CHAPTER 10
Commonly Used Classes from the `java.lang` Package

As seen in Chapter 9, there are various packages available as part of the Java API. One of the basic packages is the `java.lang` package. This package contains the core classes and interfaces used in the Java Language. These include classes and interfaces like `System`, `String`, `StringBuffer`, `StringBuilder`, `CharSequence`, `Comparable`, `Math` and the `Wrapper` classes. In this chapter we discuss some of these classes and interfaces and their common usages.

10.1 `Comparable` AND `Comparator` INTERFACES

We saw in Chapter 3 that the relational operators `< > <= >=` work only for numeric data types, i.e. the comparison is not available for reference data types. For boolean data types, it is very logical that we cannot have a comparison between two boolean values for greater than or less than; similarly not everything can be compared for greater than or less than. So, we do not have the relational operators for the reference data types as well. It is now upto the programmer. Whenever he creates a new data type, by defining a class, he should be able to decide whether the instances of the class which is being defined can be compared for greater than or less than relations. This is what can be done by using the `Comparable` interface.

10.1.1 `Comparable`

The `Comparable` interface is always used to indicate that instances of a particular class have a natural ordering. The logic for making the comparison depends on the class. Each class which is being defined as `Comparable` has its own logic for comparison. This can be specified by implementing the `compareTo` method from the `Comparable` interface. The `Comparable`

interface is defined with compareTo() method as given below:

```
1   public interface Comparable {
2       public int compareTo(Object o);
3   }
```

The compareTo() method returns an int value. This value is interpreted to indicate whether the current object being compared to another instance (passed as parameter) is greater than, less than or equal to the other object, e.g. if we have two instances x and y of a given class which implements the Comparable interface, then the int value returned from the invocation of x.compareTo(y) will be interpreted as follows:

- If the returned value is negative then x is considered to precede y, i.e. x is less than y.
- If the returned value is positive then x is considered to succeed y, i.e. x is greater than y.
- If the returned value is zero then x and y are considered to be equivalent, i.e. x is equal to y.

Whenever someone implements the compareTo() method then he is also expected to ensure consistency between the implementations of compareTo() and the equals() methods, i.e. the equality defined by the equals() method should be consistent with the equality of the compareTo() method. One should try to ensure this kind of consistency.

A sample implementation of the compareTo() method for instances of the geometry.shapes.Rectangle class could be as given below:

```
1   public int compareTo(Object o) {
2       return this.area() - ((Rectangle)o).area();
3   }
```

The above implementation of compareTo() is not really consistent with the equals() method implementation where two rectangles were considered to be equal only if their lengths and widths match.

EXERCISE 10.1 Correct the above implementation of compareTo() method to make it consistent with the equals() method as defined in Listings 8.2 and 8.3.

The above definition of Comparable interface is from an older version of API (pre Java 5). Java 5 had introduced generic data types, and Comparable from Java 5 is a generic data type (generics are discussed in Section 13.3.4). From Java 5, the Comparable interface is as given below:

```
1   public interface Comparable<T> {
2       int compareTo(T o);
3   }
```

This definition of Comparable allows us to write classes implementing Comparable interface as given in Listing 10.1.

Listing 10.1. Implementation of Comparable for Rectangle class

```
1  package geometry.shapes;
2
3  public class Rectangle implements Comparable<Rectangle> {
4      ...  // other members of the Rectangle class
5      public int compareTo(Rectangle r) {
6          return this.area() - r.area();
7      }
8  }
```

The main difference between this implementation and the earlier implementation is that here we do not have to use the explicit cast. The casting is being managed by the compiler.

EXERCISE 10.2 Rewrite the Account class of Exercise 9.1 to make it implement the Comparable interface. The natural ordering for Account objects will be based on their account numbers.

The updated Account class is given in Listing 10.2.

Listing 10.2. Account class implementing Comparable

```
1  package bank;
2
3  public abstract class Account implements Comparable<Account> {
4
5      private int accountNumber;
6      private String name;
7      protected double balance;
8
9      public Account(int acno, String n, double openBal) {
10         this.accountNumber = acno;
11         this.name = n;
12         this.balance = openBal;
13     }
14
15     public int getAccountNumber() {
16         return this.accountNumber;
17     }
18
19     public String getName() {
20         return this.name;
21     }
22
23     public double getBalance() {
24         return this.balance;
25     }
26
```

```
27      public void deposit(double amt) {
28          this.balance += amt;
29      }
30
31      public boolean withdraw(double amt) {
32          if (this.balance < amt) {
33              return false;
34          }
35          this.balance -= amt;
36          return true;
37      }
38
39      public void display() {
40  //          System.out.println("Account:"+this.accountNumber+","+this
        .name+","+this.balance);
41          System.out.println(this);
42      }
43
44      private static int lastAccountNumber = 1000;
45
46      public Account(String n, double openBal) {
47          this(++lastAccountNumber, n, openBal);
48      }
49
50      public String toString() {
51          return this.getClass().getName()+":"+this.accountNumber+","
                +this.name+","+this.balance;
52      }
53
54      public boolean equals(Object obj) {
55          if (this.getClass() != obj.getClass()) {
56              return false;
57          }
58          return this.accountNumber == ((Account)obj).accountNumber;
59      }
60
61      public int hashCode() {
62          return this.accountNumber;
63      }
64
65      public int compareTo(Account ac) {
66          return this.accountNumber - ac.accountNumber;
67      }
68  }
```

