object. It has several overloaded versions. Here are a few of its forms:

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

The string representation of each parameter is obtained, often by calling String.valueOf( ). The result is appended to the current StringBuffer object. The buffer itself is returned by each version of append( ). This allows subsequent calls to be chained together, as shown in the following example:

```
// Demonstrate append().
```

```
class appendDemo {
public static void main(String args[]) {
String s;
int a = 42;
StringBuffer sb = new StringBuffer(40);
s = sb.append("a = ").append(a).append("!").toString();
System.out.println(s); } }
```

The output of this example is shown here:

```
a = 42!
```

insert( )

The insert( ) method inserts one string into another. It is overloaded to accept values of al the primitive types, plus Strings, Objects, and CharSequences. Like append( ), it obtains the string representation of the value it is called with. This string is then inserted into the invoking
StringBuffer object. These are a few of its forms:

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

Here, index specifies the index at which point the string will be inserted into the invoking StringBuffer object.
The following sample program inserts "like" between "I" and "Java":

```
// Demonstrate insert().
class insertDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("I Java!");
sb.insert(2, "like ");
System.out.println(sb);}}
```

The output of this example is shown here:

```
I like Java!
```

reverse( )

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

```
StringBuffer reverse( )
```

This method returns the reverse of the object on which it was called. The following program demonstrates reverse( ):

```
// Using reverse() to reverse a StringBuffer.
class ReverseDemo {
public static void main(String args[]) {
StringBuffer s = new StringBuffer("abcdef");
System.out.println(s);
s.reverse();
System.out.println(s); } }
```

Here is the output produced by the program:

```
abcdef
fedcba
```

delete( ) and deleteCharAt( )

You can delete characters within a StringBuffer by using the methods delete( ) and deleteCharAt( ). These methods are shown here:

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

The delete( ) method deletes a sequence of characters from the invoking object. Here, startIndex specifies the index of the first character to remove, and endIndex specifies an index one past the last character to remove. Thus, the substring deleted runs from startIndex to endIndex–1. The resulting StringBuffer object is returned.
The deleteCharAt( ) method deletes the character at the index specified by loc. It returns the resulting StringBuffer object.
Here is a program that demonstrates the delete( ) and deleteCharAt( ) methods:

```
// Demonstrate delete() and deleteCharAt()
class deleteDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("This is a test.");
sb.delete(4, 7);
System.out.println("After delete: " + sb);
sb.deleteCharAt(0);
System.out.println("After deleteCharAt: " + sb);      } }
```

The following output is produced:

```
After delete: This a test.
After deleteCharAt: his a test.
```

replace( ): You can replace one set of characters with another set inside a StringBuffer object by calling replace( ). Its signature is shown here:

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

The substring being replaced is specified by the indexes startIndex and endIndex. Thus, the substring at startIndex through endIndex–1 is replaced. The replacement string is passed in str. The resulting StringBuffer object is returned.

The following program demonstrates replace( ):

```
// Demonstrate replace()
class replaceDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("This is a test.");
sb.replace(5, 7, "was");
System.out.println("After replace: " + sb);
}
}
```

Here is the output:

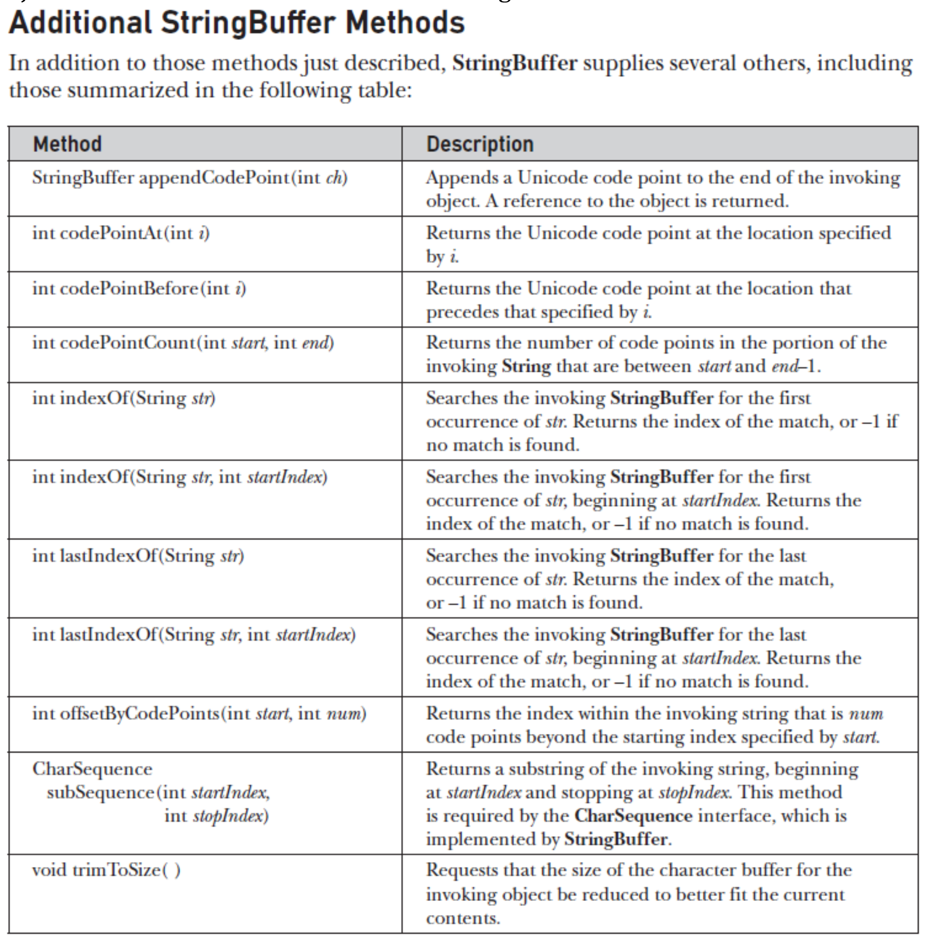
```
After replace: This was a test.
```

substring( )

You can obtain a portion of a StringBuffer by calling substring( ). It has the following two forms:

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

The first form returns the substring that starts at startIndex and runs to the end of the invoking StringBuffer object. The second form returns the substring that starts at startIndex and runs through endIndex–1.



Additional StringBuffer Methods

In addition to those methods just described, **StringBuffer** supplies several others, including those summarized in the following table:

Method	Description
StringBuffer appendCodePoint(int *ch*)	Appends a Unicode code point to the end of the invoking object. A reference to the object is returned.
int codePointAt(int *i*)	Returns the Unicode code point at the location specified by *i*.
int codePointBefore(int *i*)	Returns the Unicode code point at the location that precedes that specified by *i*.
int codePointCount(int *start*, int *end*)	Returns the number of code points in the portion of the invoking **String** that are between *start* and *end*–1.
int indexOf(String *str*)	Searches the invoking **StringBuffer** for the first occurrence of *str*. Returns the index of the match, or –1 if no match is found.
int indexOf(String *str*, int *startIndex*)	Searches the invoking **StringBuffer** for the first occurrence of *str*, beginning at *startIndex*. Returns the index of the match, or –1 if no match is found.
int lastIndexOf(String *str*)	Searches the invoking **StringBuffer** for the last occurrence of *str*. Returns the index of the match, or –1 if no match is found.
int lastIndexOf(String *str*, int *startIndex*)	Searches the invoking **StringBuffer** for the last occurrence of *str*, beginning at *startIndex*. Returns the index of the match, or –1 if no match is found.
int offsetByCodePoints(int *start*, int *num*)	Returns the index within the invoking string that is *num* code points beyond the starting index specified by *start*.
CharSequence subSequence(int *startIndex*, int *stopIndex*)	Returns a substring of the invoking string, beginning at *startIndex* and stopping at *stopIndex*. This method is required by the **CharSequence** interface, which is implemented by **StringBuffer**.
void trimToSize( )	Requests that the size of the character buffer for the invoking object be reduced to better fit the current contents.

Additional StringBuffer Methods
In addition to those methods just described, StringBuffer supplies several others, including those summarized in the following table:

The following program demonstrates indexOf( ) and lastIndexOf( ):

```
class IndexOfDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("one two one");
int i;
i = sb.indexOf("one");
System.out.println("First index: " + i);
i = sb.lastIndexOf("one");
System.out.println("Last index: " + i);}}
```

The output is shown here:

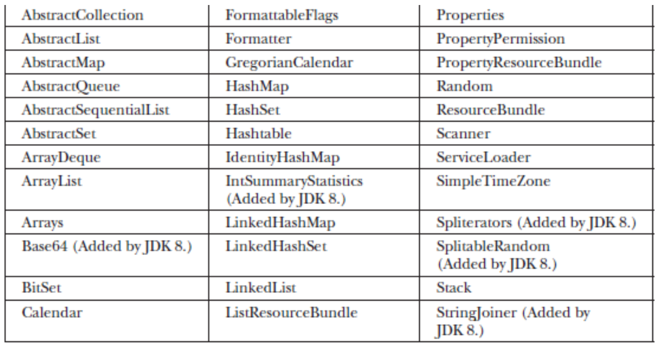
```
First index: 0
Last index: 8
```

StringBuilder

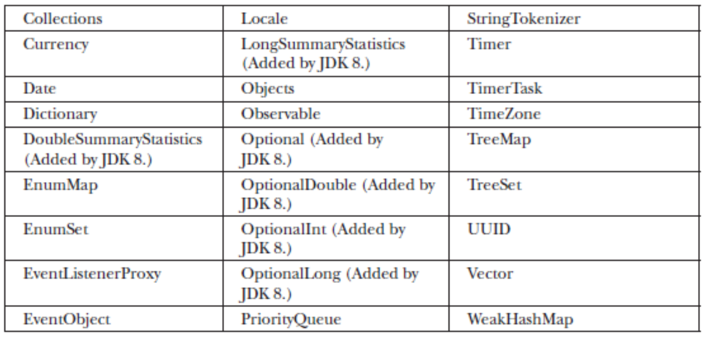
StringBuilder is similar to StringBuffer except for one important difference: it is not synchronized, which means that it is not thread-safe. The advantage of StringBuilder is faster performance. However, in cases in which a mutable string will be accessed by multiple threads, and no external synchronization is employed, you must use StringBuffer rathe than StringBuilder.

Java.util

This important package contains a large assortment of classes and interfaces that support a broad range of functionality. For example, java.util has classes that generate pseudorandom numbers, manage date and time, observe events, manipulate sets of bits, tokenize strings, and handle formatted data. 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 top-level classes:

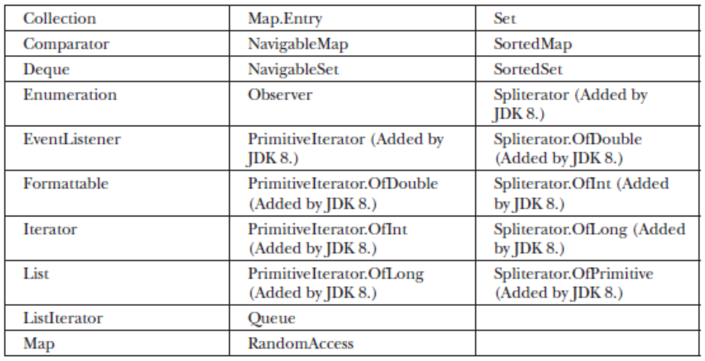


AbstractCollection	FormattableFlags	Properties
AbstractList	Formatter	PropertyPermission
AbstractMap	GregorianCalendar	PropertyResourceBundle
AbstractQueue	HashMap	Random
AbstractSequentialList	HashSet	ResourceBundle
AbstractSet	Hashtable	Scanner
ArrayDeque	IdentityHashMap	ServiceLoader
ArrayList	IntSummaryStatistics (Added by JDK 8.)	SimpleTimeZone
Arrays	LinkedHashMap	Spliterators (Added by JDK 8.)
Base64 (Added by JDK 8.)	LinkedHashSet	SplitableRandom (Added by JDK 8.)
BitSet	LinkedList	Stack
Calendar	ListResourceBundle	StringJoiner (Added by JDK 8.)

