Fundamental Classes

10

Exam Objectives

- Write code using the following methods of the `java.lang.Math` class: `abs()`, `ceil()`, `floor()`, `max()`, `min()`, `random()`, `round()`, `sin()`, `cos()`, `tan()`, `sqrt()`.
- Describe the significance of the immutability of `String` objects.
- Describe the significance of wrapper classes, including making appropriate selections in the wrapper classes to suit specified behavior requirements, stating the result of executing a fragment of code that includes an instance of one of the wrapper classes, and writing code using the following methods of the wrapper classes (e.g., `Integer`, `Double`, etc.):
  - `doubleValue()`
  - `floatValue()`
  - `intValue()`
  - `longValue()`
  - `parse`*`Xxx`*`()`
  - `get`*`Xxx`*`()`
  - `toString()`
  - `toHexString()`

Supplementary Objectives

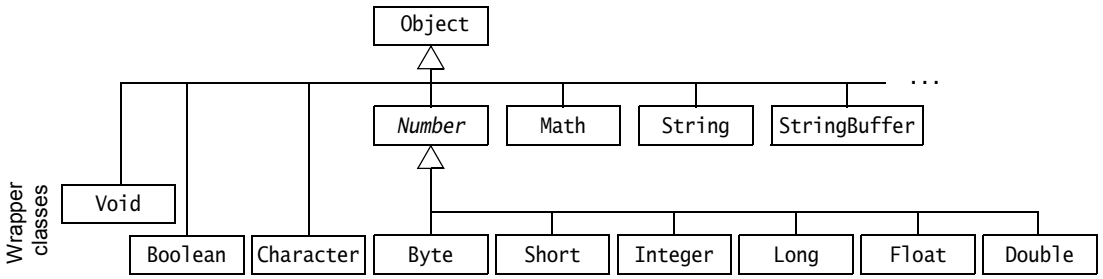
- Understand the functionality inherited by all classes from the `Object` class, which is at the top of any class hierarchy.
- Write code for manipulating immutable and dynamic strings, using the facilities provided by the `String` and `StringBuffer` classes, respectively.

10.1 Overview of the `java.lang` Package

The `java.lang` package is indispensable when programming in Java. It is automatically imported into every source file at compile time. The package contains the `Object` class that is the mother of all classes, and the wrapper classes (`Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`) used to handle primitive values as objects. It provides classes essential for interacting with the JVM (`Runtime`), for security (`SecurityManager`), for loading classes (`ClassLoader`), for dealing with threads (`Thread`), and for exceptions (`Throwable`). The `java.lang` package also contains classes that provide the standard input, output, and error streams (`System`), string handling (`String`, `StringBuffer`), and mathematical functions (`Math`).

Figure 10.1 shows the important classes that are discussed in detail in this chapter.

Figure 10.1 *Partial Inheritance Hierarchy in the* `java.lang` *Package*



10.2 The `Object` Class

All classes extend the `Object` class, either directly or indirectly. A class declaration, without the `extends` clause, implicitly extends the `Object` class (see Section 6.1, p. 226). Thus, the `Object` class is always at the top of any inheritance hierarchy. The `Object` class defines the basic functionality that all objects exhibit and that all classes inherit. Note that this also applies for arrays, since these are genuine objects in Java.

The `Object` class provides the following general utility methods (see Example 10.1 for usage of some of these methods):

```
int hashCode()
```

When storing objects in hash tables, this method can be used to get a hash value for an object. This value is guaranteed to be consistent during the execution of the program. For a detailed discussion of the `hashCode()` method, see Section 11.7 on page 461.

`boolean equals(Object obj)`

Object reference and value equality are discussed together with the == and != operators (see Section 3.10, p. 68). The `equals()` method in the `Object` class returns `true` only if the two references compared denote the same object. The `equals()` method is usually overridden to provide the semantics of object value equality, as is the case for the wrapper classes and the `String` class. For a detailed discussion of the `equals()` method, see Section 11.7 on page 461.

`final Class getClass()`

Returns the *runtime class* of the object, which is represented by an object of the class `java.lang.Class` at runtime.

`protected Object clone() throws CloneNotSupportedException`

New objects that are exactly the same (i.e., have identical states) as the current object can be created by using the `clone()` method, that is, primitive values and reference values are copied. This is called *shallow copying*. A class can override this method to provide its own notion of cloning. For example, cloning a composite object by recursively cloning the constituent objects is called *deep copying*.

When overridden, the method in the subclass is usually declared `public` to allow any client to clone objects of the class.

If the overriding `clone()` method relies on the `clone()` method in the `Object` class, then the subclass must implement the `Cloneable` marker interface to indicate that its objects can be safely cloned. Otherwise, the `clone()` method in the `Object` class will throw a checked `CloneNotSupportedException`.

`String toString()`

If a subclass does not override this method, it returns a textual representation of the object, which has the following format:

"<*name of the class*>@<*hash code value of object*>"

This method is usually overridden and used for debugging purposes. The method call `System.out.println(objRef)` will implicitly convert its argument to a textual representation using the `toString()` method. See also the binary string concatenation operator +, discussed in Section 3.6 on page 62.

`protected void finalize() throws Throwable`

This method is discussed in connection with garbage collection (see Section 8.1, p. 324). It is called on an object just before it is garbage collected, so that any cleaning up can be done. However, the default `finalize()` method in the `Object` class does not do anything useful.

In addition, the `Object` class provides support for thread communication in synchronized code, through the following methods, which are discussed in Section 9.5 on page 370:

```
final void wait(long timeout) throws InterruptedException
final void wait(long timeout, int nanos) throws InterruptedException
final void wait() throws InterruptedException
final void notify()
final void notifyAll()
```

A thread invokes these method on the object whose lock it holds. A thread waits for notification by another thread.

Example 10.1 *Methods in the* `Object` *class*

```
public class ObjectMethods {
    public static void main(String[] args) {
        // Two objects of MyClass.
        MyClass obj1 = new MyClass();
        MyClass obj2 = new MyClass();

        // Two strings.
        String str1 = new String("WhoAmI");
        String str2 = new String("WhoAmI");

        // Method hashCode() overridden in String class.
        // Strings with same content (i.e., are equal) have the same hash code.
        System.out.println("hash code for str1: " + str1.hashCode());
        System.out.println("hash code for str2: " + str2.hashCode() + "\n");

        // Hash codes are different for different MyClass objects.
        System.out.println("hash code for MyClass obj1: " + obj1.hashCode());
        System.out.println("hash code for MyClass obj2: " + obj2.hashCode()+"\n");

        // Method equals() overridden in the String class.
        System.out.println("str1.equals(str2): " + str1.equals(str2));
        System.out.println("str1 == str2 : " + (str1 == str2) + "\n");

        // Method equals() from the Object class called.
        System.out.println("obj1.equals(obj2): " + obj1.equals(obj2));
        System.out.println("obj1 == obj2 : " + (obj1 == obj2) + "\n");

        // The runtime object that represents the class of an object.
        Class rtStringClass  = str1.getClass();
        Class rtMyClassClass = obj1.getClass();
        // The name of the class represented by the runtime object.
        System.out.println("Class for str1: " + rtStringClass);
        System.out.println("Class for obj1: " + rtMyClassClass + "\n");
```

```
        // The toString() method is overridden in the String class.
        String textRepStr = str1.toString();
        String textRepObj = obj1.toString();
        System.out.println("Text representation of str1: " + textRepStr);
        System.out.println("Text representation of obj1: " + textRepObj + "\n");

        // Shallow copying of arrays.
        MyClass[] array1 = {new MyClass(), new MyClass(), new MyClass()};
        MyClass[] array2 = (MyClass[]) array1.clone(); // Cast required.
        // Array objects are different, but share the element objects.
        System.out.println("array1 == array2 : " + (array1 == array2));
        for(int i = 0; i < array1.length; i++) {
            System.out.println("array1[" + i + "] == array2[" + i + "] : "
                        + (array1[i] == array2[i]));
        }
        System.out.println();

        // Clone an object of MyClass.
        MyClass obj3 = (MyClass) obj1.clone();
        System.out.println("hash code for MyClass obj3: " + obj3.hashCode());
        System.out.println("obj1 == obj3 : " + (obj1 == obj3));
    }
}

class MyClass implements Cloneable {

    public Object clone() {
        Object obj = null;
        try { obj = super.clone();}        // Calls overridden method.
        catch (CloneNotSupportedException e) { System.out.println(e);}
        return obj;
    }
}
```

Output from the program:

```
hash code for str1: -1704812257
hash code for str2: -1704812257

hash code for MyClass obj1: 24216257
hash code for MyClass obj2: 20929799

str1.equals(str2): true
str1 == str2 : false

obj1.equals(obj2): false
obj1 == obj2 : false

Class for str1: class java.lang.String
Class for obj1: class MyClass

Text representation of str1: WhoAmI
Text representation of obj1: MyClass@17182c1
```

```
array1 == array2 : false
array1[0] == array2[0] : true
array1[1] == array2[1] : true
array1[2] == array2[2] : true

hash code for MyClass obj3: 16032330
obj1 == obj3 : false
```

Review Questions

10.1 What is the return type of the `hashCode()` method in the `Object` class?

Select the one correct answer.