The most common usage of the `Comparable` interface is while sorting an array of Objects. We have a class called `Arrays` in the java.util package. This class has only `static` methods. These methods are the various kinds of utilities for manipulating arrays of different types. One of the methods in this class is the `sort()` method. This method is overloaded to sort the

arrays of different numeric types. It is also overloaded to sort arrays of Object. So one of the methods is as given under:

```
1    public static void sort(Object[] a)
```

This method does the sorting of the Object array by assuming that all the elements in the array are Comparable. The ordering of the elements is decided by the implementation of the compareTo() method of the Comparable interface. In case any of the elements in the array are not Comparable, then the method would throw ClassCastException at runtime.

10.1.2 Comparator

The sort method for sorting the elements in an Object array, is overloaded in the Arrays class. This method takes two parameters, one is the Object[] and the other is of type Comparator. Comparator is an interface in the java.util package. In case of the Comparable interface, we have the compareTo() method, which takes only one parameter. The Comparator interface has only one method called compare() which takes two parameters. This method of Comparator has two parameters as given below:

```
1    public interface Comparator {
2        public int compare(Object o1, Object o2);
3    }
```

An instance of the Comparator interface is used for specifying the logic for ordering objects. We can have different implementation classes of the compare() method of the Comparator interface, each giving a different logic for comparison. We may have requirements of ordering objects in different manners, e.g. for the instances of Account class, we may like to order the instance of the Account by the name of the account holder, or sometimes we may like to order the instances of Account by the balances. This is where the Comparator interface is useful. The Comparable interface defines the natural ordering for elements of a class, which is mostly consistent with the equals method as well, whereas the Comparator interface defines ordering logic which may be different from the natural ordering. In fact, the Comparator can have the logic of comparing elements of classes which may not have a natural ordering.

The compare() method returns an int value. This value is interpreted to indicate whether the first object being compared to the second instance is greater than, less than or equal to the second object, e.g. if we have two instances x and y, then the return int value from the invocation of compare(x, y) will be interpreted as follows:

- If the returned value is positive then x is considered to precede y.
- If the returned value is positive then x is considered to succeed y.
- If the returned value is zero then x and y are considered to be equivalent.

EXERCISE 10.3 Define a class called NameComparator which implements Comparator to compare instances of Account. It would compare two account instances based on their names. Test the Comparator by creating an array of account and sorting them using the sort method of Arrays class, which takes the Comparator as the parameter.

A possible implementation of NameComparator is given in Listing 10.3.

Listing 10.3. NameComparator.java

```
1 package bank;
2
3 import java.util.Comparator;
4
5 public class NameComparator<Account> implements Comparator<Account>
      {
6
7     public int compare(Account ac1, Account ac2) {
8         return ac1.getName().compareTo(ac2.getName());
9     }
10 }
```

10.2 String CLASS

String is the most commonly used class. In fact we know that every data type can be converted to String. Let's try to look at the commonly used features from the String class.

What is a String? String is a sequence of unicode characters. String is a final class, whose instances are immutable, i.e. we cannot create sub-classes from the String class and by immutable, we mean that once any instance of the String class is created, there is no way that we could change the characters in the String instance.

10.2.1 String Constants

String constants can be used in any class by enclosing a sequence of characters within double quotes, e.g. "Hello world". Within JVM we have a constant pool and all instances of the String constants are allocated in the constant pool. Within the constant pool there is no repetition of the same constant value, i.e. if there are two different places from where the same value of the constant is being used, then there are no seperate instances for them in the constant pool (Listing 10.4).

Listing 10.4. String constants and equality

```
1  String s1 = "Hello";
2  String s2 = new String(s1);
3  String s3 = "Hello";
4  System.out.println(s1 == s2);
5  System.out.println(s1 == s3);  // s1 and s3 refer to the same
       String constant in the constant pool.
6  System.out.println(s2 == s3);
```

What is the output generated from Lines 4, 5 and 6 in the code in Listing 10.4? In Line 2, we created a new instance of `String` using a constructor. `String` class has a constructor that takes another `String` as a parameter and creates a new `String` object containing the same set of characters as the given `String`. There would not be two different instances for the `String` constant `"Hello"` which is used from Lines 1 and 3 or both or any other place where the same `String` constant may appear. In the above code `s2` will refer to a new object, and `s1` and `s3` both refer to the same object, which is from the constant pool.

```
1     String s1 = "Hello "+"world";
```

The above `String` is made up of a concatenation of two `String` constants. This would be known to the compiler and would be evaluated as a single constant, `"Hello world"`. So, there are no separate constants for `"Hello"` and `"world"`, in the constant pool, and there is no `String` concatenation carried out at the runtime.

