Selected API Classes 8

Chapter Topics

- An overview of the `java.lang` package
- Understanding the importance of the `Object` class and its API
- Handling primitive values as objects with the wrapper classes
- Converting between primitive values and wrapper classes that represent them, and their text representation
- Creating and manipulating immutable strings with the `String` class, including text blocks
- Creating and manipulating mutable strings with the `StringBuilder` class
- Comparing the `String` and `StringBuilder` classes
- Using common mathematical functions provided by the `Math` class
- Generating pseudorandom numbers using the `Math` and `Random` classes
- Using big numbers for arbitrary-precision arithmetic operations

Java SE 17 Developer Exam Objectives	
[1.1] Use primitives and wrapper classes including Math API, parentheses, type promotion, and casting to evaluate arithmetic and boolean expressions ❍ *Only wrapper classes and the Math API are covered in this chapter.* ❍ *For primitive data types, operators, conversions, and evaluation of expressions, see* Chapter 2, p. 29.	§8.3, p. 429 §8.6, p. 478
[1.2] Manipulate text, including text blocks, using String and StringBuilder classes	§8.4, p. 439 §8.5, p. 464

Java SE 11 Developer Exam Objectives	
[1.1] Use primitives and wrapper classes, including, operators, the use of parentheses, type promotion and casting ❍ *Only wrapper classes are covered in this chapter.* ❍ *For primitive data types, operators, and conversions, see* Chapter 2, p. 29.	§8.3, p. 429
[1.2] Handle text using String and StringBuilder classes	§8.4, p. 439 §8.5, p. 464

The `Object` class, being the superclass of all classes, defines general functionality for all classes. The wrapper classes provide the support to treat primitive values as objects. The classes `String` and `StringBuilder` provide the support to create and manipulate strings. In addition, the `Math`, `Random`, and `BigDecimal` classes are introduced to leverage mathematical functions, pseudorandom number generators, and arbitrary-precision arithmetic operations, respectively. The APIs of these classes are the main subject of this chapter.

8.1 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 superclass of all classes, and the wrapper classes (`Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`) that are 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 handling exceptions (`Throwable`, `Error`, `Exception`, `Runtime-Exception`). The `java.lang` package also contains classes that provide the standard input, output, and error streams (`System`), string handling (`String`, `StringBuilder`), and mathematical functions (`Math`).

Figure 8.1 shows the important classes whose API is discussed in subsequent sections.

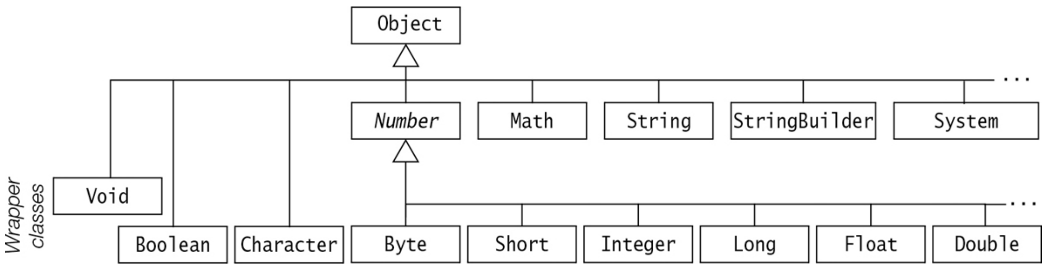


Figure 8.1 *Partial Inheritance Hierarchy in the* `java.lang` *Package*

8.2 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 (§5.1, p. 191). Thus the `Object` class is always at the root of any inheritance hierarchy. The `Object` class defines the basic functionality that all objects exhibit and all classes inherit. This relationship also applies to arrays, since these are genuine objects in Java.

The `Object` class defines common functionality for all objects through the following methods (see Example 8.1 for usage of these methods):

Click here to view code image

```
public boolean equals(Object obj)
```

The general contract of the `equals()` method is discussed in connection with object comparison (§14.2, p. 744). Object reference and value equality are discussed together with the `==` and `!=` operators (§2.13, p. 75). The `equals()` method in the `Object` class returns `true` only if the two references compared denote the same object—that is, they are aliases. The `equals()` method is normally overridden to provide the semantics of object value equality, as is the case for the wrapper classes (p. 432) and the `String` class (p. 439).

```
public int hashCode()
```

Returns an `int` value as the *hash value* of the object. The general contract of the `hashCode()` method is discussed in connection with object comparison (§14.3, p. 753). In *hash tables*, the hash value can be used to store and retrieve objects efficiently (§15.8, p. 830). The `hashCode()`

method is normally overridden by a class, as is the case for the wrapper classes (p. 432) and the `String` class (p. 439).

```
public String toString()
```

The general contract of the method is to return a text representation of the object. If a subclass does not override this method, it returns a text representation of the object, which has the following format:

```
"<class_name>@<hash_code>"
```

Since the hash value of an object is an `int` value, this value is printed as a hexadecimal number (e.g., `3e25a5`). This method is usually overridden by a class to provide appropriate text representation for its object. The method call `System.out.println(objRef)` will implicitly convert its argument to a text representation by calling the `toString()` method on the argument.