(a) `String`
(b) `int`
(c) `long`
(d) `Object`
(e) `Class`

10.2 Which statement is true?

Select the one correct answer.

(a) If the references `x` and `y` denote two different objects, then the expression `x.equals(y)` is always `false`.
(b) If the references `x` and `y` denote two different objects, then the expression `(x.hashCode() == y.hashCode())` is always `false`.
(c) The `hashCode()` method in the `Object` class is declared `final`.
(d) The `equals()` method in the `Object` class is declared `final`.
(e) All arrays have a method named `clone`.

10.3 Which exception can the `clone()` method of the `Object` class throw?

Select the one correct answer.

(a) `CloneNotSupportedException`
(b) `NotCloneableException`
(c) `IllegalCloneException`
(d) `NoClonesAllowedException`

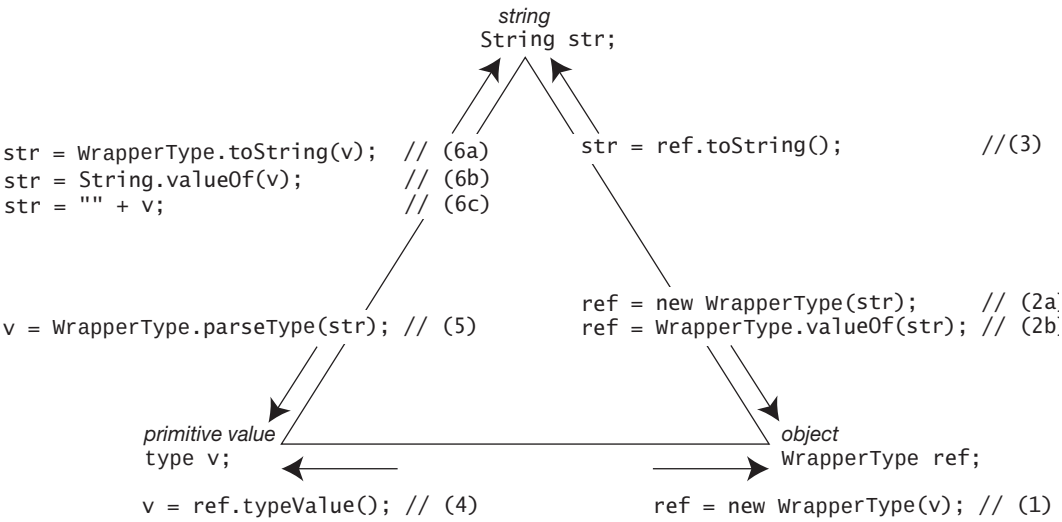
10.3 The Wrapper Classes

Wrapper classes were introduced with the discussion of the primitive data types (see Section 2.2, p. 28). Primitive values in Java are not objects. In order to manipulate these values as objects, the `java.lang` package provides a *wrapper* class for each of the primitive data types. All wrapper classes are `final`. The objects of all wrapper classes that can be instantiated are *immutable*, that is, their state cannot be changed.

Although the `Void` class is considered a wrapper class, it does not wrap any primitive value and is not instantiable (i.e., has no `public` constructors). It just denotes the `Class` object representing the keyword `void`. The `Void` class will not be discussed further in this section.

In addition to the methods defined for constructing and manipulating objects of primitive values, the wrapper classes also define useful constants, fields, and conversion methods.

Figure 10.2 *Converting Values between Primitive, Wrapper, and* `String` *Types*



type *is*:	WrapperType *is*:	*Comments:*
boolean	Boolean	(1) For `boolean` type, can also use `Boolean.valueOf(boolean b)`.
char	Character	(2a) Not for `Character` type. Can throw `NumberFormatException`.
byte	Byte	(2b) Not for `Character` type. Can throw `NumberFormatException`.
short	Short	(5) For numeric wrapper types only. Can throw `NumberFormatException`.
int	Integer	(6b) Not for `byte` and `short` primitive types.
long	Long	
float	Float	
double	Double	

Common Wrapper Class Constructors

The `Character` class has only one `public` constructor, taking a `char` value as parameter. The other wrapper classes all have two `public` one-argument constructors: one takes a primitive value and the other takes a string.

```
WrapperType( type v )
WrapperType( String str )
```

Converting Primitive Values to Wrapper Objects

A constructor that takes a primitive value can be used to create wrapper objects. See (1) in Figure 10.2.

```
Character charObj1   = new Character('\n');
Boolean   boolObj1   = new Boolean(true);
Integer   intObj1    = new Integer(2003);
Double    doubleObj1 = new Double(3.14);
```

Converting Strings to Wrapper Objects

A constructor that takes a `String` object representing the primitive value, can also be used to create wrapper objects. The constructors for the numeric wrapper types throw an unchecked `NumberFormatException` if the `String` parameter does not parse to a valid number. See (2a) in Figure 10.2.

```
Boolean boolObj2   = new Boolean("TrUe");       // case ignored: true
Boolean boolObj3   = new Boolean("XX");         // false
Integer intObj2    = new Integer("2003");
Double  doubleObj2 = new Double("3.14");
Long    longObj1   = new Long("3.14");          // NumberFormatException
```

Common Wrapper Class Utility Methods

Converting Strings to Wrapper Objects

Each wrapper class (except `Character`) defines the static method `valueOf(String s)` that returns the wrapper object corresponding to the primitive value represented by the `String` object passed as argument (see (6a) in Figure 10.2). This method for the numeric wrapper types also throws a `NumberFormatException` if the `String` parameter is not a valid number.

```
static WrapperType valueOf( String s )
```

```
Boolean boolObj4   = Boolean.valueOf("false");
Integer intObj3    = Integer.valueOf("1949");
Double  doubleObj3 = Double.valueOf("-3.0");
```

In addition to the one-argument `valueOf()` method, the integer wrapper types define an overloaded static `valueOf()` method that can take a second argument. This argument specifies the base (or *radix*) in which to interpret the string representing the signed integer in the first argument:

```
static WrapperType valueOf( String s, int base )
```

```
Byte    byteObj1  = Byte.valueOf("1010", 2);   // Decimal value 10
Short   shortObj2 = Short.valueOf("012", 8);   // Not "\012". Decimal value 10.
Integer intObj4   = Integer.valueOf("-a", 16); // Not "-0xa". Decimal value -10.
Long    longObj2  = Long.valueOf("-a", 16);    // Not "-0xa". Decimal value -10L.
```

Converting Wrapper Objects to Strings

Each wrapper class overrides the `toString()` method from the `Object` class. The overriding method returns a `String` object containing the string representation of the primitive value in the wrapper object (see (3) in Figure 10.2).

```
String toString()

String charStr   = charObj1.toString();     // "\n"
String boolStr   = boolObj2.toString();     // "true"
String intStr    = intObj1.toString();      // "2003"
String doubleStr = doubleObj1.toString();   // "3.14"
```

Converting Primitive Values to Strings

Each wrapper class defines a static method `toString(`*type* `v)` that returns the string corresponding to the primitive value of *type* passed as argument (see (6a) in Figure 10.2).

```
static String toString( type v )

String charStr2   = Character.toString('\n');  // "\n"
String boolStr2   = Boolean.toString(true);    // "true"
String intStr2    = Integer.toString(2003);    // Base 10. "2003"
String doubleStr2 = Double.toString(3.14);     // "3.14"
```

For integer primitive types, the base is assumed to be 10. For floating-point numbers, the textual representation (decimal form or scientific notation) depends on the sign and the magnitude (absolute value) of the number. The NaN value, positive infinity and negative infinity will result in the strings `"NaN"`, `"Infinity"`, and `"-Infinity"`, respectively.

In addition, the wrapper classes `Integer` and `Long` define overloaded `toString()` methods for converting integers to string representation in decimal, binary, octal, and hexadecimal notation (see p. 398).

Converting Wrapper Objects to Primitive Values

Each wrapper class defines a *type*`Value()` method which returns the primitive value in the wrapper object (see (4) in Figure 10.2).

```
type typeValue()

char    c = charObj1.charValue();           // '\n'
boolean b = boolObj2.booleanValue();        // true
int     i = intObj1.intValue();             // 2003
double  d = doubleObj1.doubleValue();       // 3.14
```

In addition, each numeric wrapper class defines *type*`Value()` methods for converting the primitive value in the wrapper object to a value of any numeric primitive data type. These methods are discussed below.