10.2.2 Interfaces Implemented by `String`

The `String` class implements the java.lang.CharSequence, java.lang.Comparable and java.io.Serializable interfaces. java.io.Serializable is a marker interface. The `String` class implements all the methods from these interfaces. What is a `CharSequence`? `CharSequence` is implemented by various classes which maintain a sequence of `char` data type. The `CharSequence` interface declares the methods given in Listing 10.5.

Listing 10.5. Methods in `CharSequence` interface

```
1 public interface CharSequence {
2     public int length();
3     public char charAt(int index);
4     public CharSequence subSequence(int start, int end);
5     public String toString();
6 }
```

The method `length()` returns the number of characters in the character sequence. The method `charAt()` returns the `char` value at the index position specified by the value of the integer parameter. The valid index positions are from `0` to the value of `length() - 1`. In case the index value passed is not in the range from `0` to `length - 1`, then the `charAt()` method throws an `IndexOutOfBoundsException`.

The method `subSequence()` takes two integer parameters `start` and `end` and returns another `CharSequence` instance which has characters from `start` index upto `end - 1` index from the target `CharSequence`. This method would throw `IndexOutOfBoundsException` if any of the index arguments is negative or greater than `length()`.

10.2.2.1 Comparison of strings

In Listing 10.4, we saw that the == operator cannot be used for comparing two strings for equality. It only compares the references. The `equals()` is overridden in the `String` class

to compare two strings for equality (having exactly same content). We also have the method equalsIgnoreCase() to make an equality comparison, ignoring case differences.

The String class implements Comparable interface and any two strings can be compared using the compareTo() method of the Comparabe interface. The compareTo() method compares two strings lexicographically. The comparison is based on the unicode value of each character in the strings. The character sequence represented by the String is compared lexicographically to the character sequence represented by the String passed as argument, e.g. if we have two strings x and y, then the int value returned by x.compareTo(y) is interpreted as follows:

- If the returned value is negative then x lexicographically precedes y.
- If the returned value is positive then x lexicographically follows y.
- If the returned value is zero then x and y are considered to be equal. The equals() method is consistent with compareTo(), i.e. x.equals(y) would always be true, whenever x.compareTo(y) returns zero and vice versa.

Two strings can also be compared lexicographically, ignoring case differences by using the method compareToIgnoreCase(). This method returns the int value which is interpreted similar to compareTo() ignoring case differences. The String class also has a Comparator called CASE_INSENSITIVE_ORDER which orders String objects lexicographically ignoring case (Listing 10.6).

Listing 10.6. Methods for comparing strings

```
1    public boolean equals(String s)
2    public boolean equalsIgnoreCase(String s)
3    public int compare(String s)
4    public int compareToIgnoreCase(String s)
5    public boolean contentEquals(CharSequence cs)
```

There are other methods which locate characters, strings or other character sequences within the string. Some of these methods are given in Listing 10.7.

Listing 10.7. Methods of String class related to locating strings and characters

```
1    public int indexOf(char ch);
2    public int indexOf(char ch, int start);
3    public int indexOf(String str);
4    public int indexOf(String str, int start);
5    public int lastIndexOf(char ch);
6    public int lastIndexOf(char ch, int start);
7    public int lastIndexOf(String str);
8    public int lastIndexOf(String str, int start);
9    public boolean contains(CharSequence s0
10   public boolean startsWith(String s)
11   public boolean startsWith(String s, int toffset)
12   public boolean endsWith(String s)
```

Let us assume a declaration of String as follows:

```
1    String s = "Hello world";
                 0123456789 10
```

String class has a method called indexOf(), lastIndexOf(), startsWith(), endsWith(), which is overloaded as shown in Listing 10.7.

The indexOf() method returns the index position of a given character or a string within the string. It would return a -1, if the character or string being searched is not part of the string. We can pass a second parameter for the starting index postion from where the search should start. A variant of the indexOf() method is the lastIndexOf() method which also returns the index position of a character or a string within the given string, but this method searches from right to left, i.e. it starts searching from the last index position and moves towards position °0. So, the s.indexOf('l') would return a value of 2, since the first occurence of the character 'l' occurs at the index position of the string "Hello world". In String, index positions start from 0. The result of s.indexOf('l',3), would return 3, since when we start searching for the character 'l' from the index position 3, the search finds a match right at position 3 itself. And when we use s.indexOf('l', 4), it would return 9, since starting to search for 'l' from index position 4 onwards, the next occurence of 'l' occurs at index position 9 within the string. The methods startsWith() and endsWith() return a boolean value to check if the string starts with or ends with a particular string. The method contains() can be used to check if any CharSequence is contained within the string. Here the parameter could be any implementation of CharSequence.