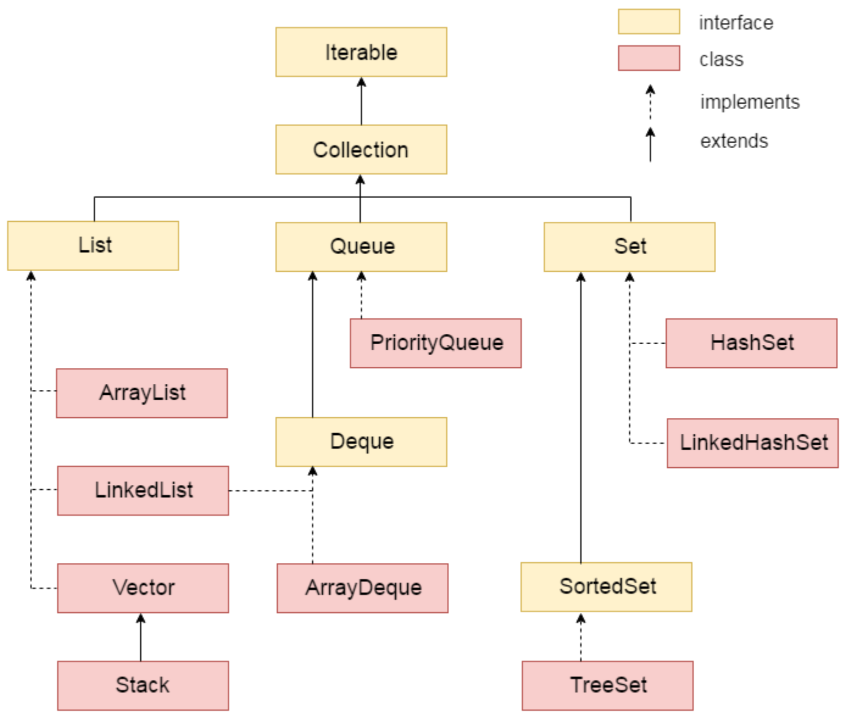


Collections	Locale	StringTokenizer
Currency	LongSummaryStatistics (Added by JDK 8.)	Timer
Date	Objects	TimerTask
Dictionary	Observable	TimeZone
DoubleSummaryStatistics (Added by JDK 8.)	Optional (Added by JDK 8.)	TreeMap
EnumMap	OptionalDouble (Added by JDK 8.)	TreeSet
EnumSet	OptionalInt (Added by JDK 8.)	UUID
EventListenerProxy	OptionalLong (Added by JDK 8.)	Vector
EventObject	PriorityQueue	WeakHashMap

The interfaces defined by **java.util** are shown next:



Collection	Map.Entry	Set
Comparator	NavigableMap	SortedMap
Deque	NavigableSet	SortedSet
Enumeration	Observer	Spliterator (Added by JDK 8.)
EventListener	PrimitiveIterator (Added by JDK 8.)	Spliterator.OfDouble (Added by JDK 8.)
Formattable	PrimitiveIterator.OfDouble (Added by JDK 8.)	Spliterator.OfInt (Added by JDK 8.)
Iterator	PrimitiveIterator.OfInt (Added by JDK 8.)	Spliterator.OfLong (Added by JDK 8.)
List	PrimitiveIterator.OfLong (Added by JDK 8.)	Spliterator.OfPrimitive (Added by JDK 8.)
ListIterator	Queue	
Map	RandomAccess	



What are we going to learn in Java Collections Framework

1. ArrayList class
2. LinkedList class
3. List interface
4. HashSet class
5. LinkedHashSet class
6. TreeSet class
7. PriorityQueue class
8. Map interface
9. HashMap class
10. LinkedHashMap class
11. TreeMap class
12. Hashtable class
13. Sorting
14. Comparable interface
15. Comparator interface
16. Properties class in Java

Collections Overview
The Java Collections Framework standardizes the way in which groups of objects are handled by your programs. Collections were not part of the original Java release, but were added by J2SE 1.2. 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. Several standard implementations (such as LinkedList, HashSet, and TreeSet) of these interfaces are provided that you may use as-is. You may also implement your own collection, if you choose. Various special-purpose implementations are created for your convenience, and some partial implementations are provided that make creating your own collection class easier. Finally, mechanisms were added that allow the integration of standard arrays into the Collections Framework.

Algorithms are another important part of the collection mechanism. Algorithms operate on collections and are defined as static methods within the Collections class. Thus, they are available for all collections. Each collection class need not implement its own versions. The algorithms provide a standard means of manipulating collections.
Another item closely associated with the Collections Framework is the Iterator interface. An iterator offers a general-purpose, standardized way of accessing the elements within a collection, one at a time. Thus, an iterator provides a means of enumerating the contents of a collection. Because each collection provides an iterator, the elements of any collection class can be accessed through the methods defined by Iterator. Thus, with only small changes, the code that cycles through a set can also be used to cycle through a list, for example. JDK 8 adds another type of iterator called a spliterator.

In brief, spliterators are iterators that provide support for parallel iteration. The interfaces that support spliterators
are Spliterator and several nested interfaces that support primitive types. JDK 8 also adds iterator interfaces designed for use with primitive types, such as PrimitiveIterator and PrimitiveIterator.OfDouble.
In addition to collections, the framework defines several map interfaces and classes. Maps store key/value pairs. Although maps are part of the Collections Framework, they are not "collections" in the strict use of the term. You can, however, obtain a collection-view of a map. Such a view contains the elements from the map stored in a collection. Thus, you can process the contents of a map as a collection, if you choose. The collection mechanism was retrofitted to some of the original classes defined by java.util so that they too could be integrated into the new system. It is important to understand that although the addition of collections altered the architecture of many
of the original utility classes, it did not cause the deprecation of any. Collections simply provide a better way of doing several things.

JDK 5 Changed the Collections Framework

When JDK 5 was released, some fundamental changes were made to the Collections Framework that significantly increased its power and streamlined its use. These changes include the addition of generics, autoboxing/unboxing, and the for-each style for loop. Although JDK 8 is three major Java releases after JDK 5, the effects of the JDK 5 features were so profound that they still warrant special attention. The main reason is that you may encounter pre-JDK 5 code. Understanding the effects and reasons for the changes is important if you will be maintaining or updating older code.

Generics Fundamentally Changed the Collections Framework

The addition of generics caused a significant change to the Collections Framework because the entire Collections Framework was 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 affected every part of the Collections Framework.Generics added the one feature that collections had been missing: type safety. Prior to generics, all collections stored Object references, which meant that any collection could store any type of object. Thus, it was possible to accidentally store incompatible types in a collection. Doing so could result in run-time type mismatch errors. With generics, it is possible to explicitly state the type of data being stored, and run-time type mismatch
errors can be avoided.
Although the addition of generics changed the declarations of most of its classes and interfaces, and several of their methods, overall, the Collections Framework still works the same as it did prior to generics. Of course, to gain the advantages that generics bring collections, older code will need to be rewritten. This is also

important because pre-generics code will generate warning messages when compiled by a modern Java compiler. To eliminate these warnings, you will need to add type information to all your collections code.

Autoboxing Facilitates the Use of Primitive Types

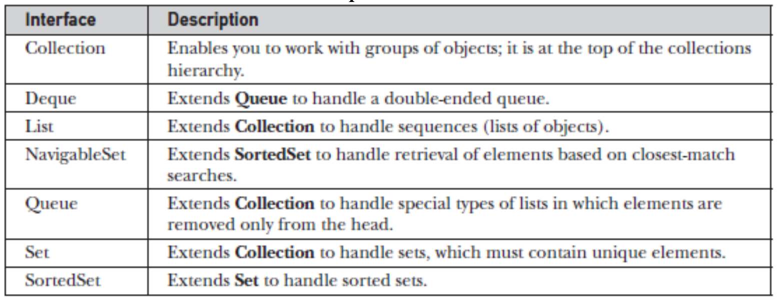
Autoboxing/unboxing facilitates the storing of primitive types in collections. As you will see, a collection can store only references, not primitive values. In the past, if you wanted to store a primitive value, such as an int, in a collection, you had to manually box it into its type wrapper. When the value was retrieved, it needed to be manually unboxed (by using an explicit cast) into its proper primitive type. Because of autoboxing/unboxing, Java can
automatically perform the proper boxing and unboxing needed when storing or retrieving primitive types. There is no need to manually perform these operations.

The For-Each Style for Loop

All collection classes in the Collections Framework were retrofitted to implement the Iterable interface, which means that a collection can be cycled through by use of the foreach style for loop. In the past, cycling through a collection required the use of an iterator, with the programmer manually constructing the loop. Although iterators are still needed for some uses, in many cases, iterator-based loops can be replaced by for loops

The Collection Interfaces

The Collections Framework defines several core interfaces. This section provides an overview of each interface. Beginning with the collection interfaces is necessary because they determine the fundamental nature of the collection classes. Put differently, the concrete classes simply provide different implementations of the standard interfaces. The interfaces that underpin collections are summarized in the following table:

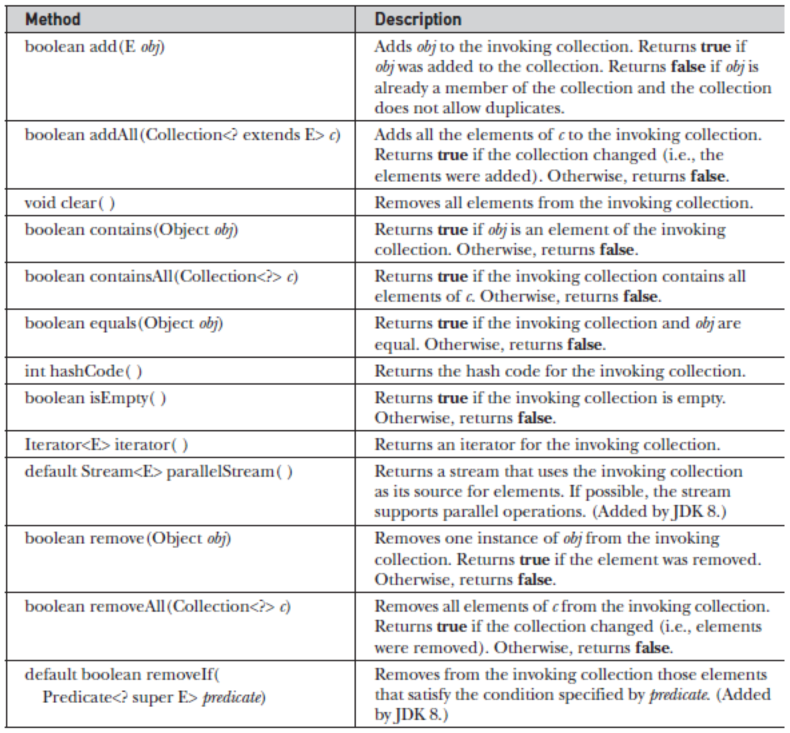


Interface	Description
Collection	Enables you to work with groups of objects; it is at the top of the collections hierarchy.
Deque	Extends **Queue** to handle a double-ended queue.
List	Extends **Collection** to handle sequences (lists of objects).
NavigableSet	Extends **SortedSet** to handle retrieval of elements based on closest-match searches.
Queue	Extends **Collection** to handle special types of lists in which elements are removed only from the head.
Set	Extends **Collection** to handle sets, which must contain unique elements.
SortedSet	Extends **Set** to handle sorted sets.

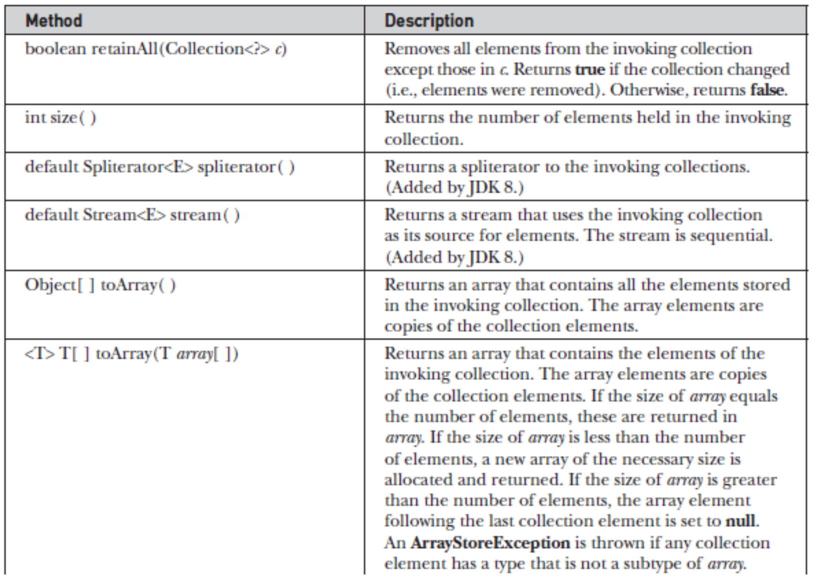
In addition to the collection interfaces, collections also use the Comparator, RandomAccess, Iterator, and ListIterator interfaces, which are described in depth later

Beginning with JDK 8, Spliterator can also be used. Briefly, Comparator defines how two objects are compared; Iterator, ListIterator, and Spliterator enumerate the objects within a collection. By implementing RandomAccess, a list indicates that it supports efficient, random access to its elements.

To provide the greatest flexibility in their use, the collection interfaces allow some methods to be optional. The optional methods enable you to modify the contents of a collection. Collections that support these methods are called modifiable. Collections that do not allow their contents to be changed are called unmodifiable. If an attempt is made to use one of these methods on an unmodifiable collection, an UnsupportedOperationException is thrown. All the built-in collections are modifiable.



Method	Description
boolean add(E *obj*)	Adds *obj* to the invoking collection. Returns **true** if *obj* was added to the collection. Returns **false** if *obj* is already a member of the collection and the collection does not allow duplicates.
boolean addAll(Collection<? extends E> *c*)	Adds all the elements of *c* to the invoking collection. Returns **true** if the collection changed (i.e., the elements were added). Otherwise, returns **false**.
void clear( )	Removes all elements from the invoking collection.
boolean contains(Object *obj*)	Returns **true** if *obj* is an element of the invoking collection. Otherwise, returns **false**.
boolean containsAll(Collection<?> *c*)	Returns **true** if the invoking collection contains all elements of *c*. Otherwise, returns **false**.
boolean equals(Object *obj*)	Returns **true** if the invoking collection and *obj* are equal. Otherwise, returns **false**.
int hashCode( )	Returns the hash code for the invoking collection.
boolean isEmpty( )	Returns **true** if the invoking collection is empty. Otherwise, returns **false**.
Iterator<E> iterator( )	Returns an iterator for the invoking collection.
default Stream<E> parallelStream( )	Returns a stream that uses the invoking collection as its source for elements. If possible, the stream supports parallel operations. (Added by JDK 8.)
boolean remove(Object *obj*)	Removes one instance of *obj* from the invoking collection. Returns **true** if the element was removed. Otherwise, returns **false**.
boolean removeAll(Collection<?> *c*)	Removes all elements of *c* from the invoking collection. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
default boolean removeIf(Predicate<? super E> *predicate*)	Removes from the invoking collection those elements that satisfy the condition specified by *predicate*. (Added by JDK 8.)