Wrapper Comparison, Equality, and Hashcode

Each wrapper class (except `Boolean`) defines the following method:

```
int compareTo(WrapperType obj2)
```

that returns a value which is less than, equal to, or greater than zero, depending on whether the primitive value in the current *`WrapperType`* object is less than, equal to, or greater than the primitive value in the *`WrapperType`* object denoted by argument `obj2`.

Each wrapper class (except `Boolean`) also implements the `Comparable` interface (see Section 11.6, p. 453), which defines the following method:

```
int compareTo(Object obj2)
```

This method is equivalent to the `compareTo(`*`WrapperType`*`)` method when the current object and the object denoted by the argument `obj2` have the same *`WrapperType`*. Otherwise, a `ClassCastException` is thrown.

```
// Comparisons based on objects created above
Character charObj2   = new Character('a');
int result1 = charObj1.compareTo(charObj2);      //  < 0
int result2 = intObj1.compareTo(intObj3);        //  > 0
int result3 = doubleObj1.compareTo(doubleObj2);  // == 0
int result4 = doubleObj1.compareTo(intObj1);     // ClassCastException
```

Each wrapper class overrides the `equals()` method from the `Object` class. The overriding method compares two wrapper objects for object value equality.

```
boolean equals(Object obj2)
```

```
// Comparisons based on objects created above
boolean charTest   = charObj1.equals(charObj2);      // false
boolean boolTest   = boolObj2.equals(Boolean.FALSE); // false
boolean intTest    = intObj1.equals(intObj2);        // true
boolean doubleTest = doubleObj1.equals(doubleObj2);  // true
```

Each wrapper class overrides the `hashCode()` method in the `Object` class. The overriding method returns a hash value based on the primitive value in the wrapper object.

```
int hashCode()
```

```
int index = charObj1.hashCode();
```

Numeric Wrapper Classes

The numeric wrapper classes `Byte`, `Short`, `Integer`, `Long`, `Float`, and `Double` are all subclasses of the `abstract` class `Number` (see Figure 10.1).

Each numeric wrapper class defines an assortment of constants, including the minimum and maximum value of the corresponding primitive data type:

```
<wrapper class name>.MIN_VALUE
<wrapper class name>.MAX_VALUE
```

The following code retrieves the minimum and maximum values of various numeric types:

```
byte   minByte   = Byte.MIN_VALUE;     // -128
int    maxInt    = Integer.MAX_VALUE;  // 2147483647
double maxDouble = Double.MAX_VALUE;   // 1.7976931348623157e+308
```

Converting any Numeric Wrapper Object to any Numeric Primitive Type

Each numeric wrapper class defines the following set of *type*`Value()` methods for converting the primitive value in the wrapper object to a value of any numeric primitive type:

```
byte   byteValue()
short  shortValue()
int    intValue()
long   longValue()
float  floatValue()
double doubleValue()
```

See also (4) in Figure 10.2.

The following code shows converting of values in numeric wrapper objects to any numeric primitive type.

```
Byte    byteObj2   = new Byte((byte) 16);       // Cast mandatory
Integer intObj5    = new Integer(42030);
Double  doubleObj4 = new Double(Math.PI);

short  shortVal  = intObj5.shortValue();        // (1)
long   longVal   = byteObj2.longValue();
int    intVal    = doubleObj4.intValue();       // (2) Truncation
double doubleVal = intObj5.doubleValue();
```

Notice the potential for loss of information at (1) and (2) above, when the primitive value in a wrapper object is converted to a narrower primitive data type.

Converting Strings to Numeric Values

Each numeric wrapper class defines a static method `parse`*Type*`(String s)`, which returns the primitive numeric value represented by the `String` object passed as argument. The *Type* in the method name `parse`*Type* stands for the name of a numeric wrapper class, except for the name of the `Integer` class which is abbreviated to `Int`. These methods throw a `NumberFormatException` if the `String` parameter is not a valid argument (see (5) in Figure 10.2.)

```
type parseType(String s)
```

```
byte   value1 = Byte.parseByte("16");
int    value2 = Integer.parseInt("2010");       // parseInt, not parseInteger.
int    value3 = Integer.parseInt("7UP");        // NumberFormatException
double value4 = Double.parseDouble("3.14");
```

For the integer wrapper types, the overloaded static method parse*Type*() can, in addition, take a second argument, which can specify the base in which to interpret the string representing the signed integer in the first argument:

```
type parseType(String s, int base)
```

```
byte  value6 = Byte.parseByte("1010", 2);    // Decimal value 10
short value7 = Short.parseShort("012", 8);   // Not "\012". Decimal value 10.
int   value8 = Integer.parseInt("-a", 16);   // Not "-0xa". Decimal value -10.
long  value9 = Long.parseLong("-a", 16);     // Not "-0xa". Decimal value -10L.
```

Converting Integer Values to Strings in different Notations

The wrapper classes `Integer` and `Long` provide static methods for converting integers to string representation in decimal, binary, octal, and hexadecimal notation. Some of these methods from the `Integer` class are listed here, but analogous methods are also defined in the `Long` class. Example 10.2 demonstrates use of these methods.

```
static String toBinaryString(int i)
static String toHexString(int i)
static String toOctalString(int i)
```

These three methods return a string representation of the integer argument as an *unsigned* integer in base 2, 16, and 8, respectively, with no extra leading zeroes.

```
static String toString(int i, int base)
static String toString(int i)
```

The first method returns the minus sign `'-'` as the first character if the integer `i` is negative. In all cases, it returns the string representation of the *magnitude* of the integer `i` in the specified base.

The last method is equivalent to the method `toString(int i, int base)`, where the base has the value 10, that is, returns the string representation as a signed decimal. (see also (6a) in Figure 10.2).

Example 10.2 *String Representation of Integers*

```
public class IntegerRepresentation {
    public static void main(String[] args) {
        int positiveInt = +41;    // 051, 0x29
        int negativeInt = -41;    // 037777777727, -051, 0xffffffd7, -0x29
```

```
        System.out.println("String representation for decimal value: "
                           + positiveInt);
        integerStringRepresentation(positiveInt);
        System.out.println("String representation for decimal value: "
                           + negativeInt);
        integerStringRepresentation(negativeInt);
    }

    public static void integerStringRepresentation(int i) {
        System.out.println("    Binary:\t\t" + Integer.toBinaryString(i));
        System.out.println("    Hex:\t\t"    + Integer.toHexString(i));
        System.out.println("    Octal:\t\t"  + Integer.toOctalString(i));
        System.out.println("    Decimal:\t"  + Integer.toString(i));

        System.out.println("    Using toString(int i, int base) method:");
        System.out.println("    Base 2:\t\t" + Integer.toString(i, 2));
        System.out.println("    Base 16:\t"  + Integer.toString(i, 16));
        System.out.println("    Base 8:\t\t" + Integer.toString(i, 8));
        System.out.println("    Base 10:\t"  + Integer.toString(i, 10));
    }
}
```

Output from the program:

```
String representation for decimal value: 41
    Binary:     101001
    Hex:        29
    Octal:      51
    Decimal:    41
    Using toString(int i, int base) method:
    Base 2:     101001
    Base 16:    29
    Base 8:     51
    Base 10:    41
String representation for decimal value: -41
    Binary:     11111111111111111111111111010111
    Hex:        ffffffd7
    Octal:      37777777727
    Decimal:    -41
    Using toString(int i, int base) method:
    Base 2:     -101001
    Base 16:    -29
    Base 8:     -51
    Base 10:    -41
```

`Character` Class

The `Character` class defines a myriad of constants, including the following which represent the minimum and the maximum value of the `char` type (see Section 2.2, p. 29):

```
Character.MIN_VALUE
Character.MAX_VALUE
```

The `Character` class also defines a plethora of static methods for handling various attributes of a character, and case issues relating to characters, as defined by Unicode:

```
static int     getNumericValue(char ch)
static boolean isLowerCase(char ch)
static boolean isUpperCase(char ch)
static boolean isTitleCase(char ch)
static boolean isDigit(char ch)
static boolean isLetter(char ch)
static boolean isLetterOrDigit(char ch)
static char    toUpperCase(char ch)
static char    toLowerCase(char ch)
static char    toTitleCase(char ch)
```

The following code converts a lowercase character to an uppercase character:

```
char ch = 'a';
if (Character.isLowerCase(ch)) ch = Character.toUpperCase(ch);
```

`Boolean` Class

The `Boolean` class defines the following wrapper objects to represent the primitive values `true` and `false`, respectively:

```
Boolean.TRUE
Boolean.FALSE
```

Review Questions

10.4 Which of the following are wrapper classes?

Select the three correct answers.

(a) `java.lang.Void`
(b) `java.lang.Int`
(c) `java.lang.Boolean`
(d) `java.lang.Long`
(e) `java.lang.String`

10.5 Which of the following classes do not extend the `java.lang.Number` class?

Select the two correct answers.