10.2.3 Commonly Used Constructors of String Class

There are various constructors in the String class. Some of the commonly used constructors are listed below:

```
1    public String()
2    public String(String original)
3    public String(char[] value)
4    public String(char[] value, int offset, int count)
5    public String(byte[] value, String charsetName)
```

The last constructor above takes two parameters, the first parameter is a byte[] and the second is the name of a character set. This constructor decodes the byte values available from the first parameter to unicode according to the character set specified by the charsetName. While decoding the value, the bytes in the byte[] are considered as unsigned byte values in the range from 0 to 255. So, if we have a byte[] which has byte with a value of 164 decimal, and the character set is specified as "ISCII91", then, this byte will get converted to the char value of \u0905. In ISCII91 the value 164 is for the Devanagari letter A, and in unicode the Devanagari letter A has a value of 0905(Hex).

10.2.4 Common Methods from String Class

Let us try to look at some more of the commonly used methods available in the String class. The String class also has methods for deriving new instances of string from a given string. The methods for case conversion are setoUpperCase() and toLowerCase(). The trim() method returns a new instance of String after removing all the leading and trailing white space characters. The method substring() returns a new string by using characters from the start index to the end index, similar to the subSequence() method. When only one parameter is used in the substring() method it returns a sub-string which starts at the specified index position and includes characters upto the end of the string. All these methods with their signatures are shown in Listing 10.8.

Listing 10.8. Methods for conversion

```
1    public String toUpperCase()
1    public String toLowerCase()
1    public String trim()
1    public String substring(int start)
1    public String substring(int start, int end)
```

10.2.5 Methods Involving Regular Expresssions

From Java 1.4 onwards, support for regular expressions was introduced in Java API. The String class also has some methods which use regular expresssions. Listing 10.9 shows some of the common methods which make use of regular expressions.

Listing 10.9. Methods using regular expressions

```
1    public boolean matches(String regex)
2    public boolean regionMatches(boolean ignorecase, int toffset,
         String other, int ooffset, int len)
3    public boolean regionMatches(int toffset, String other, int
         ooffset, int len)
4    public String replaceAll(String regex, String replacement)
5    public String replaceFirst(String regex, String replacement)
6    public String[] split(String regex)
7    public String[] split(String regex, int limit)
```

The method matches() checks if the string is a pattern which is specified in the regex parameter. The patterns are specified using a regular expressions. There is a separate package java.util.regex which has classes to compile a pattern and to match the pattern. These are discussed in Chapter 13. The method replaceAll() returns a new string by searching for the pattern specified by the first parameter in the string and by replacing all such occurrences of the pattern with the second string. The replaceFirst() method only replaces the first occurrence of the pattern and returns a new string. The split() method splits the string whenever it finds a pattern matching the regex specified, so if a string contains a pattern specified by the regex 5 times then the string array returned by the method would have 6 elements, e.g. if we have a string with comma-separated values and we would like to get all the values between the

commas, we could use the split method as follows:

```
1    String s = "1001,Tom Valesky,Savings,10000";
2    String[] values = s.split(",");
```

In the above code the comma occurs 3 times in the string, so the `String[]` for values will contain 4 elements as `"1001","Tom Valesky","Savings","10000"`. When the limit is specified in the method invocation, then the number of times the split is done cannot be more than limit-1. Two consecutive occurences of the pattern would result in a `null` value in the array element at the corresponding index value. So if we have a string as `"1001,Tom Valesky,,Savings,,10000"`. Then splitting this string using comma would result in an array of size 5 with values as `"1001","Tom Valesky","Savings",null,"10000"`.

10.2.6 Conversion of Primitives to `String`

The `String` class has methods for converting other data types to string. These are given in Listing 10.10. The `valueOf()` methods. These are given in Listing 10.10. The `valueOf()` method is overloaded and takes different data types as parameters, converts them to string and returns a `String`.

Listing 10.10. `valueOf()` methods in `String` class

```
1    public static String valueOf(byte b)
2    public static String valueOf(short s)
3    public static String valueOf(int i)
4    public static String valueOf(long l)
5    public static String valueOf(float f)
6    public static String valueOf(double d)
7    public static String valueOf(char c)
8    public static String valueOf(boolean b)
9    public static String valueOf(char[] data)
10   public static String valueOf(char[] data, int offset, int count
         )
11   public static String valueOf(Object o)
```

10.3 `StringBuffer` AND `StringBuilder` CLASSES

The instances of `String` class are immutable. The `StringBuffer` and the `StringBuilder` are the mutable character sequences. The `StringBuilder` is introduced from Java 5 onwards. It has exactly the same set of methods as are available from the `StringBuffer` class, but its methods are faster compared to those of `StringBuffer`. The methods of `StringBuffer` are slow only because they are `synchronized`. The `synchronized` keyword is discussed in Chapter 16.