Method	Description
boolean retainAll(Collection<?> *c*)	Removes all elements from the invoking collection except those in *c*. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
int size( )	Returns the number of elements held in the invoking collection.
default Spliterator<E> spliterator( )	Returns a spliterator to the invoking collections. (Added by JDK 8.)
default Stream<E> stream( )	Returns a stream that uses the invoking collection as its source for elements. The stream is sequential. (Added by JDK 8.)
Object[ ] toArray( )	Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.
<T> T[ ] toArray(T *array*[ ])	Returns an array that contains the elements of the invoking collection. The array elements are copies of the collection elements. If the size of *array* equals the number of elements, these are returned in *array*. If the size of *array* is less than the number of elements, a new array of the necessary size is allocated and returned. If the size of *array* is greater than the number of elements, the array element following the last collection element is set to **null**. An **ArrayStoreException** is thrown if any collection element has a type that is not a subtype of *array*.

The Collection Interface
The following sections examine the collection interfaces.
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. This means that all collections can be cycled through by use of the foreach style for loop. (Recall that only classes that implement Iterable can be cycled through by the for.)

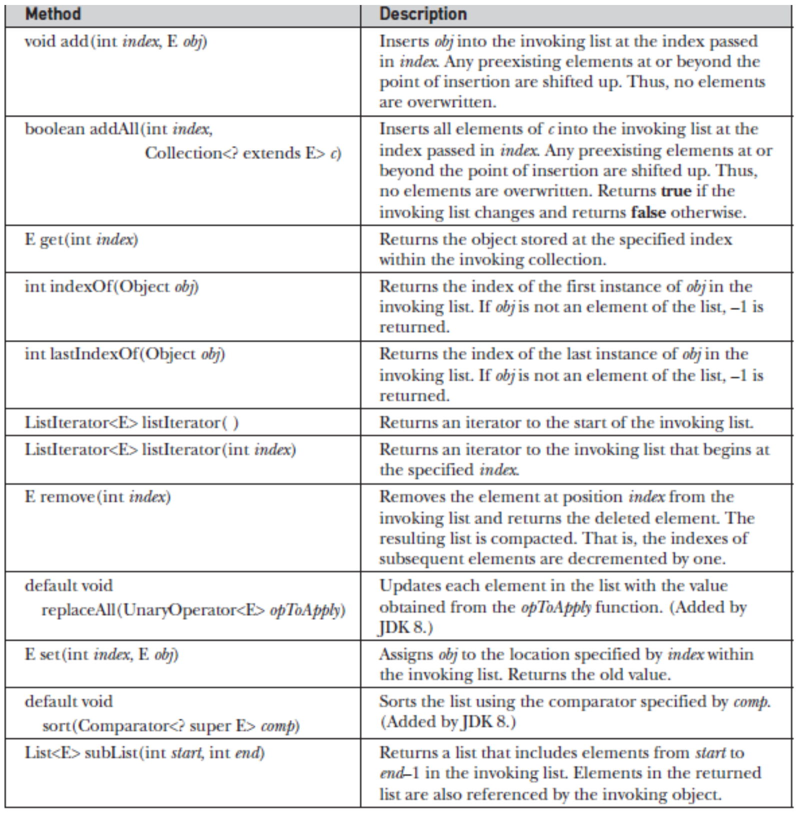
Collection declares the core methods that all collections will have. These methods are summarized in Because all collections implement Collection, familiarity with its methods is necessary for a clear understanding of the framework. Several of these methods can throw an UnsupportedOperationException. As explained, this occurs if a collection cannot be modified. A ClassCastException is generated when one object is incompatible with another, such as when an attempt is made to add an incompatible object to a collection. A NullPointerException is thrown if an attempt is made to store a null object and null elements are not allowed in the collection. An IllegalArgumentException is thrown if an invalid argument is used. An IllegalStateException is thrown if an attempt is made to add an element to a fixed-length collection that is full.

Objects are added to a collection by calling add( ). Notice that add( ) takes an argument of type E, which means that objects added to a collection must be compatible with the type of data expected by the collection. You can add the entire contents of one collection to another by calling addAll( ).
You can remove an object by using remove( ). To remove a group of objects, call removeAll( ). You can remove all elements except those of a specified group by calling retainAll( ). Beginning with JDK 8, to remove an element only if it statisfies some condition, you can use removeIf( ).

To empty a collection, call clear( ). You can determine whether a collection contains a specific object by calling contains( ). To determine whether one collection contains all the members of another, call containsAll( ). You can determine when a collection is empty by calling isEmpty( ). The number of elements currently held in a collection can be determined by calling size( ).
The toArray( ) methods return an array that contains the elements stored in the invoking collection. The first returns an array of Object. The second returns an array of elements that have the same type as the array specified as a parameter. Normally, the second form is more convenient because it returns the desired array type. These methods
are more important than it might at first seem. Often, processing the contents of a collection by using array-like syntax is advantageous. By providing a pathway between collections and arrays, you can have the best of both worlds.

Two collections can be compared for equality by calling equals( ). The precise meaning of "equality" may differ from collection to collection. For example, you can implement equals( ) so that it compares the values of elements stored in the collection. Alternatively, equals( ) can compare references to those elements. Another important method is iterator( ), which returns an iterator to a collection. The new spliterator( ) method returns a spliterator to the collection. Iterators are frequently used when working with collections. Finally, the stream( ) and parallelStream( ) methods return a Stream that uses the collection as a source of elements.

The List Interface
The List interface extends Collection and declares the behavior of a collection that stores a sequence of elements. Elements can be inserted or accessed by their position in the list, using a zero-based index. A list may contain duplicate elements. List is a generic interface that has this declaration:
interface List<E>
Here, E specifies the type of objects that the list will hold. In addition to the methods defined by Collection, List defines some of its own, which are summarized in Table Note again that several of these methods will throw an
UnsupportedOperationException if the list cannot be modified, and a ClassCastException is generated when one object is incompatible with another, such as when an attempt is made to add an incompatible object to a list. Also, several methods will throw an IndexOutOfBoundsException if an invalid index is used. A NullPointerException is
thrown if an attempt is made to store a null object and null elements are not allowed in the list. An IllegalArgumentException is thrown if an invalid argument is used.
To the versions of add( ) and addAll( ) defined by Collection, List adds the methods add(int, E) and addAll(int, Collection). These methods insert elements at the specified index. Also, the semantics of add(E) and addAll(Collection) defined by Collection are changed by List so that they add elements to the end of the list. You can modify each element in the collection by using replaceAll( ).
To obtain the object stored at a specific location, call get( ) with the index of the object. To assign a value to an element in the list, call set( ), specifying the index of the object to be changed. To find the index of an object, use indexOf( ) or lastIndexOf( ).
You can obtain a sublist of a list by calling subList( ), specifying the beginning and ending indexes of the sublist. As you can imagine, subList( ) makes list processing quite convenient. One way to sort a list is with the sort( ) method defined by List.
The Set Interface
The Set interface defines a set. It extends Collection and specifies the behavior of a collection that does not allowduplicate elements. Therefore, the add( ) method returns

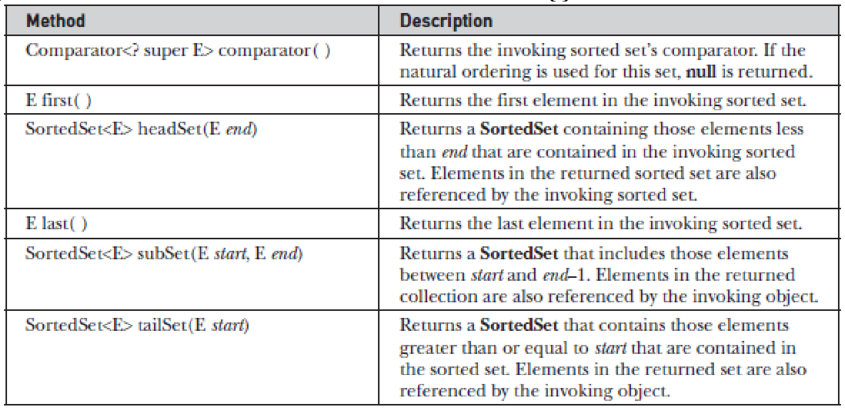


Method	Description
void add(int *index*, E *obj*)	Inserts *obj* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
boolean addAll(int *index*, Collection<? extends E> *c*)	Inserts all elements of *c* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns **true** if the invoking list changes and returns **false** otherwise.
E get(int *index*)	Returns the object stored at the specified index within the invoking collection.
int indexOf(Object *obj*)	Returns the index of the first instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.
int lastIndexOf(Object *obj*)	Returns the index of the last instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.
ListIterator<E> listIterator( )	Returns an iterator to the start of the invoking list.
ListIterator<E> listIterator(int *index*)	Returns an iterator to the invoking list that begins at the specified *index*.
E remove(int *index*)	Removes the element at position *index* from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.
default void replaceAll(UnaryOperator<E> *opToApply*)	Updates each element in the list with the value obtained from the *opToApply* function. (Added by JDK 8.)
E set(int *index*, E *obj*)	Assigns *obj* to the location specified by *index* within the invoking list. Returns the old value.
default void sort(Comparator<? super E> *comp*)	Sorts the list using the comparator specified by *comp*. (Added by JDK 8.)
List<E> subList(int *start*, int *end*)	Returns a list that includes elements from *start* to *end*–1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

false if an attempt is made to add duplicate elements to a set. It does not specify any additional methods of its own. Set is a generic interface that has this declaration: interface Set<E> Here, E specifies the type of objects that the set will hold.

The SortedSet Interface

The SortedSet interface extends Set and declares the behavior of a set sorted in ascending order. SortedSet is a generic interface that has this declaration:
interface SortedSet<E>
Here, E specifies the type of objects that the set will hold. In addition to those methods provided by Set, the SortedSet interface declares the methods summarized in Table . Several methods throw a NoSuchElementException
when no items are contained in the invoking set. A ClassCastException is thrown when an object is incompatible with the elements in a set. A NullPointerException is thrown if an attempt is made to use a null object and null is not allowed in the set. An IllegalArgumentException is thrown if an invalid argument is used.
SortedSet defines several methods that make set processing more convenient. To obtain the first object in the set, call first( ). To get the last element, use last( ). You can obtain a subset of a sorted set by calling subSet( ), specifying the first and last object in the set. If you need the subset that starts with the first element in the set, use headSet( ). If you want the subset that ends the set, use tailSet( ).



Method	Description
Comparator<? super E> comparator( )	Returns the invoking sorted set's comparator. If the natural ordering is used for this set, **null** is returned.
E first( )	Returns the first element in the invoking sorted set.
SortedSet<E> headSet(E *end*)	Returns a **SortedSet** containing those elements less than *end* that are contained in the invoking sorted set. Elements in the returned sorted set are also referenced by the invoking sorted set.
E last( )	Returns the last element in the invoking sorted set.
SortedSet<E> subSet(E *start*, E *end*)	Returns a **SortedSet** that includes those elements between *start* and *end*–1. Elements in the returned collection are also referenced by the invoking object.
SortedSet<E> tailSet(E *start*)	Returns a **SortedSet** that contains those elements greater than or equal to *start* that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object.

The NavigableSet Interface
The NavigableSet interface extends SortedSet and declares the behavior of a collection that supports the retrieval of elements based on the closest match to a given value or values. NavigableSet is a generic interface that has this declaration: interface NavigableSet<E>
Here, E specifies the type of objects that the set will hold. In addition to the methods that it inherits from SortedSet, NavigableSet adds those summarized in Table .

Method	Description
E ceiling(E *obj*)	Searches the set for the smallest element *e* such that *e* >= *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
Iterator<E> descendingIterator( )	Returns an iterator that moves from the greatest to least. In other words, it returns a reverse iterator.
NavigableSet<E> descendingSet( )	Returns a **NavigableSet** that is the reverse of the invoking set. The resulting set is backed by the invoking set.
E floor(E *obj*)	Searches the set for the largest element *e* such that *e* <= *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
NavigableSet<E> headSet(E *upperBound*, boolean *incl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are less than *upperBound*. If *incl* is **true**, then an element equal to *upperBound* is included. The resulting set is backed by the invoking set.
E higher(E *obj*)	Searches the set for the largest element *e* such that *e* > *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
E lower(E *obj*)	Searches the set for the largest element *e* such that *e* < *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
E pollFirst( )	Returns the first element, removing the element in the process. Because the set is sorted, this is the element with the least value. **null** is returned if the set is empty.
E pollLast( )	Returns the last element, removing the element in the process. Because the set is sorted, this is the element with the greatest value. **null** is returned if the set is empty.
NavigableSet<E> subSet(E *lowerBound*, boolean *lowIncl*, E *upperBound*, boolean *highIncl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are greater than *lowerBound* and less than *upperBound*. If *lowIncl* is **true**, then an element equal to *lowerBound* is included. If *highIncl* is **true**, then an element equal to *upperBound* is included. The resulting set is backed by the invoking set.
NavigableSet<E> tailSet(E *lowerBound*, boolean *incl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are greater than *lowerBound*. If *incl* is **true**, then an element equal to *lowerBound* is included. The resulting set is backed by the invoking set.