(a) `java.lang.Float`
(b) `java.lang.Byte`
(c) `java.lang.Character`
(d) `java.lang.Boolean`
(e) `java.lang.Short`

10.6 Which of these classes define immutable objects?

Select the three correct answers.

(a) `Character`
(b) `Byte`
(c) `Thread`
(d) `Short`
(e) `Object`

10.7 Which of these classes have a one-parameter constructor taking a string?

Select the two correct answers.

(a) `Void`
(b) `Integer`
(c) `Boolean`
(d) `Character`
(e) `Object`

10.8 Which of the wrapper classes have a `booleanValue()` method?

Select the one correct answer.

(a) All wrapper classes.
(b) All wrapper classes except `Void`.
(c) All wrapper classes that also implement the `compareTo()` method.
(d) All wrapper classes extending `Number`.
(e) Only the class `Boolean`.

10.9 Which statements are true about wrapper classes?

Select the two correct answers.

(a) `String` is a wrapper class.
(b) `Double` has a `compareTo()` method.
(c) `Character` has a `intValue()` method.
(d) `Byte` extends `Number`.

10.4 The `Math` Class

The `final` class `Math` defines a set of `static` methods to support common mathematical functions, including functions for rounding numbers, performing trigonometry, generating pseudo random numbers, finding maximum and minimum of two numbers, calculating logarithms and exponentiation. The `Math` class cannot be instantiated. Only the class name `Math` can be used to invoke the static methods.

The `final` class `Math` provides constants to represent the value of *e*, the base of the natural logarithms, and the value π (*pi*), the ratio of the circumference of a circle to its diameter:

```
Math.E
Math.PI
```

Miscellaneous Rounding Functions

```
static int    abs(int i)
static long   abs(long l)
static float  abs(float f)
static double abs(double d)
```

The overloaded method `abs()` returns the absolute value of the argument. For a non-negative argument, the argument is returned. For a negative argument, the negation of the argument is returned.

```
static int    min(int a, int b)
static long   min(long a, long b)
static float  min(float a, float b)
static double min(double a, double b)
```

The overloaded method `min()` returns the smaller of the two values `a` and `b` for any numeric type.

```
static int    max(int a, int b)
static long   max(long a, long b)
static float  max(float a, float b)
static double max(double a, double b)
```

The overloaded method `max()` returns the greater of the two values `a` and `b` for any numeric type.

The following code illustrates the use of these methods from the `Math` class:

```
long   ll = Math.abs(2010L);             // 2010L
double dd = Math.abs(-Math.PI);          // 3.141592653589793

double d1 = Math.min(Math.PI, Math.E);   // 2.718281828459045
long   m1 = Math.max(1984L, 2010L);      // 2010L
int    i1 = (int) Math.max(3.0, 4);      // Cast required.
```

Note the cast required in the last example. The method with the signature `max(double, double)` is executed, with implicit conversion of the `int` argument to a `double`. Since this method returns a `double`, it must be explicitly cast to an `int`.

```
static double ceil(double d)
```

The method `ceil()` returns the *smallest* `double` value that is *greater than or equal* to the argument `d`, and is equal to a mathematical integer.

```
static double floor(double d)
```

The method `floor()` returns the *largest* `double` value that is *less than or equal* to the argument `d`, and is equal to a mathematical integer.

```
static int  round(float f)
static long round(double d)
```

The overloaded method `round()` returns the integer closest to the argument. This is equivalent to adding 0.5 to the argument, taking the floor of the result, and casting it to the return type. This is not the same as rounding to a specific number of decimal places, as the name of the method might suggest.

If the fractional part of a *positive* argument is *less than* 0.5, then the result returned is the same as `Math.floor()`. If the fractional part of a positive argument is *greater than or equal* to 0.5, then the result returned is the same as `Math.ceil()`.

If the fractional part of a *negative* argument is *less than or equal* to 0.5, then the result returned is the same as `Math.ceil()`. If the fractional part of a negative argument is *greater than* 0.5, then the result returned is the same as `Math.floor()`.

It is important to note the result obtained on negative arguments, keeping in mind that a negative number whose absolute value is less than that of another negative number, is actually greater than the other number (e.g., –3.2 is greater than –4.7). Compare also the results returned by these methods, shown in Table 10.1.

```
double upPI    = Math.ceil(Math.PI);        // 4.0
double downPI  = Math.floor(Math.PI);       // 3.0
long   roundPI = Math.round(Math.PI);       // 3L

double upNegPI    = Math.ceil(-Math.PI);    // -3.0
double downNegPI  = Math.floor(-Math.PI);   // -4.0
long   roundNegPI = Math.round(-Math.PI);   // -3L
```

Table 10.1 *Applying Rounding Functions*

Argument:	7.0	7.1	7.2	7.3	7.4	7.5	7.6	7.7	7.8	7.9	8.0
ceil:	7.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0
floor:	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	8.0
round:	7	7	7	7	7	8	8	8	8	8	8
Argument:	**-7.0**	**-7.1**	**-7.2**	**-7.3**	**-7.4**	**-7.5**	**-7.6**	**-7.7**	**-7.8**	**-7.9**	**-8.0**
ceil:	-7.0	-7.0	-7.0	-7.0	-7.0	-7.0	-7.0	-7.0	-7.0	-7.0	-8.0
floor:	-7.0	-8.0	-8.0	-8.0	-8.0	-8.0	-8.0	-8.0	-8.0	-8.0	-8.0
round:	-7	-7	-7	-7	-7	-7	-8	-8	-8	-8	-8

Exponential Functions

`static double pow(double d1, double d2)`

The method `pow()` returns the value of `d1` raised to the power of `d2` (i.e., $d1^{d2}$).

`static double exp(double d)`

The method `exp()` returns the exponential number e raised to the power of `d` (i.e., e^{d}).

`static double log(double d)`

The method `log()` returns the natural logarithm (base e) of `d` (i.e., $log_e d$).

`static double sqrt(double d)`

The method `sqrt()` returns the square root of `d` (i.e., $d^{0.5}$). For a `NaN` or a negative argument, the result is a `NaN` (see Section 3.5, p. 52).

Some examples of exponential functions:

```
double r = Math.pow(2.0, 4.0);           // 16.0
double v = Math.exp(2.0);                // 7.38905609893065
double l = Math.log(Math.E);             // 0.9999999999999981
double c = Math.sqrt(3.0*3.0 + 4.0*4.0); // 5.0
```

Trigonometry Functions

`static double sin(double d)`

The method `sin()` returns the trigonometric sine of an angle `d` specified in radians.

`static double cos(double d)`

The method `cos()` returns the trigonometric cosine of an angle `d` specified in radians.

`static double tan(double d)`

The method `tan()` returns the trigonometric tangent of an angle `d` specified in radians.

`static double toRadians(double degrees)`

The method `toRadians()` converts an angle in degrees to its approximation in radians.

`static double toDegrees(double radians)`

The method `toRadians()` converts an angle in radians to its approximation in degrees.

Some examples of trigonometry functions:

```
double deg1  = Math.toDegrees(Math.PI/4.0);  // 45 degrees
double deg2  = Math.toDegrees(Math.PI/2.0);  // 90 degrees

double rad1  = Math.toRadians(deg1);         // 0.7853981633974483
double rad2  = Math.toRadians(deg2);         // 1.5707963267948966

double r1 = Math.sin(Math.PI/2.0);           // 1.0
double r2 = Math.sin(Math.PI*2);             // -2.4492935982947064E-16 (0.0)

double r3 = Math.cos(Math.PI);               // -1.0
double r4 = Math.cos(Math.toRadians(360.0)); // 1.0

double r5 = Math.tan(Math.toRadians(90.0));  // 1.633123935319537E16 (infinity)
double r6 = Math.tan(Math.toRadians(45.0));  // 0.9999999999999999   (1.0)
```

Expected mathematical values are shown in parentheses.

Pseudorandom Number Generator

> `static double random()`
>
> The method `random()` returns a random number greater than or equal to 0.0 and less than 1.0, where the value is selected randomly from the range according to a uniform distribution.

An example of using the pseudorandom number generator is as follows:

```
for (int i = 0; i < 10; i++)
    System.out.println((int)(Math.random()*10));   // int values in range [0 .. 9].
```

The loop will generate a run of ten pseudorandom integers between 0 (inclusive) and 10 (exclusive).

Review Questions

10.10 Given the following program, which lines will print 11 exactly?

```
class MyClass {
    public static void main(String[] args) {
        double v = 10.5;

        System.out.println(Math.ceil(v));        // (1)
        System.out.println(Math.round(v));       // (2)
        System.out.println(Math.floor(v));       // (3)
        System.out.println((int) Math.ceil(v));  // (4)
        System.out.println((int) Math.floor(v)); // (5)
    }
}
```

Select the two correct answers.