The instances of `StringBuffer` hold the sequence of characters in some internal array of characters. The `StringBuffer` has a capacity, which is the number of characters it can

hold without requiring reallocation. When the length of characters in the StringBuffer overflow and exceed the capacity, the capacity is automatically increased. The constructors for StringBuffer are given in Listing 10.11.

Listing 10.11. StringBuffer constructors

```
1    public StringBuffer()
2    public StringBuffer(int capacity)
3    public StringBuffer(CharSequence cs)
4    public StringBuffer(String s)
```

A StringBuffer could be created by specifying the initial capacity or it could be created from any existing CharSequence object. The initial default capacity of a StringBuffer is 16 characters.

In StringBuffer class, methods do not return void, instead where an operation is done and no value is to be returned, the methods have been made to return the StringBuffer instance on which the method is invoked. This is done to help in method chaining, i.e. we could carry out multiple operations in a single statement. Example, The append() and the insert() methods both return the StringBuffer, so on an instance of StringBuffer sb, we could invoke the two methods in a chain like sb.append("world").insert(0, "Hello "); The append() and insert() are the most common methods used on a StringBuffer. These methods are overloaded to take any kind of data type as a parameter. All these append() and insert() methods convert the parameter value to string and append or insert the string into the StringBuffer instance. The various forms of append() and insert() methods are given in Listing 10.12.

Listing 10.12. append() and insert() methods

```
1    public StringBuffer append(byte b)
2    public StringBuffer insert(int offset, byte b)
3    public StringBuffer append(short s)
4    public StringBuffer insert(int offset, short s)
5    public StringBuffer append(int i)
6    public StringBuffer insert(int offset, int i)
7    public StringBuffer append(long l)
8    public StringBuffer insert(int offset, long l)
9    public StringBuffer append(float f)
10   public StringBuffer insert(int offset, float f)
11   public StringBuffer append(double d)
12   public StringBuffer insert(int offset, double d)
13   public StringBuffer append(char c)
14   public StringBuffer insert(int offset, char c)
15   public StringBuffer append(boolean b)
16   public StringBuffer insert(int offset, boolean b)
17   public StringBuffer append(char[] ch)
18   public StringBuffer insert(int offset, char[] ch)
19   public StringBuffer append(char[] ch, int offset, int len)
20   public StringBuffer insert(int index, char[] ch, int offset,
         int len)
21   public StringBuffer append(CharSequence cs)
```

```
22    public StringBuffer insert(int offset, CharSequence cs)
23    public StringBuffer append(CharSequence cs, int offset, int len
          )
24    public StringBuffer insert(int index, CharSequence cs, int
          offset, int len)
25    public StringBuffer append(Object o)
26    public StringBuffer insert(int offset, Object o)
27    public StringBuffer append(String s)
28    public StringBuffer insert(int offset, String s)
29    public StringBuffer append(StringBuffer sb)
30    public StringBuffer appendCodePoint(int codepoint)
```

The StringBuffer has methods to remove and replace characters within the character sequence of the StringBuffer. These methods are given in Listing 10.13.

Listing 10.13. Methods to remove and replace characters from a StringBuffer

```
1    public StringBuffer delete(int start, int end)
2    public StringBuffer deleteCharAt(int index)
3    public StringBuffer replace(int start, int end, String str)
4    public StringBuffer reverse()
5    public StringBuffer setCharAt(int index, char ch)
```

The delete() and deleteCharAt() methods remove characters from the character sequence at the specified index range or a single character at the specified index position. The replace() method replaces the characters within the specified range with the characters from the specified string. The reverse() method reverses all the characters in the character sequence, so that the last character becomes first, second last becomes second and so on. The setCharAt() method allows us to replace the character at a specified index position with the new character value.

The StringBuffer and the StringBuilder both implement the CharSequence, Appendable and the Serializable interfaces. The Appendable interface has the three append() methods which append a single character, a character sequence or a segment of a character sequence.

The methods related to the size and capacity of a StringBuffer are as given in Listing 10.14.

Listing 10.14. Capacity and size of StringBuffer

```
1    public int capacity()
2    public StringBuffer ensureCapacity(int minimumCapacity)
3    public StringBuffer trimToSize()
4    public int length()
5    public StringBuffer setLength(int newlength)
```

The capacity() method returns the current capacity of the StringBuffer. The ensureCapacity() method increases the capacity to atleast the size specified if it is less than

the specified value. The trimToSize() method, reduces the capacity of the StringBuffer to match the length of the StringBuffer. The setLength() method sets the length of the StringBuffer to the specified new length. This may result in either truncation of characters from the end or appending of \U0000 at the end.