Table 18-4 The Methods Declared by **NavigableSet**

ClassCastException is thrown when an object is incompatible with the elements in the set. A NullPointerException is thrown if an attempt is made to use a null object and null is not allowed in the set. An IllegalArgumentException is thrown if an invalid argument is used.

The Queue Interface
The Queue interface extends Collection and declares the behavior of a queue, which is often a first-in, first-out list. However, there are types of queues in which the ordering is based upon other criteria. Queue is a generic interface that has this declaration: interface Queue<E>
Here, E specifies the type of objects that the queue will hold. The methods declared by Queue are shown in

Method	Description
E element( )	Returns the element at the head of the queue. The element is not removed. It throws **NoSuchElementException** if the queue is empty.
boolean offer(E *obj*)	Attempts to add *obj* to the queue. Returns **true** if *obj* was added and **false** otherwise.
E peek( )	Returns the element at the head of the queue. It returns **null** if the queue is empty. The element is not removed.
E poll( )	Returns the element at the head of the queue, removing the element in the process. It returns **null** if the queue is empty.
E remove( )	Removes the element at the head of the queue, returning the element in the process. It throws **NoSuchElementException** if the queue is empty.

Table.

Several methods throw a ClassCastException when an object is incompatible with the elements in the queue. A NullPointerException is thrown if an attempt is made to store a null object and null elements are not allowed in the queue. An IllegalArgumentException is thrown if an invalid argument is used. An IllegalStateException is thrown if an attempt is made to add an element to a fixed-length queue that is full. A NoSuchElementException is thrown if an attempt is made to remove an element from an empty queue. Despite its simplicity, Queue offers several points of interest. First, elements can only be removed from the head of the queue. Second, there are two methods that obtain and remove elements: poll( ) and remove( ). The difference between them is that poll( ) returns null if the queue is empty, but remove( ) throws an exception. Third, there are two methods, element( ) and peek( ), that obtain but don't remove the element at the head of the queue. They differ only in that element( ) throws an exception if the queue is empty, but peek( ) returns null. Finally, notice that offer( ) only attempts to add an element to a queue. Because some queues have a fixed length and might be full, offer( ) can fail.

The Deque Interface

The Deque interface extends Queue and declares the behavior of a double-ended queue. Double-ended queues can function as standard, first-in, first-out queues or as last-in, firstout stacks. Deque is a generic interface that has this declaration: interface Deque<E> Here, E specifies the type of objects that the deque will hold. In addition to the methods that it inherits from Queue, Deque adds those methods.

Method	Description
void addFirst(E *obj*)	Adds *obj* to the head of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
void addLast(E *obj*)	Adds *obj* to the tail of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
Iterator<E> descendingIterator( )	Returns an iterator that moves from the tail to the head of the deque. In other words, it returns a reverse iterator.
E getFirst( )	Returns the first element in the deque. The object is not removed from the deque. It throws **NoSuchElementException** if the deque is empty.
E getLast( )	Returns the last element in the deque. The object is not removed from the deque. It throws **NoSuchElementException** if the deque is empty.
boolean offerFirst(E *obj*)	Attempts to add *obj* to the head of the deque. Returns **true** if *obj* was added and **false** otherwise. Therefore, this method returns **false** when an attempt is made to add *obj* to a full, capacity-restricted deque.
boolean offerLast(E *obj*)	Attempts to add *obj* to the tail of the deque. Returns **true** if *obj* was added and **false** otherwise.
E peekFirst( )	Returns the element at the head of the deque. It returns **null** if the deque is empty. The object is not removed.
E peekLast( )	Returns the element at the tail of the deque. It returns **null** if the deque is empty. The object is not removed.
E pollFirst( )	Returns the element at the head of the deque, removing the element in the process. It returns **null** if the deque is empty.
E pollLast( )	Returns the element at the tail of the deque, removing the element in the process. It returns **null** if the deque is empty.
E pop( )	Returns the element at the head of the deque, removing it in the process. It throws **NoSuchElementException** if the deque is empty.
void push(E *obj*)	Adds *obj* to the head of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
E removeFirst( )	Returns the element at the head of the deque, removing the element in the process. It throws **NoSuchElementException** if the deque is empty.
boolean removeFirstOccurrence(Object *obj*)	Removes the first occurrence of *obj* from the deque. Returns **true** if successful and **false** if the deque did not contain *obj*.
E removeLast( )	Returns the element at the tail of the deque, removing the element in the process. It throws **NoSuchElementException** if the deque is empty.
boolean removeLastOccurrence(Object *obj*)	Removes the last occurrence of *obj* from the deque. Returns **true** if successful and **false** if the deque did not contain *obj*.

Several methods throw a ClassCastException when an object is incompatible with the elements in the deque. A NullPointerException is thrown if an attempt is made to store a null object and null elements are not allowed in the deque. An IllegalArgumentException is thrown if an invalid argument is used. An IllegalStateException is thrown if an attempt is made to add an element to a fixed-length deque that is full. A NoSuchElementException is thrown
if an attempt is made to remove an element from an empty deque. Notice that Deque includes the methods push( ) and pop( ). These methods enable a Deque to function as a stack. Also, notice the descendingIterator( ) method. It returns an iterator that returns elements in reverse order. In other words, it returns an iterator that moves from the end of the collection to the start. A Deque implementation can be capacityrestricted, which means that only a limited number of elements can be added to the deque.
When this is the case, an attempt to add an element to the deque can fail. Deque allows you to handle such a failure in two ways. First, methods such as addFirst( ) and addLast( ) throw an IllegalStateException if a

capacity-restricted deque is full. Second, methods such as offerFirst( ) and offerLast( ) return false if the element cannot be added.

The Collection Classes

Now that you are familiar with the collection interfaces, you are ready to examine the standard classes that implement them. Some of the classes provide full implementations that can be used as-is. Others are abstract, providing skeletal implementations that are used as starting points for creating concrete collections. As a general rule, the collection classes are not synchronized.

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.
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 essence, an ArrayList is a variable-length array of object references. That is, an ArrayList can dynamically increase or decrease in size. Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array can be shrunk

ArrayList has the constructors shown here:

```
ArrayList( )
ArrayList(Collection<? extends E> c)
ArrayList(int capacity)
```

The first constructor builds an empty array list. The second constructor builds an array list that is initialized with the elements of the collection c. The third constructor builds an array list that has the specified initial capacity. The capacity is the size of the underlying array that is used to store the elements. The capacity grows automatically as elements are added to an array list.

The following program shows a simple use of ArrayList. An array list is created for objects of type String, and then several strings are added to it. The list is then displayed. Some of the elements are removed and the list is displayed again.

```
// Demonstrate ArrayList.
import java.util.*;
class ArrayListDemo {
public static void main(String args[]) {
// Create an array list.
ArrayList<String> al = new ArrayList<String>();
System.out.println("Initial size of al: " +
al.size());
// Add elements to the array list.
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");
al.add("F");
al.add(1, "A2");
System.out.println("Size of al after additions: " +
al.size());
// Display the array list.
System.out.println("Contents of al: " + al);
// Remove elements from the array list.
al.remove("F");
al.remove(2);
System.out.println("Size of al after deletions: " +
al.size());
System.out.println("Contents of al: " + al);
}
}
```

The output from this program is shown here:

```
Initial size of al: 0
Size of al after additions: 7
Contents of al: [C, A2, A, E, B, D, F]
Size of al after deletions: 5
Contents of al: [C, A2, E, B, D]
```

Notice that a1 starts out empty and grows as elements are added to it. When elements are removed, its size is reduced. In the preceding example, the contents of a collection are displayed using the defaultconversion provided by toString( ), which was inherited from AbstractCollection. Although it is sufficient for short, sample programs, you seldom use this method to display the contents of a real-world collection. Usually, you provide your own output routines. But, for the next few examples, the default output created by toString( ) is sufficient. Although the capacity of an ArrayList object increases automatically as objects are stored in it, you can increase the capacity of an ArrayList object manually by calling ensureCapacity( ). You might want to do this if you know in advance that you will be storing many more items in the collection than it can currently hold. By increasing its capacity once, at the start, you can prevent several reallocations later. Because reallocations are costly in terms of time, preventing unnecessary ones improves performance. The signature for ensureCapacity( ) is shown here:

```
void ensureCapacity(int cap)
```

Here, cap specifies the new minimum capacity of the collection. Conversely, if you want to reduce the size of the array that underlies an ArrayList object so that it is precisely as large as the number of items that it is currently holding, call trimToSize( ), shown here:

```
void trimToSize( )
```

Obtaining an Array from an ArrayList

When working with ArrayList, you will sometimes want to obtain an actual array that contains the contents of the list. You can do this by calling toArray( ), which is defined by Collection. Several reasons exist why you might want to convert a collection into an array, such as:

• To obtain faster processing times for certain operations
• To pass an array to a method that is not overloaded to accept a collection
• To integrate collection-based code with legacy code that does not understand collections

Whatever the reason, converting an ArrayList to an array is a trivial matter. As explained earlier, there are two versions of toArray( ), which are shown again here for your convenience:

```
object[ ] toArray( )
<T> T[ ] toArray(T array[ ])
```

The first returns an array of Object. The second returns an array of elements that have the same type as T. Normally, the second form is more convenient because it returns the proper type of array. The following program demonstrates its use:

```
// Convert an ArrayList into an array.
import java.util.*;
class ArrayListToArray {
public static void main(String args[]) {
// Create an array list.
ArrayList<Integer> al = new ArrayList<Integer>();
// Add elements to the array list.
al.add(1);
al.add(2);
al.add(3);
al.add(4);
System.out.println("Contents of al: " + al);
// Get the array.
Integer ia[] = new Integer[al.size()];
ia = al.toArray(ia);
int sum = 0;
// Sum the array.
for(int i : ia) sum += i;
System.out.println("Sum is: " + sum);
}
}
```

The output from the program is shown here:

```
Contents of al: [1, 2, 3, 4]
Sum is: 10
```

The program begins by creating a collection of integers. Next, toArray( ) is called and it obtains an array of Integers. Then, the contents of that array are summed by use of a for-each style for loop.

There is something else of interest in this program. As you know, collections can store only references, not values of primitive types. However, autoboxing makes it possible to pass values of type int to add( ) without having to manually wrap them within an Integer, as the program shows. Autoboxing causes them to be automatically wrapped. In this way, autoboxing significantly improves the ease with which collections can be used to store primitive values.

The LinkedList Class

The LinkedList 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)
```

The first constructor builds an empty linked list. The second constructor builds a linked list that is initialized with the elements of the collection c. Because LinkedList implements the Deque interface, you have access to the methods defined by Deque. For example, to add elements to the start of a list, you can use addFirst( ) or offerFirst( ). To add elements to the end of the list, use addLast( ) or offerLast( ). To obtain the first element, you can use getFirst( ) or peekFirst( ). To obtain the last element, use getLast( ) or peekLast( ). To remove the first element, use removeFirst( ) or pollFirst( ). To remove the last element, use removeLast( ) or pollLast( ). The following program illustrates LinkedList:

```
// Demonstrate LinkedList.
import java.util.*;
class LinkedListDemo {
public static void main(String args[]) {
// Create a linked list.
LinkedList<String> ll = new LinkedList<String>();
// Add elements to the linked list.
ll.add("F");
ll.add("B");
ll.add("D");
ll.add("E");
ll.add("C");
ll.addLast("Z");
ll.addFirst("A");
ll.add(1, "A2");
System.out.println("Original contents of ll: " + ll);
// Remove elements from the linked list.
ll.remove("F");
ll.remove(2);
System.out.println("Contents of ll after deletion: "
+ ll);
// Remove first and last elements.
ll.removeFirst();
ll.removeLast();
System.out.println("ll after deleting first and last: "
+ ll);
// Get and set a value.
String val = 11.get(2);
ll.set(2, val + " Changed");
System.out.println("ll after change: " + ll);
}
}
```

The output from this program is shown here:

```
Original contents of ll: [A, A2, F, B, D, E, C, Z]
Contents of ll after deletion: [A, A2, D, E, C, Z]
ll after deleting first and last: [A2, D, E, C]
ll after change: [A2, D, E Changed, C]
```

Because LinkedList implements the List interface, calls to add(E) append items to the end of the list, as do calls to addLast( ). To insert items at a specific location, use the add(int, E) form of add( ), as illustrated by the call to add(1, "A2") in the example.

Notice how the third element in ll is changed by employing calls to get( ) and set( ). To obtain the current value of an element, pass get( ) the index at which the element is stored. To assign a new value to that index, pass set( ) the index and its new value.

The HashSet Class

HashSet extends AbstractSet and implements the Set interface. It creates a collection that uses a hash table for storage. HashSet is a generic class that has this declaration:

```
class HashSet<E>
```

Here, E specifies the type of objects that the set will hold. As most readers likely know, a hash table stores information by using a mechanism called hashing. In hashing, the informational content of a key is used to determine a unique value, called its hash code. The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically—you never see the hash code itself. Also, your code can't directly index the hash table. The advantage of hashing is that it allows the execution time of add( ), contains( ), remove( ), and size( ) to remain constant even for large sets.

The following constructors are defined:

- HashSet( )
- HashSet(Collection<? extends E> c)
- HashSet(int capacity)
- HashSet(int capacity, float fillRatio)

The first form constructs a default hash set. The second form initializes the hash set by using the elements of c. The third form initializes the capacity of the hash set to capacity. (The default capacity is 16.) The fourth form initializes both the capacity and the fill ratio (also called load capacity) of the hash set from its arguments. The fill ratio must be between 0.0 and 1.0, and it determines how full the hash set can be before it is resized upward . Specifically, when the number of elements is greater than the capacity of the hash set multiplied by its fill ratio, the hash set is expanded. For constructors that do not take a fill ratio, 0.75 is used. HashSet does not define any additional methods beyond those provided by its superclasses and interfaces.

It is important to note that HashSet does not guarantee the order of its elements, because the process of hashing doesn't usually lend itself to the creation of sorted sets. If you need sorted storage, then another collection, such as TreeSet, is a better choice.
Here is an example that demonstrates HashSet:

```
// Demonstrate HashSet.
import java.util.*;
class HashSetDemo {
public static void main(String args[]) {
// Create a hash set.
HashSet<String> hs = new HashSet<String>();
// Add elements to the hash set.
hs.add("Beta");
hs.add("Alpha");
hs.add("Eta");
hs.add("Gamma");
hs.add("Epsilon");
hs.add("Omega");
System.out.println(hs);
}
}
```

The following is the output from this program:

```
[Gamma, Eta, Alpha, Epsilon, Omega, Beta]
```

As explained, the elements are not stored in sorted order, and the precise output may vary.
The LinkedHashSet Class
The LinkedHashSet class extends HashSet and adds no members of its own. It is a generic class that has this declaration:

```
class LinkedHashSet<E>
```

Here, E specifies the type of objects that the set will hold. Its constructors parallel those in HashSet.

LinkedHashSet maintains a linked list of the entries in the set, in the order in which they were inserted. This allows insertion-order iteration over the set. That is, when cycling through a LinkedHashSet using an iterator, the elements will be returned in the order in which they were inserted. This is also the order in which they are contained in the string returned by toString( ) when called on a LinkedHashSet object. To see the effect of LinkedHashSet, try substituting LinkedHashSet for HashSet in the preceding program.
The output will be [Beta, Alpha, Eta, Gamma, Epsilon, Omega]
which is the order in which the elements were inserted.

The TreeSet Class

TreeSet extends AbstractSet and implements the NavigableSet interface. It creates a collection that uses a tree for storage. Objects are stored in sorted, ascending order. Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.
TreeSet is a generic class that has this declaration:

```
class TreeSet<E>
```

Here, E specifies the type of objects that the set will hold.
TreeSet has the following constructors:

```
TreeSet( )
TreeSet(Collection<? extends E> c)
TreeSet(Comparator<? super E> comp)
TreeSet(SortedSet<E> ss)
```

The first form constructs an empty tree set that will be sorted in ascending order according to the natural order of its elements. The second form builds a tree set that contains the elements of c. The third form constructs an empty tree set that will be sorted according to the comparator specified by comp. The fourth form builds a tree set that contains the elements of ss.
Here is an example that demonstrates a TreeSet:

```
// Demonstrate TreeSet.
import java.util.*;
class TreeSetDemo {
```

```
public static void main(String args[]) {
// Create a tree set.
TreeSet<String> ts = new TreeSet<String>();
// Add elements to the tree set.
ts.add("C");
ts.add("A");
ts.add("B");
ts.add("E");
ts.add("F");
ts.add("D");
System.out.println(ts);
}
}
```

The output from this program is shown here:

```
[A, B, C, D, E, F]
```

As explained, because TreeSet stores its elements in a tree, they are automatically arranged in sorted order, as the output confirms. Because TreeSet implements the NavigableSet interface, you can use the methods defined
by NavigableSet to retrieve elements of a TreeSet. For example, assuming the preceding program, the following statement uses subSet( ) to obtain a subset of ts that contains the elements between C (inclusive) and F (exclusive). It then displays the resulting set.

```
System.out.println(ts.subSet("C", "F"));
```

The output from this statement is shown here:

```
[C, D, E]
```

You might want to experiment with the other methods defined by NavigableSet.

The PriorityQueue Class

PriorityQueue extends AbstractQueue and implements the Queue interface. It creates a queue that is prioritized based on the queue's comparator. PriorityQueue is a generic class that has this declaration:

```
class PriorityQueue<E>
```

Here, E specifies the type of objects stored in the queue. PriorityQueues are dynamic, growing as necessary. PriorityQueue defines the six constructors shown here:

- PriorityQueue( )
- PriorityQueue(int capacity)
- PriorityQueue(Comparator<? super E> comp) (Added by JDK 8.)
- PriorityQueue(int capacity, Comparator<? super E> comp)
- PriorityQueue(Collection<? extends E> c)
- PriorityQueue(PriorityQueue<? extends E> c)
- PriorityQueue(SortedSet<? extends E> c)

The first constructor builds an empty queue. Its starting capacity is 11. The second constructor builds a queue that has the specified initial capacity. The third constructor specifies a comparator, and the fourth builds a queue with the specified capacity and comparator. The last three constructors create queues that are initialized with the elements of the collection passed in c. In all cases, the capacity grows automatically as elements are added.

If no comparator is specified when a PriorityQueue is constructed, then the default comparator for the type of data stored in the queue is used. The default comparator will order the queue in ascending order. Thus, the head of the queue will be the smallest value.

However, by providing a custom comparator, you can specify a different ordering scheme. For example, when storing items that include a time stamp, you could prioritize the queue such that the oldest items are first in the queue. You can obtain a reference to the comparator used by a PriorityQueue by calling its comparator( ) method, shown here:

```
Comparator<? super E> comparator( )
```

It returns the comparator. If natural ordering is used for the invoking queue, null is returned. One word of caution: Although you can iterate through a PriorityQueue using an iterator, the order of that iteration is undefined. To properly use a PriorityQueue, you must call methods such as offer( ) and poll( ), which are defined by the Queue interface.

The ArrayDeque Class

The ArrayDeque class extends AbstractCollection and implements the Deque interface. It adds no methods of its own. ArrayDeque creates a dynamic array and has no capacity restrictions. (The Deque interface supports implementations that restrict capacity, but does not require such restrictions.) ArrayDeque is a generic class that has this declaration:
class ArrayDeque<E>
Here, E specifies the type of objects stored in the collection.

- ArrayDeque defines the following constructors:
- ArrayDeque( )
- ArrayDeque(int size)
- ArrayDeque(Collection<? extends E> c)

The first constructor builds an empty deque. Its starting capacity is 16. The second constructor builds a deque that has the specified initial capacity. The third constructor creates a deque that is initialized with the elements of the collection passed in c. In all cases, the capacity grows as needed to handle the elements added to the deque.
The following program demonstrates ArrayDeque by using it to create a stack:

```
// Demonstrate ArrayDeque.
import java.util.*;
class ArrayDequeDemo {
public static void main(String args[]) {
// Create an array deque.
ArrayDeque<String> adq = new ArrayDeque<String>();
// Use an ArrayDeque like a stack.
adq.push("A");
adq.push("B");
adq.push("D");
adq.push("E");
adq.push("F");
System.out.print("Popping the stack: ");
while(adq.peek() != null)
System.out.print(adq.pop() + " ");
System.out.println();
}
}
```

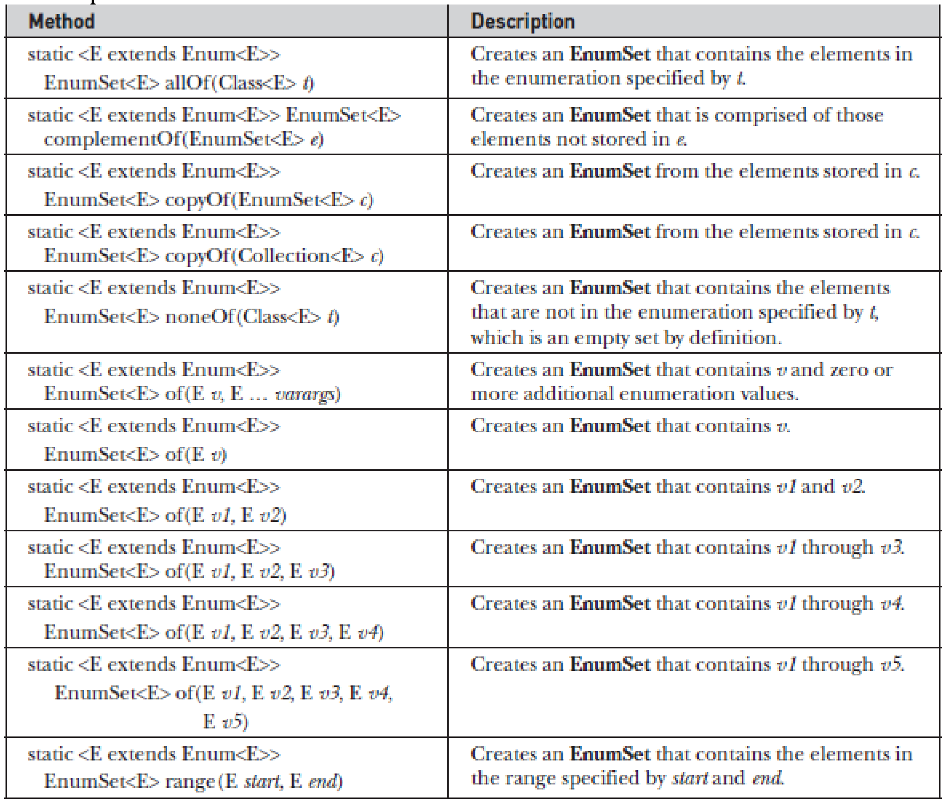
The output is shown here:

```
Popping the stack: F E D B A
```

The EnumSet Class
EnumSet extends AbstractSet and implements Set. It is specifically for use with elements of an enum type. It is a generic class that has this declaration:
class EnumSet<E extends Enum<E>>
Here, E specifies the elements. Notice that E must extend Enum<E>, which enforces the requirement that the elements must be of the specified enum type.
EnumSet defines no constructors. Instead, it uses the factory methods shown in Table to create objects. All methods can throw NullPointerException. The copyOf( ) and range( ) methods can also throw IllegalArgumentException. Notice that the of( ) method is overloaded a number of times. This is in the interest of efficiency. Passing a known number of arguments can be faster than using a vararg parameter when the number of arguments is small.
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 bidirectional traversal of a list, and the modification of elements. Iterator and ListIterator are generic interfaces which are declared as shown here:

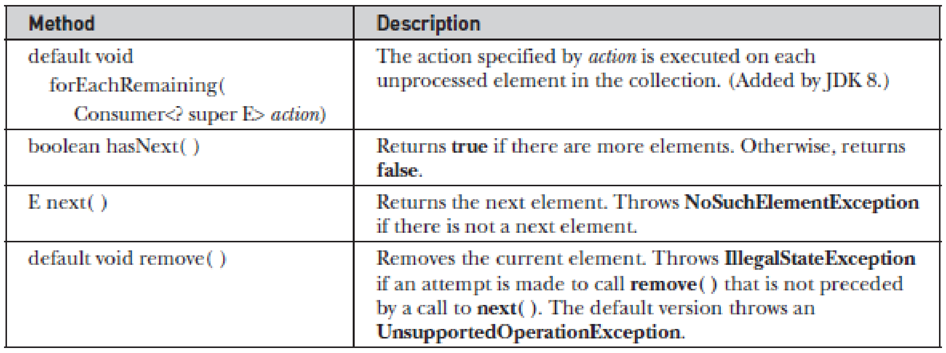
- interface Iterator<E>
- interface ListIterator<E>

Here, E specifies the type of objects being iterated. The Iterator interface declares the methods shown in Table. The methods declared by ListIterator. In both cases, operations that modify the underlying collection are optional. For example, remove( ) will throw UnsupportedOperationException when used with a read-only collection. Various other exceptions are possible.

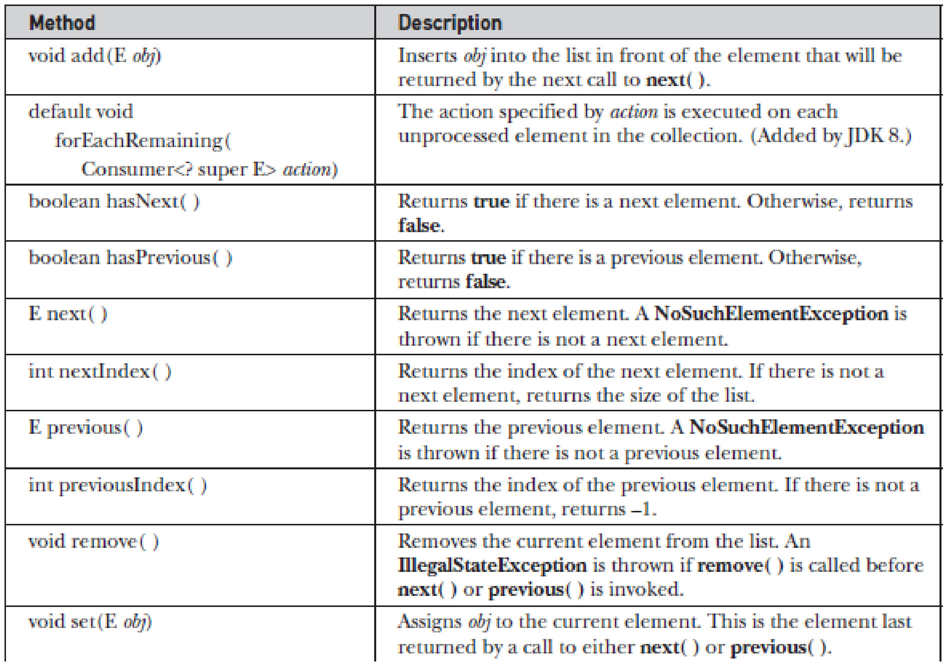


Method	Description
static <E extends Enum<E>> EnumSet<E> allOf(Class<E> *t*)	Creates an **EnumSet** that contains the elements in the enumeration specified by *t*.
static <E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> *e*)	Creates an **EnumSet** that is comprised of those elements not stored in *e*.
static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> *c*)	Creates an **EnumSet** from the elements stored in *c*.
static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> *c*)	Creates an **EnumSet** from the elements stored in *c*.
static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> *t*)	Creates an **EnumSet** that contains the elements that are not in the enumeration specified by *t*, which is an empty set by definition.
static <E extends Enum<E>> EnumSet<E> of(E *v*, E … *varargs*)	Creates an **EnumSet** that contains *v* and zero or more additional enumeration values.
static <E extends Enum<E>> EnumSet<E> of(E *v*)	Creates an **EnumSet** that contains *v*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*)	Creates an **EnumSet** that contains *v1* and *v2*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*, E *v3*)	Creates an **EnumSet** that contains *v1* through *v3*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*, E *v3*, E *v4*)	Creates an **EnumSet** that contains *v1* through *v4*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*, E *v3*, E *v4*, E *v5*)	Creates an **EnumSet** that contains *v1* through *v5*.
static <E extends Enum<E>> EnumSet<E> range(E *start*, E *end*)	Creates an **EnumSet** that contains the elements in the range specified by *start* and *end*.

Table 18-7 The Methods Declared by **EnumSet**



Method	Description
default void forEachRemaining(Consumer<? super E> *action*)	The action specified by *action* is executed on each unprocessed element in the collection. (Added by JDK 8.)
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.
default 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 default version throws an **UnsupportedOperationException**.



Method	Description
void add(E *obj*)	Inserts *obj* into the list in front of the element that will be returned by the next call to **next( )**.
default void forEachRemaining(Consumer<? super E> *action*)	The action specified by *action* is executed on each unprocessed element in the collection. (Added by JDK 8.)
boolean hasNext( )	Returns **true** if there is a next element. Otherwise, returns **false**.
boolean hasPrevious( )	Returns **true** if there is a previous element. Otherwise, returns **false**.
E next( )	Returns the next element. A **NoSuchElementException** is thrown if there is not a next element.
int nextIndex( )	Returns the index of the next element. If there is not a next element, returns the size of the list.
E previous( )	Returns the previous element. A **NoSuchElementException** is thrown if there is not a previous element.
int previousIndex( )	Returns the index of the previous element. If there is not a previous element, returns –1.
void remove( )	Removes the current element from the list. An **IllegalStateException** is thrown if **remove( )** is called before **next( )** or **previous( )** is invoked.
void set(E *obj*)	Assigns *obj* to the current element. This is the element last returned by a call to either **next( )** or **previous( )**.

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:
1. Obtain an iterator to the start of the collection by calling the collection's iterator( ) method.
2. Set up a loop that makes a call to hasNext( ). Have the loop iterate as long as hasNext( ) returns true.
3. Within the loop, obtain each element by calling next( ).

For collections that implement List, you can also obtain an iterator by calling listIterator( ). As explained, a list iterator gives you the ability to access the collection in either the forward or backward direction and lets you modify an element. Otherwise, ListIterator is used just like Iterator.
The following example implements these steps, demonstrating both the Iterator and ListIterator interfaces. It uses an ArrayList object, but the general principles apply to any type of collection. Of course, ListIterator is available only to those collections that implement the List interface.