(a) The line labeled (1).
(b) The line labeled (2).
(c) The line labeled (3).
(d) The line labeled (4).
(e) The line labeled (5).

10.11 Which method is not defined in the `Math` class?

Select the one correct answer.

(a) `double tan2(double)`
(b) `double cos(double)`
(c) `int abs(int a)`
(d) `double ceil(double)`
(e) `float max(float, float)`

10.12 What is the return type of the method `round(float)` from the `Math` class?

Select the one correct answer.

(a) `int`
(b) `float`
(c) `double`
(d) `Integer`
(e) `Float`

10.13 What is the return type of the method `ceil(double)` from the `Math` class?

Select the one correct answer.

(a) `int`
(b) `float`
(c) `double`
(d) `Integer`
(e) `Double`

10.14 What will the following program print when run?

```
public class Round {
    public static void main(String[] args) {
        System.out.println(Math.round(-0.5) + " " + Math.round(0.5));
    }
};
```

Select the one correct answer.

(a) `0 0`
(b) `0 1`
(c) `-1 0`
(d) `-1 1`
(e) None of the above.

10.15 Which statements are true about the expression `((int)(Math.random()*4))`?

Select the three correct answers.

(a) It may evaluate to a negative number.
(b) It may evaluate to the number 0.
(c) The probability of it evaluating to the number 1 or the number 2 is the same.
(d) It may evaluate to the number 3.
(e) It may evaluate to the number 4.

10.5 The `String` Class

Handling character strings is supported through two `final` classes: `String` and `StringBuffer`. The `String` class implements immutable character strings, which are read-only once the string has been created and initialized, whereas the `StringBuffer` class implements dynamic character strings.

Character strings implemented using these classes are genuine objects, and the characters in such a string are represented using 16-bit characters (see Section 2.1, p. 23).

This section discusses the class `String` that provides facilities for creating, initializing, and manipulating character strings. The next section discusses the `StringBuffer` class.

Creating and Initializing Strings

String Literals Revisited

The easiest way of creating a `String` object is using a string literal:

```
String str1 = "You cannot change me!";
```

A string literal is a reference to a `String` object. The value in the `String` object is the character sequence enclosed in the double quotes of the string literal. Since a string literal is a reference, it can be manipulated like any other `String` reference. The reference value of a string literal can be assigned to another `String` reference: the reference `str1` will denote the `String` object with the value `"You cannot change me!"` after the assignment above. A string literal can be used to invoke methods on its `String` object:

```
int len = "You cannot change me!".length(); // 21
```

The compiler optimizes handling of string literals (and compile-time constant expressions that evaluate to strings): only one `String` object is shared by all string-valued constant expressions with the same character sequence. Such strings are said to be *interned*, meaning that they share a unique `String` object if they have the same content. The `String` class maintains a private pool where such strings are interned.

```
String str2 = "You cannot change me!";
```

Both String references str1 and str2 denote the same String object, initialized with the character string: "You cannot change me!". So does the reference str3 in the following code. The compile-time evaluation of the constant expression involving the two string literals, results in a string that is already interned:

```
String str3 = "You cannot" + " change me!"; // Compile-time constant expression
```

In the following code, both the references can1 and can2 denote the same String object that contains the string "7Up":

```
String can1 = 7 + "Up";  // Value of compile-time constant expression: "7Up"
String can2 = "7Up";     // "7Up"
```

However, in the code below, the reference can4 will denote a *new* String object that will have the value "7Up" at runtime:

```
String word = "Up";
String can4 = 7 + word;  // Not a compile-time constant expression.
```

The sharing of String objects between string-valued constant expressions poses no problem, since the String objects are immutable. Any operation performed on one String reference will never have any effect on the usage of other references denoting the same object. The String class is also declared final, so that no subclass can override this behavior.

String Constructors

The String class has numerous constructors to create and initialize String objects based on various types of arguments. The following shows two of them:

> `String(String s)`
>
> This constructor creates a new String object, whose contents are the same as those of the String object passed as argument.
>
> `String()`
>
> This constructor creates a new String object, whose content is the empty string, "".

Note that using a constructor creates a brand new String object, that is, using a constructor does not intern the string. A reference to an interned string can be obtained by calling the intern() method in the String class—in practice, there is usually no reason to do so.

In the following code, the String object denoted by str4 is different from the String object passed as argument:

```
String str4 = new String("You cannot change me!");
```

Constructing String objects can also be done from arrays of bytes, arrays of characters, or string buffers:

```
byte[] bytes = {97, 98, 98, 97};
char[] characters = {'a', 'b', 'b', 'a'};
```

```
StringBuffer strBuf = new StringBuffer("abba");
//...
String byteStr = new String(bytes);      // Using array of bytes: "abba"
String charStr = new String(character);  // Using array of chars: "abba"
String buffStr = new String(strBuf);     // Using string buffer: "abba"
```

In Example 10.3, note that the reference `str1` does not denote the same `String` object as references `str4` and `str5`. Using the `new` operator with a `String` constructor always creates a new `String` object. The expression `"You cannot" + words` is not a constant expression and, therefore, results in a new `String` object. The local references `str2` and `str3` in the `main()` method and the static reference `str1` in the `Auxiliary` class all denote the same interned string. Object value equality is hardly surprising between these references. It might be tempting to use the operator == for object value equality of string literals, but this is not advisable.

Example 10.3 *String Construction and Equality*

```
public class StringConstruction {

    static String str1 = "You cannot change me!";            // Interned

    public static void main(String[] args) {
        String emptyStr = new String();                      // ""
        System.out.println("emptyStr: " + emptyStr);

        String str2 = "You cannot change me!";               // Interned
        String str3 = "You cannot" + " change me!";          // Interned
        String str4 = new String("You cannot change me!");   // New String object

        String words = " change me!";
        String str5 = "You cannot" + words;                  // New String object

        System.out.println("str1 == str2:                " +
                           (str1 == str2));                  // (1) true
        System.out.println("str1.equals(str2):           " +
                           str1.equals(str2));               // (2) true

        System.out.println("str1 == str3:                " +
                           (str1 == str3));                  // (3) true
        System.out.println("str1.equals(str3):           " +
                           str1.equals(str3));               // (4) true

        System.out.println("str1 == str4:                " +
                           (str1 == str4));                  // (5) false
        System.out.println("str1.equals(str4):           " +
                           str1.equals(str4));               // (6) true

        System.out.println("str1 == str5:                " +
                           (str1 == str5));                  // (7) false
        System.out.println("str1.equals(str5):           " +
                           str1.equals(str5));               // (8) true
```

```
        System.out.println("str1 == Auxiliary.str1:      " +
                           (str1 == Auxiliary.str1));        // (9) true
        System.out.println("str1.equals(Auxiliary.str1): " +
                           str1.equals(Auxiliary.str1));     // (10) true

        System.out.println("\"You cannot change me!\".length(): " +
                           "You cannot change me!".length());// (11) 21
    }
}

class Auxiliary {
    static String str1 = "You cannot change me!";            // Interned
}
```

Output from the program:

```
emptyStr:
str1 == str2:                true
str1.equals(str2):           true
str1 == str3:                true
str1.equals(str3):           true
str1 == str4:                false
str1.equals(str4):           true
str1 == str5:                false
str1.equals(str5):           true
str1 == Auxiliary.str1:      true
str1.equals(Auxiliary.str1): true
"You cannot change me!".length(): 21
```

Reading Characters from a String

`char charAt(int index)`

A character at a particular index in a string can be read using the `charAt()` method. The first character is at index 0 and the last one at index one less than the number of characters in the string. If the index value is not valid, a `StringIndexOutOfBoundsException` is thrown.

`void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`

This method copies characters from the current string into the destination character array. Characters from the current string are read from index `srcBegin` to the index `srcEnd-1`, inclusive. They are copied into the destination array, starting at index `dstBegin` and ending at index `dstbegin+(srcEnd-srcBegin)-1`. The number of characters copied is `(srcEnd-srcBegin)`. An `IndexOutOfBoundsException` is thrown if the indices do not meet the criteria for the operation.

`int length()`

This method returns the number of characters in a string.

Example 10.4 uses these methods at (3), (4), (5), and (6). The program prints the frequency of a character in a string and illustrates copying from a string into a character array.