The StringBuffer has methods indexOf() and lastIndexOf() to locate sub-strings within the character sequence of the StringBuffer; these are similar to their equivalent methods in the String class.

10.4 SUPPLEMENTARY CHARACTERS

From Java 5 onwards there is support for supplementary characters in strings. Unicode characters have a range from 0 to 10FFFF(Hex). In Java the char data type has only 16 bits. So, all unicode characters cannot be represented using a single char value. The characters which have a code point value greater than FFFF(Hex) are called the supplementary characters. These characters cannot be represented using a single char value. These characters can be encoded using UTF-16 encoding mechanism into two 16-bit values, which can be represented as a sequence of two char values inside any of the character sequences in Java. The UTF-16 encoding mechanism is explained in Appendix A. The supplementary characters get encoded as two char values, where the first value is a high surrogate value and the second value is a low surrogate value. The range for high surrogate is from D800 to DBFF(Hex), and the range for low surrogate is from DC00 to DFFF(Hex).

In Java the commonly used character sequences like String, StringBuffer and the StringBuilder have methods to decode any sequence of surrogate pair (high surrogate followed by a low surrogate) within the character sequence to get its code point value. The code point value for a unicode character in Java is therefore represented as an int and not as char. Some of the related methods commonly found in these character sequence classes are given in Listing 10.15.

Listing 10.15. Code point-related methods in character sequences

```
1    public int codePointAt(int index)
2    public int codePointBefore(int index)
3    public int codePointCount()
4    public int offsetByCodePoints(int index, int codePointOffset)
```

The method codePointAt() returns the int value of the character at the specified index position within the string. Here, if the character at the specified index is a high surrogate and the next character is a low surrogate then the surrogate pair is decoded to the value of the supplementary character, otherwise the character value of the char at the specified index is returned. The method codePointBefore() checks if the preceeding two characters (characters preceding the specified index position) form a surrogate pair. If it is a surrogate pair then this method returns the decoded codepoint of the unicode character represented by the surrogate pair, otherwise it returns the char value of the character at the preceding index position. The method codePointCount() counts the total number of unicode characters in the string, and the occurrence of a surrogate pair within the string is counted as one Unicode character. The offsetByCodePoints() method returns the index of the specified code point

value as an offset of the unicode characters, i.e. any occurrence of a surrogate pair is treated as a single index position, while calculating the offset index by the code point.

The `StringBuffer` has an additional method related to using the code point. The `appendCodePoint()` takes an `int` value and if the code point value passed as the parameter is a supplementary character then it appends two `char` values corresponding to the surrogate pair for the code point. If the code point is a 16-bit value then it appends a single character and if the code point is not a valid unicode character, then it would throw an `IllegalArgumentException`.

10.5 PASS BY VALUE AND PASS BY REFERENCE

What do we have in Java, pass by value or pass by reference? Let us look at an example to understand this.

We have a method defined as follows:

```
1 public void method(Account ac, int i) {
2     i += 100;
3     ac.deposit(10000);
4     i += 100;
5     ac = new CurrentAccount(1005, "Tom", 4000);
6     i += 100;
7     ac.deposit(3000);
8 }
```

This method is being used from another code as follows:

```
1 Account ac = new CurrentAccount(1005, "Tom", 2000);
2 int a = 100;
3 ac.deposit(3000);
4 method(ac, a);
5 System.out.println(ac.getBalance()+", "+a);
```

What will be the output from the above code? It prints ``15000, 100''.

In Java we have a pass by value. When we use primitive types as parameters then it is a pass by value; when we use the reference type as a parameter in a method then it becomes a pass by value for the reference value. The reference types refer to an object, so when we use the reference data type as a parameter, a copy of the reference value is made available to the invoked method, which enables the invoked method to work on the same object as the one referred by the invoking method.

Since primitive types are passed by value, the value of a is unchanged. In Line 1, balance in Account (ac) is 2000, in Line 3 balance in ac is 5000. Now the reference to ac is copied to the parameter ac in the method, so ac in method refers to the same object ac which is declared in Line 1. In Line 7, the balance would be 15000, now Line 9 makes ac in the method refer to a new `Object`, so the `Object` refered by ac in Line 1 is unaffected. Line 9 would make the method loose the reference to the object which was passed as a parameter in Line 4.

10.6 WRAPPER CLASSES

Corresponding to every primitive data type there is a class which can encapsulate its value. These classes are known as the wrapper classes. Wrapper classes corresponding to the primitive data types are given in Table 10.1.

All wrapper classes are `final` and immutable, i.e. we cannot create any sub-class of the wrapper classes, and an instance of wrapper class is more like a constant since the value encapsulated by an instance of wrapper cannot be modified.