```
// Demonstrate iterators.
import java.util.*;
class IteratorDemo {
public static void main(String args[]) {
// Create an array list.
ArrayList<String> al = new ArrayList<String>();
// Add elements to the array list.
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");
al.add("F");
// Use iterator to display contents of al.
System.out.print("Original contents of al: ");
Iterator<String> itr = al.iterator();
while(itr.hasNext()) {
String element = itr.next();
System.out.print(element + " ");
}
System.out.println();
// Modify objects being iterated.
ListIterator<String> litr = al.listIterator();
while(litr.hasNext()) {
String element = litr.next();
litr.set(element + "+");
}
System.out.print("Modified contents of al: ");
itr = al.iterator();
while(itr.hasNext()) {
String element = itr.next();
System.out.print(element + " ");
}
System.out.println();
// Now, display the list backwards.
System.out.print("Modified list backwards: ");
while(litr.hasPrevious()) {

String element = litr.previous();
System.out.print(element + " ");
}
System.out.println();
}
}
```

The output is shown here:

```
Original contents of al: C A E B D F
Modified contents of al: C+ A+ E+ B+ D+ F+
Modified list backwards: F+ D+ B+ E+ A+ C+
```

Pay special attention to how the list is displayed in reverse. After the list is modified, litr points to the end of the list. (Remember, litr.hasNext( ) returns false when the end of the list has been reached.) To traverse the list in reverse, the program continues to use litr, but this time it checks to see whether it has a previous element. As long as it does, that element is obtained and displayed.

The For-Each Alternative to Iterators

If you won't be modifying the contents of a collection or obtaining elements in reverse order, then the for-each version of the for loop is often a more convenient alternative to cycling through a collection than is using an iterator. Recall that the for can cycle through any collection of objects that implement the Iterable interface. Because all of the collection classes implement this interface, they can all be operated upon by the for.

The following example uses a for loop to sum the contents of a collection:

```
// Use the for-each for loop to cycle through a collection.
import java.util.*;
class ForEachDemo {
public static void main(String args[]) {
// Create an array list for integers.
ArrayList<Integer> vals = new ArrayList<Integer>();
// Add values to the array list.
vals.add(1);
vals.add(2);
vals.add(3);
vals.add(4);
vals.add(5);
// Use for loop to display the values.
System.out.print("Contents of vals: ");
for(int v : vals)
System.out.print(v + " ");
System.out.println();
// Now, sum the values by using a for loop.

int sum = 0;
for(int v : vals)
sum += v;
System.out.println("Sum of values: " + sum);
}
}
```

The output from the program is shown here:

```
Contents of vals: 1 2 3 4 5
Sum of values: 15
```

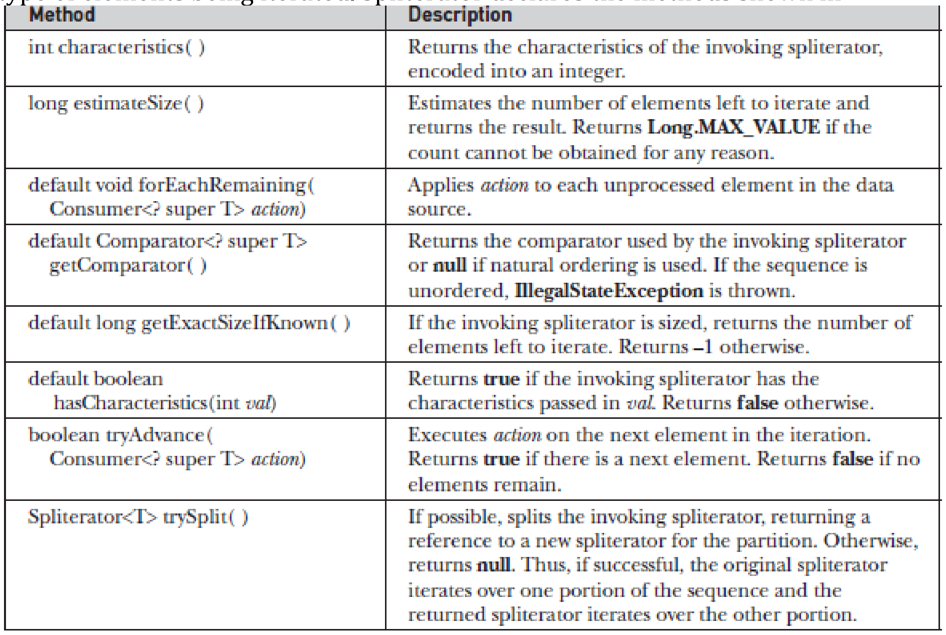
As you can see, the for loop is substantially shorter and simpler to use than the iteratorbased approach. However, it can only be used to cycle through a collection in the forward direction, and you can't modify the contents of the collection.

Spliterators

JDK 8 adds a new type of iterator called a spliterator that is defined by the Spliterator interface. A spliterator cycles through a sequence of elements, and in this regard, it is similar to the iterators just described. However, the techniques required to use it differ. Furthermore, it offers substantially more functionality than does either Iterator or ListIterator. Perhaps the most important aspect of Spliterator is its ability to provide support for parallel iteration of
portions of the sequence. Thus, Spliterator supports parallel programming. However, you can use Spliterator even if you won't be using parallel execution. One reason you might want to do so is because it offers a streamlined approach that combines the hasNext and next operations into one method. Spliterator is a generic interface that is declared like this:

```
interface Spliterator<T>
```

Here, T is the type of elements being iterated. Spliterator declares the methods shown in



Method	Description
int characteristics( )	Returns the characteristics of the invoking spliterator, encoded into an integer.
long estimateSize( )	Estimates the number of elements left to iterate and returns the result. Returns **Long.MAX_VALUE** if the count cannot be obtained for any reason.
default void forEachRemaining(Consumer<? super T> *action*)	Applies *action* to each unprocessed element in the data source.
default Comparator<? super T> getComparator( )	Returns the comparator used by the invoking spliterator or **null** if natural ordering is used. If the sequence is unordered, **IllegalStateException** is thrown.
default long getExactSizeIfKnown( )	If the invoking spliterator is sized, returns the number of elements left to iterate. Returns –1 otherwise.
default boolean hasCharacteristics(int *val*)	Returns **true** if the invoking spliterator has the characteristics passed in *val*. Returns **false** otherwise.
boolean tryAdvance(Consumer<? super T> *action*)	Executes *action* on the next element in the iteration. Returns **true** if there is a next element. Returns **false** if no elements remain.
Spliterator<T> trySplit( )	If possible, splits the invoking spliterator, returning a reference to a new spliterator for the partition. Otherwise, returns **null**. Thus, if successful, the original spliterator iterates over one portion of the sequence and the returned spliterator iterates over the other portion.

Using Spliterator for basic iteration tasks is quite easy: simply call tryAdvance( ) until it returns false. If you will be applying the same action to each element in the sequence, forEachRemaining( ) offers a streamlined alternative. In both cases, the action that will occur with each iteration is defined by what the Consumer object does with each element. Consumer is a functional interface that applies an action to an object. It is a generic functional interface declared in java.util.function. Consumer specifies only one abstract method, accept( ), which is shown here:

```
void accept(T objRef)
```

In the case of tryAdvance( ), each iteration passes the next element in the sequence to objRef. Often, the easiest way to implement Consumer is by use of a lambda expression.

The following program provides a simple example of Spliterator. Notice that the program demonstrates both tryAdvance( ) and forEachRemaining( ). Also notice how these methods combine the actions of Iterator's next( ) and hasNext( ) methods into a single call.

```
// A simple Spliterator demonstration.
import java.util.*;
class SpliteratorDemo {
public static void main(String args[]) {
// Create an array list for doubles.
ArrayList<Double> vals = new ArrayList<>();
// Add values to the array list.
vals.add(1.0);
vals.add(2.0);
vals.add(3.0);
vals.add(4.0);
vals.add(5.0);

// Use tryAdvance() to display contents of vals.
System.out.print("Contents of vals:\n");
Spliterator<Double> spltitr = vals.spliterator();
while(spltitr.tryAdvance((n) -> System.out.println(n)));
System.out.println();
// Create new list that contains square roots.
spltitr = vals.spliterator();
ArrayList<Double> sqrs = new ArrayList<>();
while(spltitr.tryAdvance((n) -> sqrs.add(Math.sqrt(n))));
// Use forEachRemaining() to display contents of sqrs.
System.out.print("Contents of sqrs:\n");
spltitr = sqrs.spliterator();
spltitr.forEachRemaining((n) -> System.out.println(n));
System.out.println();
}
}
```

The output is shown here:

```
Contents of vals:
1.0
2.0
3.0
4.0
5.0
Contents of sqrs:
1.0
1.4142135623730951
1.7320508075688772
2.0
2.23606797749979
```

Although this program demonstrates the mechanics of using Spliterator, it does not reveal its full power. As mentioned, Spliterator's maximum benefit is found in situations that involve parallel processing. In, notice the methods characteristics( ) and hasCharacteristics( ). Each Spliterator has a set of attributes, called characteristics, associated with it. These are defined by static int fields in Spliterator, such as SORTED, DISTINCT, SIZED, and IMMUTABLE, to name a few. You can obtain the characteristics by calling characteristics( ). You can determine if a characteristic is present by calling hasCharacteristics( ). Often, you won't need to access a Spliterator's characteristics, but in some cases, they can aid in creating efficient, resilient code.

There are several nested subinterfaces of Spliterator designed for use with the primitive types double, int, and long. These are called Spliterator.OfDouble, Spliterator.OfInt, and Spliterator.OfLong. There is also a generalized version called Spliterator.OfPrimitive( ), which offers additional flexibility and serves as a superinterface of the aforementioned ones.

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:

```
// A simple mailing list example.
import java.util.*;
class Address {
private String name;
private String street;
private String city;
private String state;
private String code;
Address(String n, String s, String c,
String st, String cd) {
name = n;
street = s;
city = c;
state = st;
code = cd;   }
public String toString() {
return name + "\n" + street + "\n" +city + " " + state + " " + code; }  }
class MailList {
public static void main(String args[]) {
LinkedList<Address> ml = new LinkedList<Address>();
// Add elements to the linked list.
ml.add(new Address("J.W. West", "11 Oak Ave",
"Urbana", "IL", "61801"));
ml.add(new Address("Ralph Baker", "1142 Maple Lane",
"Mahomet", "IL", "61853"));
ml.add(new Address("Tom Carlton", "867 Elm St",
"Champaign", "IL", "61820"));
// Display the mailing list.
for(Address element : ml)
System.out.println(element + "\n");
System.out.println();  }  }
```

The output from the program is shown here:

```
J.W. West
11 Oak Ave
Urbana IL 61801
Ralph Baker
1142 Maple Lane
Mahomet IL 61853
Tom Carlton
867 Elm St
Champaign IL 61820
```

Aside from storing a user-defined class in a collection, another important thing to notice about the preceding program is that it is quite short. When you consider that it sets up a linked list that can store, retrieve, and process mailing addresses in about 50 lines of code, the power of the Collections Framework begins to become apparent. As most readers know, if all of this functionality had to be coded manually, the program would be several times longer. Collections offer off-the-shelf solutions to a wide variety of programming problems. You should use them whenever the situation presents itself.

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.) RandomAccess 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.

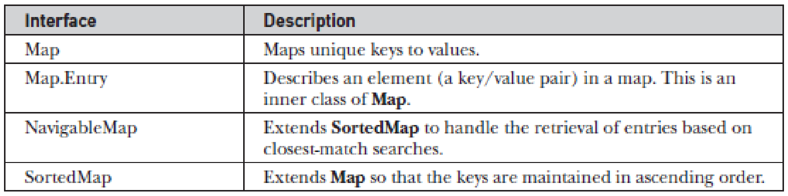
The Map Interfaces

Because the map interfaces define the character and nature of maps, this discussion of maps begins with them. The following interfaces support maps: Each interface is examined next, in turn.

The Map Interface

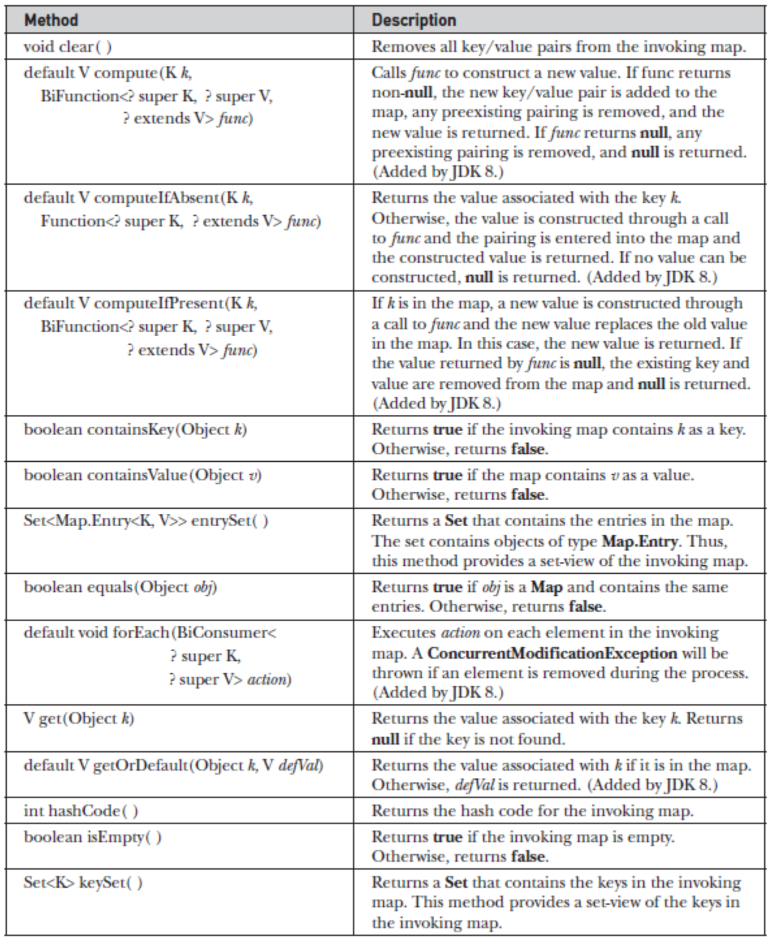
The Map interface maps unique keys to values. A key is an object that you use to retrieve a value at a later date. Given a key and a value, you can store the value in a Map object. After the value is stored, you can retrieve it by using its key. Map is generic and is declared as shown here:

interface Map<K, V>

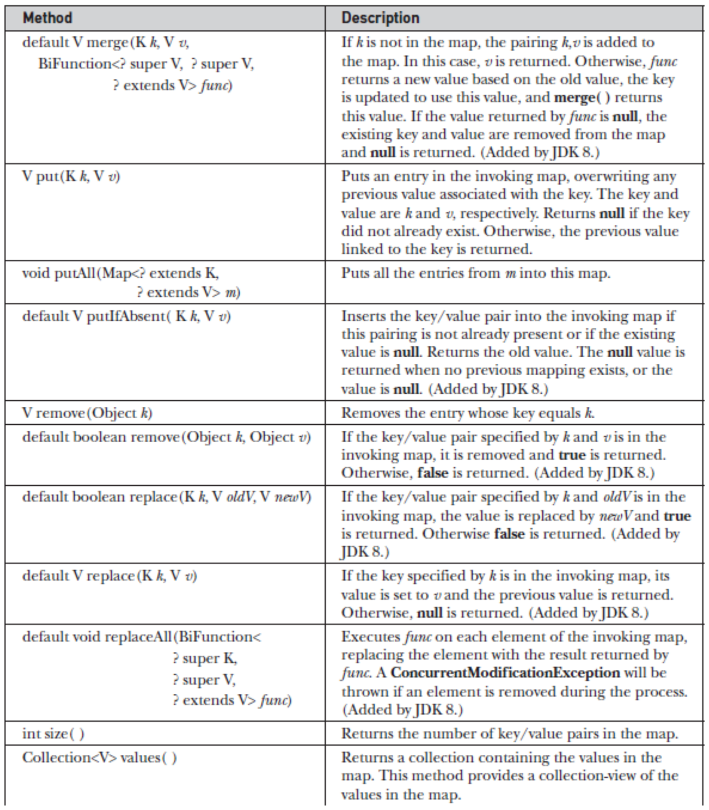


Interface	Description
Map	Maps unique keys to values.
Map.Entry	Describes an element (a key/value pair) in a map. This is an inner class of **Map**.
NavigableMap	Extends **SortedMap** to handle the retrieval of entries based on closest-match searches.
SortedMap	Extends **Map** so that the keys are maintained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

Here, K specifies the type of keys, and V specifies the type of values. Several methods throw a ClassCastException 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 and null is not allowed in the map. An UnsupportedOperationException is thrown when an attempt is made to change an unmodifiable map. An IllegalArgumentException is thrown if an invalid argument is used. Maps revolve around two basic operations: get( ) and put( ). To put a value into a map, use put( ), specifying the key and the value. To obtain a value, call get( ), passing the key as an argument. The value is returned.



Method	Description
void clear( )	Removes all key/value pairs from the invoking map.
default V compute(K *k*, BiFunction<? super K, ? super V, ? extends V> *func*)	Calls *func* to construct a new value. If func returns non-**null**, the new key/value pair is added to the map, any preexisting pairing is removed, and the new value is returned. If *func* returns **null**, any preexisting pairing is removed, and **null** is returned. (Added by JDK 8.)
default V computeIfAbsent(K *k*, Function<? super K, ? extends V> *func*)	Returns the value associated with the key *k*. Otherwise, the value is constructed through a call to *func* and the pairing is entered into the map and the constructed value is returned. If no value can be constructed, **null** is returned. (Added by JDK 8.)
default V computeIfPresent(K *k*, BiFunction<? super K, ? super V, ? extends V> *func*)	If *k* is in the map, a new value is constructed through a call to *func* and the new value replaces the old value in the map. In this case, the new value is returned. If the value returned by *func* is **null**, the existing key and value are removed from the map and **null** is returned. (Added by JDK 8.)
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**.
default void forEach(BiConsumer< ? super K, ? super V> *action*)	Executes *action* on each element in the invoking map. A **ConcurrentModificationException** will be thrown if an element is removed during the process. (Added by JDK 8.)
V get(Object *k*)	Returns the value associated with the key *k*. Returns **null** if the key is not found.
default V getOrDefault(Object *k*, V *defVal*)	Returns the value associated with *k* if it is in the map. Otherwise, *defVal* is returned. (Added by JDK 8.)
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.



Method	Description
default V merge(K *k*, V *v*, BiFunction<? super V, ? super V, ? extends V> *func*)	If *k* is not in the map, the pairing *k*,*v* is added to the map. In this case, *v* is returned. Otherwise, *func* returns a new value based on the old value, the key is updated to use this value, and **merge( )** returns this value. If the value returned by *func* is **null**, the existing key and value are removed from the map and **null** is returned. (Added by JDK 8.)
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.
default V putIfAbsent(K *k*, V *v*)	Inserts the key/value pair into the invoking map if this pairing is not already present or if the existing value is **null**. Returns the old value. The **null** value is returned when no previous mapping exists, or the value is **null**. (Added by JDK 8.)
V remove(Object *k*)	Removes the entry whose key equals *k*.
default boolean remove(Object *k*, Object *v*)	If the key/value pair specified by *k* and *v* is in the invoking map, it is removed and **true** is returned. Otherwise, **false** is returned. (Added by JDK 8.)
default boolean replace(K *k*, V *oldV*, V *newV*)	If the key/value pair specified by *k* and *oldV* is in the invoking map, the value is replaced by *newV* and **true** is returned. Otherwise **false** is returned. (Added by JDK 8.)
default V replace(K *k*, V *v*)	If the key specified by *k* is in the invoking map, its value is set to *v* and the previous value is returned. Otherwise, **null** is returned. (Added by JDK 8.)
default void replaceAll(BiFunction< ? super K, ? super V, ? extends V> *func*)	Executes *func* on each element of the invoking map, replacing the element with the result returned by *func*. A **ConcurrentModificationException** will be thrown if an element is removed during the process. (Added by JDK 8.)
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 Map Classes

Several classes provide implementations of the map interfaces. The classes that can be used for maps are summarized here:
Notice that AbstractMap is a superclass for all concrete map implementations. WeakHashMap implements a map that uses “weak keys,” which allows an element in a map to be garbage-collected when its key is otherwise unused. This class is not discussed further here. The other map classes are described next.

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 a default hash map. The second form initializes the hash map by using the elements of m. The third form initializes the capacity of the hash map to capacity. The fourth form initializes both the capacity and fill ratio of the hash map by using its arguments. The meaning of capacity and fill ratio is the same as for HashSet, described earlier. The default capacity is 16. The default fill ratio is 0.75. HashMap implements Map and extends AbstractMap. It does not add any methods of its own.
You should note that a hash map does not guarantee the order of its elements. Therefore, the order in which elements are added to a hash map is not necessarily the order in which they are read by an iterator. The following program illustrates HashMap. It maps names to account balances. Noticebhow a set-view is obtained and used.

```
import java.util.*;
class HashMapDemo {
public static void main(String args[]) {
// Create a hash map.
HashMap<String, Double> hm = new HashMap<String, Double>();
// Put elements to the map
hm.put("John Doe", new Double(3434.34));
hm.put("Tom Smith", new Double(123.22));
hm.put("Jane Baker", new Double(1378.00));
hm.put("Tod Hall", new Double(99.22));
hm.put("Ralph Smith", new Double(-19.08));
// Get a set of the entries.
Set<Map.Entry<String, Double>> set = hm.entrySet();
// Display the set.
for(Map.Entry<String, Double> me : set) {
System.out.print(me.getKey() + ": ");
System.out.println(me.getValue());
}
System.out.println();
// Deposit 1000 into John Doe's account.
double balance = hm.get("John Doe");
hm.put("John Doe", balance + 1000);
System.out.println("John Doe's new balance: " +
hm.get("John Doe"));
}
}
```

Output from this program is shown here (the precise order may vary):