Example 10.4 *Reading Characters from a String*

```
public class ReadingCharsFromString {
    public static void main(String[] args) {
        int[] frequencyData = new int [Character.MAX_VALUE];// (1)
        String str = "You cannot change me!";               // (2)

        // Count the frequency of each character in the string.
        for (int i = 0; i < str.length(); i++)              // (3)
            try {
                frequencyData[str.charAt(i)]++;             // (4)
            } catch(StringIndexOutOfBoundsException e) {
                System.out.println("Index error detected: "+ i +" not in range.");
            }
        // Print the character frequency.
        System.out.println("Character frequency for string: \"" + str + "\"");
        for (int i = 0; i < frequencyData.length; i++)
            if (frequencyData[i] != 0)
                System.out.println((char)i + " (code "+ i +"): " +
                                   frequencyData[i]);

        System.out.println("Copying into a char array:");
        char[] destination = new char [str.length()];
        str.getChars( 0,            7, destination, 0);     // (5) "You can"
        str.getChars(10, str.length(), destination, 7);     // (6) " change me!"
        // Print the character array.
        for (int i = 0; i < 7 + (str.length() - 10); i++)
            System.out.print(destination[i]);
        System.out.println();
    }
}
```

Output from the program:

```
Character Frequency for string: "You cannot change me!"
  (code 32): 3
! (code 33): 1
Y (code 89): 1
a (code 97): 2
c (code 99): 2
e (code 101): 2
g (code 103): 1
h (code 104): 1
m (code 109): 1
n (code 110): 3
o (code 111): 2
t (code 116): 1
u (code 117): 1
Copying into a char array:
You can change me!
```

In Example 10.4, the `frequencyData` array at (1) stores the frequency of each character that can occur in a string. The string in question is declared at (2). Since a `char` value is promoted to an `int` value in arithmetic expressions, it can be used as an index in an array. Each element in the `frequencyData` array functions as a frequency counter for the character corresponding to the index value of the element:

```
frequencyData[str.charAt(i)]++;                    // (4)
```

The calls to the `getChars()` method at (5) and (6) copy particular substrings from the string into designated places in the `destination` array, before printing the whole character array.

Comparing Strings

Characters are compared based on their integer values.

```
boolean test = 'a' < 'b';    // true since 0x61 < 0x62
```

Two strings are compared *lexicographically*, as in a dictionary or telephone directory, by successively comparing their corresponding characters at each position in the two strings, starting with the characters in the first position. The string `"abba"` is less than `"aha"`, since the second character `'b'` in the string `"abba"` is less than the second character `'h'` in the string `"aha"`. The characters in the first position in each of these strings are equal.

The following public methods can be used for comparing strings:

```
boolean equals(Object obj)
boolean equalsIgnoreCase(String str2)
```

The `String` class overrides the `equals()` method from the `Object` class. The `String` class `equals()` method implements `String` object value equality as two `String` objects having the same sequence of characters. The `equalsIgnoreCase()` method does the same, but ignores the case of the characters.

```
int compareTo(String str2)
int compareTo(Object obj)
```

The first `compareTo()` method compares the two strings and returns a value based on the outcome of the comparison:

- the value `0`, if this string is equal to the string argument
- a value less than `0`, if this string is lexicographically less than the string argument
- a value greater than `0`, if this string is lexicographically greater than the string argument

The second `compareTo()` method (required by the `Comparable` interface) behaves like the first method if the argument `obj` is actually a `String` object; otherwise, it throws a `ClassCastException`.

Here are some examples of string comparisons:

```
String strA = new String("The Case was thrown out of Court");
String strB = new String("the case was thrown out of court");

boolean b1 = strA.equals(strB);             // false
boolean b2 = strA.equalsIgnoreCase(strB);   // true

String str1 = new String("abba");
String str2 = new String("aha");

int compVal1 = str1.compareTo(str2);         // negative value => str1 < str2
```

Character Case in a String

```
String toUpperCase()
String toUpperCase(Locale locale)
String toLowerCase()
String toLowerCase(Locale locale)
```

Note that the original string is returned if none of the characters need their case changed, but a new `String` object is returned if any of the characters need their case changed. These methods delegate the character-by-character case conversion to corresponding methods from the `Character` class.

These methods use the rules of the (default) *locale* (returned by the method `Locale.getDefault()`), which embodies the *idiosyncrasies* of a specific geographical, political, or cultural region regarding number/date/currency formats, character classification, alphabet (including case idiosyncrasies), and other localizations.

Example of case in strings:

```
String strA = new String("The Case was thrown out of Court");
String strB = new String("the case was thrown out of court");

String strC = strA.toLowerCase();  // Case conversion => New String object:
                                   // "the case was thrown out of court"
String strD = strB.toLowerCase();  // No case conversion => Same String object
String strE = strA.toUppperCase(); // Case conversion => New String object:
                                   // "THE CASE WAS THROWN OUT OF COURT"

boolean test1 = strC == strA; // false
boolean test2 = strD == strB; // true
boolean test3 = strE == strA; // false
```

Concatenation of Strings

Concatenation of two strings results in a string that consists of the characters of the first string followed by the characters of the second string. The overloaded operator + for string concatenation is discussed in Section 3.6 on page 62. In addition, the following method can be used to concatenate two strings:

```
String concat(String str)
```

The concat() method does not modify the String object on which it is invoked, as String objects are immutable. Instead the concat() method returns a reference to a brand new String object:

```
String billboard = "Just";
billboard.concat(" lost in space.");    // (1) Returned reference value not stored.
System.out.println(billboard);          // (2) "Just"
billboard = billboard.concat(" grooving").concat(" in heap.");  // (3) Chaining.
System.out.println(billboard);          // (4) "Just grooving in heap."
```

At (1), the reference value of the String object returned by the method concat() is not stored. This String object becomes inaccessible after (1). We see that the reference billboard still denotes the string literal "Just" at (2).

At (3), two method calls to the concat() method are *chained*. The first call returns a reference value to a new String object whose content is "Just grooving". The second method call is invoked on this String object using the reference value that was returned in the first method call. The second call results in yet another String object whose content is "Just grooving in heap." The reference value of this String object is assigned to the reference billboard. Because String objects are immutable, the creation of the temporary String object with the content "Just grooving" is inevitable at (3).

The compiler uses a string buffer to avoid this overhead of temporary String objects when applying the string concatenation operator (p. 424).

A simple way to convert any primitive value to its string representation is by concatenating it with the empty string (""), using the string concatenation operator (+) (see also (6c) in Figure 10.2):

```
String strRepresentation = "" + 2003;  // "2003"
```

Some more examples of string concatenation follow:

```
String motto = new String("Program once");     // (1)
motto += ", execute everywhere.";              // (2)
motto = motto.concat(" Don't bet on it!");     // (3)
```

Note that a new String object is assigned to the reference motto each time in the assignment at (1), (2), and (3). The String object with the contents "Program once" becomes inaccessible after the assignment at (2). The String object with the contents "Program once, execute everywhere." becomes inaccessible after (3). The reference motto denotes the String object with the following contents after execution of the assignment at (3):

```
"Program once, execute everywhere. Don't bet on it!"
```

Searching for Characters and Substrings

The following overloaded methods can be used to find the index of a character, or the start index of a substring in a string. These methods search *forward* toward the end of the string. In other words, the index of the *first* occurrence of the character or substring is found. If the search is unsuccessful, the value –1 is returned.

`int indexOf(int ch)`

Finds the index of the first occurrence of the argument character in a string.

`int indexOf(int ch, int fromIndex)`

Finds the index of the first occurrence of the argument character in a string, starting at the index specified in the second argument. If the index argument is negative, the index is assumed to be 0. If the index argument is greater than the length of the string, it is effectively considered to be equal to the length of the string—returning the value -1.

`int indexOf(String str)`

Finds the start index of the first occurrence of the substring argument in a string.

`int indexOf(String str, int fromIndex)`

Finds the start index of the first occurrence of the substring argument in a string, starting at the index specified in the second argument.

The `String` class also defines a set of methods that search for a character or a substring, but the search is *backward* toward the start of the string. In other words, the index of the *last* occurrence of the character or substring is found.

```
int lastIndexOf(int ch)
int lastIndexOf(int ch, int fromIndex)
int lastIndexOf(String str)
int lastIndexOf(String str, int fromIndex)
```

The following method can be used to create a string in which all occurrences of a character in a string have been replaced with another character:

```
String replace(char oldChar, char newChar)
```

Examples of search methods:

```
String funStr = "Java Jives";
//               0123456789

String newStr = funStr.replace('J', 'W');     // "Wava Wives"

int jInd1a = funStr.indexOf('J');             // 0
int jInd1b = funStr.indexOf('J', 1);          // 5
int jInd2a = funStr.lastIndexOf('J');         // 5
int jInd2b = funStr.lastIndexOf('J', 4);      // 0

String banner = "One man, One vote";
//               01234567890123456

int subInd1a = banner.indexOf("One");         // 0
int subInd1b = banner.indexOf("One", 3);      // 9
int subInd2a = banner.lastIndexOf("One");     // 9
int subInd2b = banner.lastIndexOf("One", 10); // 9
int subInd2c = banner.lastIndexOf("One", 8);  // 0
int subInd2d = banner.lastIndexOf("One", 2);  // 0
```

Extracting Substrings

```
String trim()
```

This method can be used to create a string where white space (in fact all characters with values less than or equal to the space character `'\u0020'`) from the front (leading) and the end (trailing) of a string has been removed.