All wrapper classes except for the `Character` class have two constructors. One constructor would take the corresponding primitive type as the parameter and another constructor, which would take a string as a parameter. In case of `Character` class, we have only one constructor which takes only `char` as the parameter. So all wrapper instances can be created by providing the primitive value to be encapsulated or by giving the string value. The constructor with the string would convert the string value to a primitive value.

Similar to the constructors, which can be used for creating any new instances of a wrapper class, we have a `static` method called `valueOf()` in all the wrapper classes, with similar arguments as the constructor. All the `valueOf()` methods return instances of the corresponding wrapper class. In case we use the `valueOf()` method a new instance of the wrapper is not necessarily created. The instance returned could be an instance from the cache of the instances for that wrapper class. In case the primitive value which is to be wrapped is not cached, it only then creates a new instance and returns the new instance. It is always preferred to use the `valueOf()` method than using a constructor for getting instances of the wrapper class. There are only two possible values for the `boolean` type, and both the values have been cached. These are also available as `public static final` instances from the `Boolean` class. The `Boolean` class contains two `public static final` variables called `TRUE` and `FALSE`. The wrapped values in the range from −128 to 127 are cached for the wrapper classes `Byte`, `Short` and `Integer`. The wrapped values in the range from 0 to 255 are cached for the wrapper class `Character`.

The wrapper classes are a place holder for a lot of static members to handle the primitive types. A few common members for all the wrapper classes are described below:

All wrapper classes have a `public static final` variable called `TYPE` which is of type `Class`. This variable is the `Class` object for the corresponding primitive type. Byte.TYPE is the same as byte.class. We have one more class which is not considered as a wrapper class but has this variable called `TYPE`.

TABLE 10.1 Primitive types and their wrapper classes

Primitive type	Wrapper class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`
`void`	`Void` (not a wrapper)

The Void class is not a wrapper class since it does not encapsulate any value.
All the wrapper classes also have another static and final variable of type int called SIZE, which gives the size of the corresponding primitive type in bytes, e.g. Integer.SIZE is the value 4.

10.6.1 Number Classes and their Methods

The wrapper classes for the numeric types have a common super-class called Number. Therefore, Number is a super-class for the classes Byte, Short, Integer, Long, Float and Double. The Number is an abstract class with some abstract methods. All these Number sub-classes have a common final variable named MAX_VALUE, MIN_VALUE, e.g. Byte.MAX_VALUE would be 127, Byte.MIN_VALUE would be −128. so we could use Integer.MAX_VALUE which is 2 to power 31 −1, which is 2147483247. For the floating-point types, the Float and the Double, what's the result of the following code?

```
1 double a = 75.0;
2 double b = 0.0;
3 double c = a / b;
```

What is the result in c? We have a representation of infinity values for the floating-point types.

```
1     Float.POSITIVE_INFINITY,  Float.NEGATIVE_INFINITY and Float.NaN.
2     Double.POSITIVE_INFINITY,  Double.NEGATIVE_INFINITY and Double.NaN.
```

For the integral types, i.e. byte, short, int and long, we have static methods which convert from String to primitive value and vice versa. The following static methods are available:

```
1     public static byte parseByte(String s, int radix) // the value
          of the radix could be anything from 2 - 36.
```

The characters allowed in the String depend on the radix value, if characters in the String are not according to the radix specified, it would result in an error condition.

The method is overloaded and we also have a method without the radix parameter, where the radix value is considered as 10.

```
1     public static byte parseByte(String s) // radix is 10.
```

We have methods to convert from primitive to the String type as follows:

```
1     public static String toString(int value, int radix)
```

e.g. String s = Integer.toString(255, 16); would return the string "FF".

10.6.2 Boxing and Unboxing Conversions

From Java 5 onwards there is an automatic conversion between primitive data types and their corresponding wrapper classes depending on the context. The boxing conversion is the conversion of a primitive value to its wrapper type, and the unboxing conversion is the conversion of a wrapper instance into its corresponding primitive value.

In case of an assignment like Integer someValue = 25; there is a boxing conversion which automatically takes place and the primitive value 25 gets boxed into a wrapper instance of the Integer class (using a valueOf() method). Similarly an unboxing conversion would take place when someone tries to assign the wrapper instance to a primitive variable. So, if we try to assign the Integer instance someValue to an int type then it would result in an unboxing conversion, e.g. int x = someValue; would result in x getting a value of int by using the intValue() method on the someValue instance. These conversions are carried out by the compiler automatically. We cannot access any member of the wrapper using a primitive-type expression, e.g. int y = x.int Value(), is not valid. Here we can use casting to make a boxing conversion and to get a wrapper instance and then use the method on the instance, e.g. int y = ((Integer)x).intValue(); is valid.

10.7 Math CLASS

The Math class in the java.lang package is a place holder for a lot of static methods. There are two important constants called PI and E, which have the constant values for *pi*, and Euler's constant *e*, useful in many mathematical computations. The Math class provides lots of methods for mathematical computations. Listing 10.16 gives the commonly used methods from the Math class.