```
Ralph Smith: -19.08
Tom Smith: 123.22
John Doe: 3434.34
Tod Hall: 99.22
Jane Baker: 1378.0
John Doe's new balance: 4434.34
```

The program begins by creating a hash map and then adds the mapping of names to balances. Next, the contents of the map are displayed by using a set-view, obtained by calling entrySet( ). The keys and values are displayed by calling the getKey( ) and getValue( ) methods that are defined by Map.Entry. Pay close attention to how the deposit is made into John Doe's account. The put( ) method automatically replaces any preexisting value that is associated with the specified key with the new value. Thus, after John Doe's account is updated, the hash map will still contain just one "John Doe" account.

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>

Here, T specifies the type of objects being compared. Prior to JDK 8, the Comparator interface defined only two methods: compare( ) and equals( ). The compare( ) method, shown here, compares two elements for order:

int compare(T obj1, T obj2)

obj1 and obj2 are the objects to be compared. Normally, this method returns zero if th objects are equal. It returns a positive value if obj1 is greater than obj2. Otherwise, a negative value is returned. The method can throw a ClassCastException if the types of the objects are not compatible for comparison. By implementing compare( ), you can alter the way that objects are ordered. For example, to sort in reverse order, you can create a comparator that
reverses the outcome of a comparison.

The equals( ) method, shown here, tests whether an object equals the invoking comparator:

boolean equals(object obj)

Here, obj is the object to be tested for equality. The method returns true if obj and the invoking object are both Comparator objects and use the same ordering. Otherwise, it returns false. Overriding equals( ) is not necessary, and most simple comparators will not do so.

For many years, the preceding two methods were the only methods defined by Comparator. With the release of JDK 8, the situation has dramatically changed. JDK 8 adds significant new functionality to Comparator through the use of default and static interface methods. Each is described here.

Arrays

The Arrays class provides various methods that are useful when working with arrays. These methods help bridge the gap between collections and arrays. Each method defined by Arrays.

The asList( ) method returns a List that is backed by a specified array. In other words, both the list and the array refer to the same location. It has the following signature:

static <T> List asList(T... array)

Here, array is the array that contains the data.

The binarySearch( ) method uses a binary search to find a specified value. This method must be applied to sorted arrays. Here are some of its forms.

- static int binarySearch(byte array[ ], byte value)
- static int binarySearch(char array[ ], char value)
- static int binarySearch(double array[ ], double value)
- static int binarySearch(float array[ ], float value)
- static int binarySearch(int array[ ], int value)
- static int binarySearch(long array[ ], long value)
- static int binarySearch(short array[ ], short value)
- static int binarySearch(Object array[ ], Object value)
- static <T> int binarySearch(T[ ] array, T value, Comparator<? super T> c)

Here, array is the array to be searched, and value is the value to be located. The last two forms throw a ClassCastException if array contains elements that cannot be compared (for example, Double and StringBuffer) or if value is not compatible with the types in array. In the last form, the Comparator c is used to

determine the order of the elements in array. In all cases, if value exists in array, the index of the element is returned. Otherwise, a negative
value is returned.
The copyOf( ) method returns a copy of an array and has the following forms:

- static boolean[ ] copyOf(boolean[ ] source, int len)
- static byte[ ] copyOf(byte[ ] source, int len)
- static char[ ] copyOf(char[ ] source, int len)
- static double[ ] copyOf(double[ ] source, int len)
- static float[ ] copyOf(float[ ] source, int len)
- static int[ ] copyOf(int[ ] source, int len)
- static long[ ] copyOf(long[ ] source, int len)
- static short[ ] copyOf(short[ ] source, int len)
- static <T> T[ ] copyOf(T[ ] source, int len)
- static <T,U> T[ ] copyOf(U[ ] source, int len, Class<? extends T[ ]> resultT)

The original array is specified by source, and the length of the copy is specified by len. If the copy is longer than source, then the copy is padded with zeros (for numeric arrays), nulls (for object arrays), or false (for boolean arrays). If the copy is shorter than source, then the copy is truncated. In the last form, the type of resultT becomes the type of the array returned. If len is negative, a NegativeArraySizeException is thrown. If source is null, a
NullPointerException is thrown. If resultT is incompatible with the type of source, an ArrayStoreException is thrown.
The copyOfRange( ) method returns a copy of a range within an array and has the following forms:

- static boolean[ ] copyOfRange(boolean[ ] source, int start, int end)
- static byte[ ] copyOfRange(byte[ ] source, int start, int end)
- static char[ ] copyOfRange(char[ ] source, int start, int end)
- static double[ ] copyOfRange(double[ ] source, int start, int end)
- static float[ ] copyOfRange(float[ ] source, int start, int end)
- static int[ ] copyOfRange(int[ ] source, int start, int end)
- static long[ ] copyOfRange(long[ ] source, int start, int end)
- static short[ ] copyOfRange(short[ ] source, int start, int end)
- static <T> T[ ] copyOfRange(T[ ] source, int start, int end)
- static <T,U> T[ ] copyOfRange(U[ ] source, int start, int end,

Class<? extends T[ ]> resultT)
The original array is specified by source. The range to copy is specified by the indices passed via start and end. The range runs from start to end – 1. If the range is longer than source, then the copy is padded with zeros (for numeric arrays), nulls (for object arrays), or false (for boolean arrays). In the last form, the type of resultT becomes the
type of the array returned. If start is negative or greater than the length of source, an
ArrayIndexOutOfBoundsException is thrown.

The Enumeration Interface
The Enumeration 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 superseded by Iterator. Although not deprecated, Enumeration is considered obsolete for new code. However, it is used by several methods defined by the legacy classes (such as Vector and Properties) and is used by several other API classes. Because it is still in use, it was retrofitted for generics by JDK 5. It has this declaration:
interface Enumeration<E>
where E specifies the type of element being enumerated.
Enumeration specifies the following two methods:
boolean hasMoreElements( )
nextElement( )
When implemented, hasMoreElements( ) must return true while there are still more elements to extract, and false when all the elements have been enumerated. nextElement( ) return the next object in the enumeration. That is, each call to nextElement( ) obtains the next object in the enumeration. It throws NoSuchElementException when the enumeration is complete.
Vector

Vector implements a dynamic array. It is similar to ArrayList, but with two differences: Vector is synchronized, and it contains many legacy methods that duplicate the functionality of methods defined by the Collections Framework. With the advent of collections, Vector was reengineered to extend AbstractList and to implement the List interface. With the release of JDK 5, it was retrofitted for generics and reengineered to implement Iterable. This means that Vector is fully compatible with collections, and a Vector can have its contents iterated by the enhanced for loop.
Vector is declared like this:
class Vector<E> Here, E specifies the type of element that will be stored.
Here are the Vector constructors:

- Vector( )
- Vector(int size)
- Vector(int size, int incr)
- Vector(Collection<? extends E> c)

The first form creates a default vector, which has an initial size of 10. The second form creates a vector whose initial capacity is specified by size. The third form creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward. The fourth form creates a vector that contains the elements of collection c.

Stack
Stack is a subclass of Vector that implements a standard last-in, first-out stack. Stack only defines the default constructor, which creates an empty stack. With the release of JDK 5, Stack was retrofitted for generics and is declared as shown here:
class Stack<E>
Here, E specifies the type of element stored in the stack. Stack includes all the methods defined by Vector and adds several of its own.
To put an object on the top of the stack, call push( ). To remove and return the top element, call pop( ). You can use peek( ) to return, but not remove, the top object. An EmptyStackException is thrown if you call pop( ) or peek( ) when the invoking stack is empty. The empty( ) method returns true if nothing is on the stack. The search( ) method
determines whether an object exists on the stack and returns the number of pops that are required to bring it to the top of the stack. Here is an example that creates a stack, pushes several Integer objects onto it, and then pops them off again:

```
// Demonstrate the Stack class.
import java.util.*;
class StackDemo {
static void showpush(Stack<Integer> st, int a) {
st.push(a);
System.out.println("push(" + a + ")");
System.out.println("stack: " + st);
}
static void showpop(Stack<Integer> st) {
System.out.print("pop -> ");
Integer a = st.pop();
System.out.println(a);
System.out.println("stack: " + st);
}
public static void main(String args[]) {
Stack<Integer> st = new Stack<Integer>();
System.out.println("stack: " + st);
showpush(st, 42);
showpush(st, 66);
showpush(st, 99);
showpop(st);
showpop(st);
showpop(st);

try {
```

```
showpop(st);
} catch (EmptyStackException e) {
System.out.println("empty stack");
}
}
}
```

The following is the output produced by the program; notice how the exception handler for EmptyStackException is caught so that you can gracefully handle a stack underflow:

```
stack: [ ]
push(42)
stack: [42]
push(66)
stack: [42, 66]
push(99)
stack: [42, 66, 99]
pop -> 99
stack: [42, 66]
pop -> 66
stack: [42]
pop -> 42
stack: [ ]
pop -> empty stack
```

One other point: although Stack is not deprecated, ArrayDeque is a better choice.

Dictionary

Dictionary is an abstract class that represents a key/value storage repository and operates much like Map. Given a key and value, you can store the value in a Dictionary object. Once the value is stored, you can retrieve it by using its key. Thus, like a map, a dictionary can be thought of as a list of key/value pairs. Although not currently deprecated, Dictionary is classified as obsolete, because it is fully superseded by Map. However, Dictionary is still in use and thus is discussed here.

With the advent of JDK 5, Dictionary was made generic. It is declared as shown here:

```
class Dictionary<K, V>
```

Here, K specifies the type of keys, and V specifies the type of values. To add a key and a value, use the put( ) method. Use get( ) to retrieve the value of a given key. The keys and values can each be returned as an Enumeration by the keys( ) and elements( ) methods, respectively. The size( ) method returns the number of key/value pairs stored in a dictionary, and isEmpty( ) returns true when the dictionary is empty. You can use the remove( ) method to delete a key/value pair.

Generics in Java

The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects.

Before generics, we can store any type of objects in collection i.e. non-generic. Now generics, forces the java programmer to store specific type of objects.

Advantage of Java Generics

There are mainly 3 advantages of generics. They are as follows:

1) Type-safety : We can hold only a single type of objects in generics. It doesn't allow to store other objects.

2) Type casting is not required: There is no need to typecast the object.

Before Generics, we need to type cast.

```
1. List list = new ArrayList();
2. list.add("hello");
3. String s = (String) list.get(0);//typecasting
```

After Generics, we don't need to typecast the object.

```
1. List<String> list = new ArrayList<String>();
2. list.add("hello");
3. String s = list.get(0);
```

3) Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

```
1. List<String> list = new ArrayList<String>();
```

```
2. list.add("hello");
3. list.add(32);//Compile Time Error
```

Syntax to use generic collection

```
1. ClassOrInterface<Type>
```

Example to use Generics in java

```
1. ArrayList<String>
```

Full Example of Generics in Java

Here, we are using the ArrayList class, but you can use any collection class such as ArrayList, LinkedList, HashSet, TreeSet, HashMap, Comparator etc.

```
import java.util.*;
class TestGenerics1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();
list.add("rahul");
list.add("jai");
//list.add(32);//compile time error
String s=list.get(1);//type casting is not required
System.out.println("element is: "+s);
Iterator<String> itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

```
Output:element is: jai
       rahul
       jai
```

Example of Java Generics using Map

Now we are going to use map elements using generics. Here, we need to pass key and value. Let us understand it by a simple example:

```
import java.util.*;
class TestGenerics2{
public static void main(String args[]){
Map<Integer,String> map=new HashMap<Integer,String>();
map.put(1,"vijay");
map.put(4,"umesh");
map.put(2,"ankit");
  //Now use Map.Entry for Set and Iterator
Set<Map.Entry<Integer,String>> set=map.entrySet();
 Iterator<Map.Entry<Integer,String>> itr=set.iterator();
while(itr.hasNext()){
Map.Entry e=itr.next();//no need to typecast
System.out.println(e.getKey()+" "+e.getValue());
}
}}
```

```
Output:1 vijay
     2 ankit
     4 umesh
```

Generic class

A class that can refer to any type is known as generic class. Here, we are using **T** type parameter to create the generic class of specific type.

Let's see the simple example to create and use the generic class.

Creating generic class:

```
class MyGen<T>{
T obj;
void add(T obj){this.obj=obj;}
T get(){return obj;}
}
```

The T type indicates that it can refer to any type (like String, Integer, Employee etc.). The type you specify for the class, will be used to store and retrieve the data.

Using generic class:

Let's see the code to use the generic class.

```
class TestGenerics3{
public static void main(String args[]){
MyGen<Integer> m=new MyGen<Integer>();
m.add(2);
//m.add("vivek");//Compile time error
System.out.println(m.get());
}}
```

```
Output:2
```

Type Parameters

The type parameters naming conventions are important to learn generics thoroughly. The commonly type parameters are as follows:

1. T - Type
2. E - Element
3. K - Key
4. N - Number
5. V - Value

Generic Method

Like generic class, we can create generic method that can accept any type of argument.

Let's see a simple example of java generic method to print array elements. We are using here **E** to denote the element.

```
public class TestGenerics4{

  public static < E > void printArray(E[] elements) {
      for ( E element : elements){
         System.out.println(element );
      }
```

```
        System.out.println();
    }
    public static void main( String args[] ) {
        Integer[] intArray = { 10, 20, 30, 40, 50 };
        Character[] charArray = { 'J', 'A', 'V', 'A', 'T','P','O','I','N','T' };

        System.out.println( "Printing Integer Array" );
        printArray( intArray  );

       System.out.println( "Printing Character Array" );
        printArray( charArray );
    }
}
```

```
Output:Printing Integer Array
        10
        20
        30
        40
        50
        Printing Character Array
        J
        A
        V
        A
        T
        P
        O
        I
        N
        T
```

Wildcard in Java Generics

The ? (question mark) symbol represents wildcard element. It means any type. If we write <? extends Number>, it means any child class of Number e.g. Integer, Float, double etc. Now we can call the method of Number class through any child class object.

Let's understand it by the example given below:

```
import java.util.*;
abstract class Shape{
abstract void draw();
}
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
}
class Circle extends Shape{
void draw(){System.out.println("drawing circle");}
}


class GenericTest{
```

```
//creating a method that accepts only child class of Shape
public static void drawShapes(List<? extends Shape> lists){
for(Shape s:lists){
s.draw();//calling method of Shape class by child class instance
}
}
public static void main(String args[]){
List<Rectangle> list1=new ArrayList<Rectangle>();
list1.add(new Rectangle());
  List<Circle> list2=new ArrayList<Circle>();
list2.add(new Circle());
list2.add(new Circle());
  drawShapes(list1);
drawShapes(list2);
}}
```

```
drawing rectangle
drawing circle
drawing circle
```