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

The `String` class provides these overloaded methods to extract substrings from a string. A new `String` object containing the substring is created and returned. The first method extracts the string that starts at the given index `startIndex` and extends to the end of the string. The end of the substring can be specified by using a second argument `endIndex` that is the index of the first character *after* the substring, that is, the last character in the substring is at index `endIndex-1`. If the index value is not valid, a `StringIndexOutOfBoundsException` is thrown.

Examples of extracting substrings:

```
String utopia = "\t\n  Java Nation \n\t  ";
utopia = utopia.trim();                          // "Java Nation"
utopia = utopia.substring(5);                    // "Nation"
String radioactive = utopia.substring(3,6);      // "ion"
```

Converting Primitive Values and Objects to Strings

The `String` class overrides the `toString()` method in the `Object` class and returns the `String` object itself:

```
String toString()
```

The `String` class also defines a set of static overloaded `valueOf()` methods to convert objects and primitive values into strings.

```
static String valueOf(Object obj)
static String valueOf(char[] character)
static String valueOf(boolean b)
static String valueOf(char c)
```

All these methods return a string representing the given parameter value. A call to the method with the parameter `obj` is equivalent to `obj.toString()`. The `boolean` values `true` and `false` are converted into the strings `"true"` and `"false"`. The `char` parameter is converted to a string consisting of a single character.

```
static String valueOf(int i)
static String valueOf(long l)
static String valueOf(float f)
static String valueOf(double d)
```

The static `valueOf()` method that accepts a primitive value as argument is equivalent to the static `toString()` method in the corresponding wrapper class for each of the primitive data types (see also (6a) and (6b) in Figure 10.2 on p. 393).

Note that there are no `valueOf()` methods that accept a `byte` or a `short`.

Examples of string conversions:

```
String anonStr   = String.valueOf("Make me a string.");      // "Make me a string."
String charStr   = String.valueOf(new char[] {'a', 'h', 'a'});// "aha"
String boolTrue  = String.valueOf(true);                       // "true"
String doubleStr = String.valueOf(Math.PI);                  // "3.141592653589793"
```

Other miscellaneous methods exist for reading the string characters into an array of characters (`toCharArray()`), converting the string into an array of bytes (`getBytes()`), and searching for prefixes (`startsWith()`) and suffixes (`endsWith()`) of the string. The method `hashCode()` can be used to compute a hash value based on the characters in the string.

Review Questions

10.16 Which of the following operators cannot be used in conjunction with a `String` object?

Select the two correct answers.

(a) `+`
(b) `-`
(c) `+=`
(d) `.`
(e) `&`

10.17 Which expression will extract the substring `"kap"` from a string defined by `String str = "kakapo"`?

Select the one correct answer.

(a) `str.substring(2, 2)`
(b) `str.substring(2, 3)`
(c) `str.substring(2, 4)`
(d) `str.substring(2, 5)`
(e) `str.substring(3, 3)`

10.18 What will be the result of attempting to compile and run the following code?

```
class MyClass {
    public static void main(String[] args) {
        String str1 = "str1";
        String str2 = "str2";
        String str3 = "str3";

        str1.concat(str2);
        System.out.println(str3.concat(str1));
    }
}
```

Select the one correct answer.

(a) The code will fail to compile since the expression `str3.concat(str1)` will not result in a valid argument for the `println()` method.
(b) The program will print `str3str1str2` when run.
(c) The program will print `str3` when run.
(d) The program will print `str3str1` when run.
(e) The program will print `str3str2` when run.

10.19 What function does the `trim()` method of the `String` class perform?

Select the one correct answer.

(a) It returns a string where the leading white space of the original string has been removed.
(b) It returns a string where the trailing white space of the original string has been removed.
(c) It returns a string where both the leading and trailing white space of the original string has been removed.
(d) It returns a string where all the white space of the original string has been removed.
(e) None of the above.

10.20 Which statements are true?

Select the two correct answers.

(a) `String` objects are immutable.
(b) Subclasses of the `String` class can be mutable.
(c) All wrapper classes are declared `final`.
(d) All objects have a `public` method named `clone()`.
(e) The expression `((new StringBuffer()) instanceof String)` is always `true`.

10.21 Which of these expressions are legal?

Select the four correct answers.

(a) `"co".concat("ol")`
(b) `("co" + "ol")`

(c) `('c' + 'o' + 'o' + 'l')`
(d) `("co" + new String('o' + 'l'))`
(e) `("co" + new String("co"))`

10.22 What will be the result of attempting to compile and run the following code?

```
public class RefEq {
    public static void main(String[] args) {
        String s = "ab" + "12";
        String t = "ab" + 12;
        String u = new String("ab12");
        System.out.println((s==t) + " " + (s==u));
    }
}
```

Select the one correct answer.

(a) The code will fail to compile.
(b) The program will print `false false` when run.
(c) The program will print `false true` when run.
(d) The program will print `true false` when run.
(e) The program will print `true true` when run.

10.23 Which of these parameter lists have a corresponding constructor in the `String` class?

Select the three correct answers.

(a) `()`
(b) `(int capacity)`
(c) `(char[] data)`
(d) `(String str)`

10.24 Which method is not defined in the `String` class?

Select the one correct answer.

(a) `trim()`
(b) `length()`
(c) `concat(String)`
(d) `hashCode()`
(e) `reverse()`

10.25 Which statement concerning the `charAt()` method of the `String` class is true?

Select the one correct answer.

(a) The `charAt()` method takes a `char` value as an argument.
(b) The `charAt()` method returns a `Character` object.
(c) The expression `("abcdef").charAt(3)` is illegal.
(d) The expression `"abcdef".charAt(3)` evaluates to the character `'d'`.
(e) The index of the first character is 1.

10.26 Which expression will evaluate to `true`?

Select the one correct answer.

(a) `"hello: there!".equals("hello there")`
(b) `"HELLO THERE".equals("hello there")`
(c) `("hello".concat("there")).equals("hello there")`
(d) `"Hello There".compareTo("hello there") == 0`
(e) `"Hello there".toLowerCase().equals("hello there")`

10.27 What will the following program print when run?

```
public class Search {
    public static void main(String[] args) {
        String s = "Contentment!";
        int middle = s.length()/2;
        String nt = s.substring(middle-1, middle+1);
        System.out.println(s.lastIndexOf(nt, middle));
    }
};
```

Select the one correct answer.

(a) `2`
(b) `4`
(c) `5`
(d) `7`
(e) `9`
(f) `11`

10.6 The `StringBuffer` Class

In contrast to the `String` class, which implements immutable character strings, the `StringBuffer` class implements mutable character strings. Not only can the character string in a string buffer be changed, but the capacity of the string buffer can also change dynamically. The *capacity* of a string buffer is the maximum number of characters that a string buffer can accommodate before its size is automatically augmented.

Although there is a close relationship between objects of the `String` and `StringBuffer` classes, these are two independent `final` classes, both directly extending the `Object` class. Hence, `String` references cannot be stored (or cast) to `StringBuffer` references and vice versa. Both `String` and `StringBuffer` are thread-safe. String buffers are preferred when heavy modification of character strings is involved.

The `StringBuffer` class provides various facilities for manipulating string buffers:

- constructing string buffers
- changing, deleting, and reading characters in string buffers

- constructing strings from string buffers
- appending, inserting, and deleting in string buffers
- controlling string buffer capacity

Constructing String Buffers

The `final` class `StringBuffer` provides three constructors that create and initialize `StringBuffer` objects and set their initial capacity.

`StringBuffer(String s)`

The contents of the new `StringBuffer` object are the same as the contents of the `String` object passed as argument. The initial capacity of the string buffer is set to the length of the argument string, plus room for 16 more characters.

`StringBuffer(int length)`

The new `StringBuffer` object has no content. The initial capacity of the string buffer is set to the value of the argument `length`, which cannot be less than 0.

`StringBuffer()`

This constructor also creates a new `StringBuffer` object with no content. The initial capacity of the string buffer is set for 16 characters.

Examples of `StringBuffer` object creation and initialization:

```
StringBuffer strBuf1 = new StringBuffer("Phew!");      // "Phew!", capacity 21
StringBuffer strBuf2 = new StringBuffer(10);          // "", capacity 10
StringBuffer strBuf3 = new StringBuffer();            // "", capacity 16
```

Reading and Changing Characters in String Buffers

`int length()`

Returns the number of characters in the string buffer.