Listing 10.16. Methods from Math class

```
1   public static abs(int i)
2   public static abs(long l)
3   public static abs(float f)
4   public static abs(double d)
5   public static max(int i1, int i2)
6   public static max(long l1, long l2)
7   public static max(float f1, float f2)
8   public static max(double d1, double d2)
9   public static min(int i1, int i2)
10  public static min(long l1, long l2)
11  public static min(float f1, float f2)
12  public static min(double d1, double d2)
13  public static exp(double a)
14  public static sqrt(double a)
15  public static cbrt(double a)
16  public static pow(double a, double b)
17  public static long round(double a)
18  public static int round(float a)
19  public static double sin(double a)
20  public static double cos(double a)
21  public static double tan(double a)
```

```
22    public static double asin(double a)
23    public static double acos(double a)
24    public static double atan(double a)
25    public static double ceil(double a)
26    public static double floor(double a)
```

10.7.1 Numeric Values Requiring more than 64 Bits

For mathematical calculations which require more than 64 bits, we have an additional package java.math, which has the `BigInteger` and `BigDecimal` classes. The instances of `BigInteger` can be used to create integral values which require more than 64 bits, and the instances of `BigDecimal` can be used for floating-point values which require more than 64 bits. These classes provide the methods for the common numeric operations of such values. These two classes are also sub-classed from the `Number` class.

LESSONS LEARNED

- A class whose instances have a natural ordering should implement the `Comparable` interface.
- Various implementations of the `Comparator` interface may be provided to define different kinds of orderings for instances of any class based on different criteria.
- Instances of the `String` class are immutable, and every string concatenation operation results in a new instance of the `String` class being created. `StringBuffer` and the `StringBuilder` are the mutable classes for having a character sequence.
- The `String` class supports the supplementary characters of unicode, which are specified in a `String` instance by using a surrogate pair. A high surrogate value followed by a low surrogate value will be treated as a single unicode character. It has methods to derive the code point value from the surrogate pair.
- A case-insensitive comparision for strings is also supported by using the `Comparator` called `CASE_INSENSITIVE_ORDER` available as a `static` member in the `String` class.
- The `String` class has methods to support text operations based on regular expressions.
- Java always uses a pass by value for primitive types. In case of reference type the value of the reference is copied to the parameter variables of the method being invoked.
- Corresponding to each of the primitive data types, there is a final immutable class called the wrapper class, which encapsulates the values of the corresponding primitive type.
- From Java 5 onwards there is a support for boxing and unboxing conversions between the primitive and the wrapper types.
- The `Math` class provides methods for various kinds of mathematical functions.
- There is a support for handling very large numerical values using the classes `BigInteger` and the `BigDecimal` from the java.math package.

EXERCISES

1. State which of the following are true or false:
   (a) We can create our own sub-class of the `java.lang.String` class.
   (b) `Integer` is a primitive data type in Java.
   (c) The method `Double.parseDouble()` converts a `String` to a primitive `double` value.
   (d) Instances of `StringBuffer` class are immutable.
   (e) Numeric values more than 64 bits cannot be handled in Java.
   (f) All unicode characters are in the range from 0 to 0xFFFF.
2. Fill in the blanks in the following:
   (a) _____ is a method in the `String` class which returns the number of characters in the `String` instance.
   (b) The _____ method of `Integer` class is used to convert `int` to `String`.
   (c) The _____ method of `Integer` class is used to convert `int` to `String` in hexadecimal form.
   (d) The _____ method of `Integer` class is used to convert `int` to `String` in binary form.
   (e) The _____ method of `Integer` class is used to convert `int` to `String` in octal form.
   (f) The method used for unwrapping from an instance of integer is _____.
3. What are the possible results for a floating-point division by zero? (Include results for `float` and `double` types.)
4. Explain the difference between `Comparable` and the `Comparator` interfaces.
5. List the names of all the wrapper classes in Java.
6. Explain the boxing and unboxing conversions.
7. Write a Java application to print a pyramid by using the characters from the `String` taken as a command line argument. The pyramid should print the first character in the first line, first two characters in the second line and so on.
8. What is the output of the following code snippet?

```
1 public static void main(String[] args) {
2     String s = "\u0905\u0906";
3     System.out.println(s.length());
4 }
```

   (a) 12
   (b) 8
   (c) 2
   (d) Does not compile
   (e) None of the above

9. What is the output of the following code?

```
1  public class Test {
2      public static void main(String[] args) {
3          String s = "Hello\uD800\uDC00\u0905\u0906";
4          System.out.println(s.length());
5          System.out.println(s.codePointCount(0, s.length()));
6      }
7  }
```

(a) 29
29

(b) 29
27

(c) 9
8

(d) 9
9

(e) 25
8

(f) Compilation error

(g) None of the above