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

These methods read and change the character at a specified index in the string buffer, respectively. The first character is at index 0 and the last one at index one less than the number of characters in the string buffer. A `StringIndexOutOf-BoundsException` is thrown if the index is not valid.

The following is an example of reading and changing string buffer contents:

```
StringBuffer strBuf = new StringBuffer("Javv");          // "Javv", capacity 20
strBuf.setCharAt(strBuf.length()-1, strBuf.charAt(1));   // "Java"
```

Constructing Strings from String Buffers

The `StringBuffer` class overrides the `toString()` method from the `Object` class. It returns the contents of a string buffer in a `String` object.

```
String fromBuf = strBuf.toString();                    // "Java"
```

Since the `StringBuffer` class does not override the `equals()` method from the `Object` class, contents of string buffers should be converted to `String` objects for string comparison.

Appending, Inserting, and Deleting Characters in String Buffers

Appending, inserting, and deleting characters automatically results in adjustment of the string buffer's capacity, if necessary. The indices passed as arguments in the methods must be equal to or greater than 0. A `StringIndexOutOfBoundsException` is thrown if an index is not valid.

Appending Characters to a String Buffer

The overloaded method `append()` can be used to append characters at the end of a string buffer.

```
StringBuffer append(Object obj)
```

The `obj` argument is converted to a string as if by the static method call `String.valueOf(obj)`, and this string is appended to the current string buffer.

```
StringBuffer append(String str)
StringBuffer append(char[] str)
StringBuffer append(char[] str, int offset, int len)
StringBuffer append(char c)
```

These methods allow characters from various sources to be appended at the end of the current string buffer.

```
StringBuffer append(boolean b)
StringBuffer append(int i)
StringBuffer append(long l)
StringBuffer append(float f)
StringBuffer append(double d)
```

These methods convert the primitive value of the argument to a string by applying the static method `String.valueOf()` to the argument, before appending the result to the string buffer:

Inserting Characters in a String Buffer

The overloaded method `insert()` can be used to insert characters at a given position in a string buffer.

```
StringBuffer insert(int offset, Object obj)
StringBuffer insert(int offset, String str)
StringBuffer insert(int offset, char[] str)
StringBuffer insert(int offset, char c)
StringBuffer insert(int offset, boolean b)
StringBuffer insert(int offset, int i)
StringBuffer insert(int offset, long l)
StringBuffer insert(int offset, float f)
StringBuffer insert(int offset, double d)
```

The argument is converted, if necessary, by applying the static method `String.valueOf()`. The `offset` argument specifies where the characters are to be inserted and must be greater than or equal to 0.

Deleting Characters in a String Buffer

The following methods can be used to delete characters from specific positions in a string buffer:

```
StringBuffer deleteCharAt(int index)
StringBuffer delete(int start, int end)
```

The first method deletes a character at a specified index in the string buffer, contracting the string buffer by one character. The second method deletes a substring, which is specified by the `start` index (inclusive) and the `end` index (exclusive).

Among other miscellaneous methods included in the class `StringBuffer` is the following method, which reverses the contents of a string buffer:

```
StringBuffer reverse()
```

Examples of appending, inserting, and deleting in string buffers:

```
StringBuffer buffer = new StringBuffer("banana split");   // "banana split"
buffer.delete(4,12);                                      // "bana"
buffer.append(42);                                        // "bana42"
buffer.insert(4,"na");                                    // "banana42"
buffer.reverse();                                         // "24ananab"
buffer.deleteCharAt(buffer.length()-1);                   // "24anana"
buffer.append('s');                                       // "24ananas"
```

All the previous methods modify the contents of the string buffer and also return a reference value denoting the string buffer. This allows *chaining* of method calls. The method calls invoked on the string buffer denoted by the reference `buffer` can be chained as follows, giving the same result:

```
buffer.delete(4,12).append(42).insert(4,"na").reverse().
       deleteCharAt(buffer.length()-1).append('s');       // "24ananas"
```

The method calls in the chain are evaluated from left to right, so that the previous chain of calls is interpreted as follows:

```
(((((buffer.delete(4,12)).append(42)).insert(4,"na")).reverse()).
     deleteCharAt(buffer.length()-1)).append('s');        // "24ananas"
```

Each method call returns the reference value of the modified string buffer. This value is used to invoke the next method. The string buffer remains denoted by the reference `buffer`.

The compiler uses string buffers to implement the string concatenation, +. The following example code of string concatenation

```
String str1 = 4 + "U" + "Only";                                    // (1) "4UOnly"
```

is equivalent to the following code using one string buffer:

```
String str2 = new StringBuffer().
                  append(4).append("U").append("Only").toString(); // (2)
```

The code at (2) does not create any temporary `String` objects when concatenating several strings, since a single `StringBuffer` object is modified and finally converted to a `String` object.

Controlling String Buffer Capacity

`int capacity()`

Returns the current capacity of the string buffer, that is, the number of characters the current buffer can accommodate without allocating a new, larger array to hold characters.

`void ensureCapacity(int minCapacity)`

Ensures that there is room for at least `minCapacity` number of characters. It expands the string buffer, depending on the current capacity of the buffer.

`void setLength(int newLength)`

This method ensures that the actual number of characters, that is, length of the string buffer, is exactly equal to the value of the `newLength` argument, which must be greater than or equal to 0. This operation can result in the string being truncated or padded with null characters (`'\u0000'`).

This method only affects the capacity of the string buffer if the value of the parameter `newLength` is greater than current capacity.

One use of this method is to clear the string buffer:

```
buffer.setLength(0);      // Empty the buffer.
```

Review Questions

10.28 What will be the result of attempting to compile and run the following program?

```
public class MyClass {
    public static void main(String[] args) {
        String s = "hello";
        StringBuffer sb = new StringBuffer(s);
        sb.reverse();
```

```
            if (s == sb) System.out.println("a");
            if (s.equals(sb)) System.out.println("b");
            if (sb.equals(s)) System.out.println("c");
        }
    }
```

Select the one correct answer.

(a) The code will fail to compile since the constructor of the `String` class is not called properly.
(b) The code will fail to compile since the expression `(s == sb)` is illegal.
(c) The code will fail to compile since the expression `(s.equals(sb))` is illegal.
(d) The program will print `c` when run.
(e) The program will throw a `ClassCastException` when run.

10.29 What will be the result of attempting to compile and run the following program?

```
public class MyClass {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("have a nice day");
        sb.setLength(6);
        System.out.println(sb);
    }
}
```

Select the one correct answer.

(a) The code will fail to compile since there is no method named `setLength` in the `StringBuffer` class.
(b) The code will fail to compile since the `StringBuffer` reference `sb` is not a legal argument to the `println()` method.
(c) The program will throw a `StringIndexOutOfBoundsException` when run.
(d) The program will print `have a nice day` when run.
(e) The program will print `have a` when run.
(f) The program will print `ce day` when run.

10.30 Which of these parameter lists have a corresponding constructor in the `StringBuffer` class?

Select the three correct answers.

(a) `()`
(b) `(int capacity)`
(c) `(char[] data)`
(d) `(String str)`

10.31 Which method is not defined in the `StringBuffer` class?

Select the one correct answer.

(a) `trim()`
(b) `length()`
(c) `append(String)`
(d) `reverse()`
(e) `setLength(int)`

10.32 What will be the result of attempting to compile and run the following code?

```
public class StringMethods {
    public static void main(String[] args) {
        String str = new String("eenny");
        str.concat(" meeny");
        StringBuffer strBuf = new StringBuffer(" miny");
        strBuf.append(" mo");
        System.out.println(str + strBuf);
    }
}
```

Select the one correct answer.

(a) The code will fail to compile.
(b) The program will print `eenny meeny miny mo` when run.
(c) The program will print `meeny miny mo` when run.
(d) The program will print `eenny miny mo` when run.
(e) The program will print `eenny meeny miny` when run.

Chapter Summary

The following information was included in this chapter:

- discussion of the `Object` class, which is the most fundamental class in Java
- discussion of the wrapper classes, which not only allow primitive values to be treated as objects, but also contain useful methods for converting values
- discussion of the `Math` class, which provides an assortment of mathematical functions
- discussion of the `String` class, showing how immutable strings are created and used
- discussion of the `StringBuffer` class, showing how dynamic string buffers are created and manipulated

Programming Exercises

10.1 Create a class named `Pair`, which aggregates two arbitrary objects. Implement the `equals()` and `hashCode()` methods in such a way that a `Pair` object is identical to another `Pair` object if, and only if, the pair of constituent objects are identical. Make the `toString()` implementation return the textual representation of both the constituent objects in a `Pair` object. Objects of the `Pair` class should be immutable.

10.2 A palindrome is a text phrase that spells the same thing backward and forward. The word *redivider* is a palindrome, since the word would spell the same even if the character sequence were reversed. Write a program that takes a word as an argument and reports whether the word is a palindrome.