Click here to view code image

```
public final Class<?> getClass()
```

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

Click here to view code image

```
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 as `public` to allow any client to clone objects of the class. Not declaring the overriding method as `public` or `protected` violates the contract for overriding, as it narrows the accessibility of the method, and will be flagged as a compile-time error.

If the overriding `clone()` method in the subclass relies on the `clone()` method in the `Object` class (i.e., a shallow copy), 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 `CloneNotSupported-Exception`.

Click here to view code image

```
@Deprecated(since="9") protected void finalize() throws Throwable
```

This method is called on an object just before it is garbage collected so that any clean-up can be done. However, the `finalize()` method in the `Object` class has been deprecated since Java 9—it should no longer be used and it may be removed in the future. The method is deemed problematic, and better solutions exist for implementing the same functionality.

In addition, the following `final` methods in the `Object` class provide support for thread communication in synchronized code (§22.4, p. 1400).

Click here to view code image

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

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

Example 8.1 *Methods in the* `Object` *Class*

Click here to view code image

```
// File: ObjectMethods.java
class MyClass implements Cloneable {
  @Override
  public MyClass clone() {
    MyClass obj = null;
    try { obj = (MyClass) super.clone(); }  // Calls overridden method.
    catch (CloneNotSupportedException e) { System.out.println(e);}
    return obj;
  }
}
//______________________________________________________________________________
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");

    System.out.println("str1: " + str1.toString());
    System.out.println("str2: " + str2.toString() + "\n");

    // Method hashCode() overridden in String class.
    // Strings that 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<? extends String> rtStringClass  = str1.getClass();
    Class<? extends MyClass> 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 = array1.clone();
    // 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 = obj1.clone();
    System.out.println("hash code for cloned MyClass obj3: " + obj3.hashCode());
    System.out.println("obj1 == obj3: " + (obj1 == obj3));
  }
}
```

Probable output from the program:

Click here to view code image

```
str1: WhoAmI
str2: WhoAmI

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

hash code for MyClass obj1: 2036368507
hash code for MyClass obj2: 1785210046

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@7960847b

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

```
hash code for cloned MyClass obj3: 1552787810
obj1 == obj3: false
```

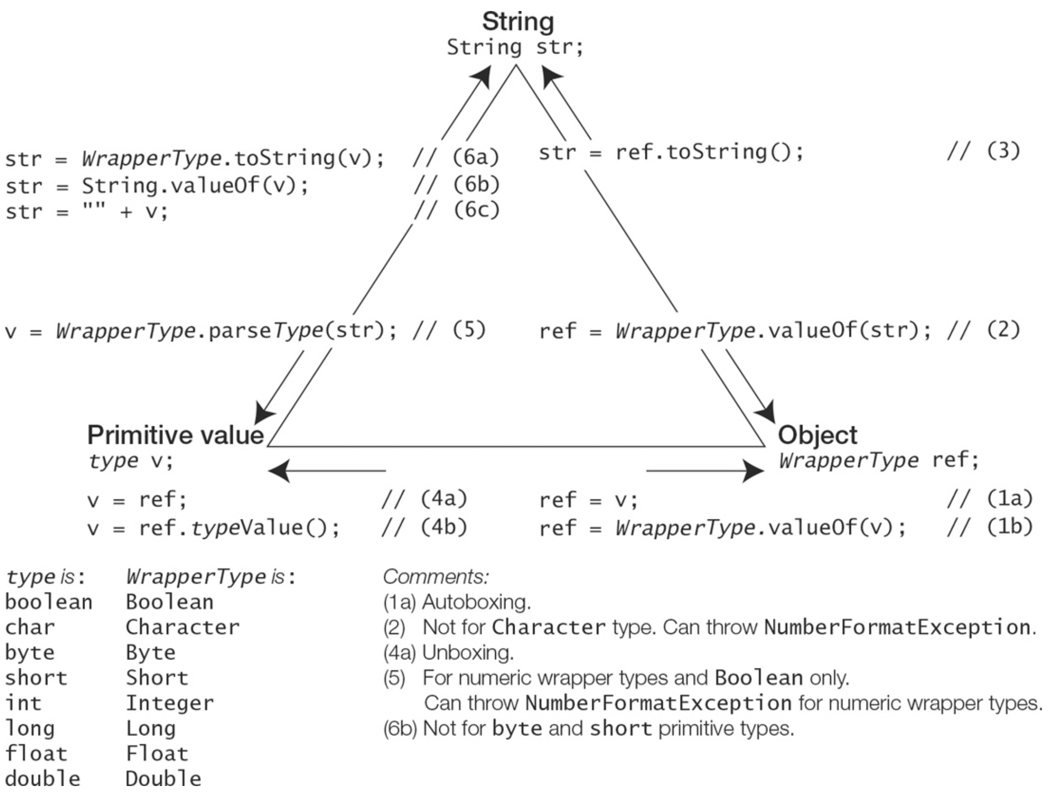
8.3 The Wrapper Classes

Wrapper classes were introduced with the discussion of the primitive data types (Table 2.16, p. 43), and also in connection with boxing and unboxing of primitive values (§2.3, p. 45). Primitive values in Java are not objects. To manipulate these values as objects, the `java.lang` package provides a *wrapper* class for each of the primitive data types (shown in the bottom left of Figure 8.2). The name of the wrapper class is the name of the primitive data type with an uppercase letter, except for `int` (`Integer`) and `char` (`Character`). All wrapper classes are `final`, meaning that they cannot be extended. The objects of all wrapper classes that can be instantiated are *immutable*; in other words, the value in the wrapper object 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 define useful constants, fields, and conversion methods.

A very important thing to note about the wrapper classes is that their constructors are now *deprecated*. So these constructors should not be used to create wrapper objects, as these constructors will be removed from the API in the future. As we shall see later in this section, the APIs include methods for this purpose that provide significantly improved space and time performance.



```
str = WrapperType.toString(v);  // (6a)      str = ref.toString();            // (3)
str = String.valueOf(v);        // (6b)
str = "" + v;                   // (6c)

v = WrapperType.parseType(str); // (5)       ref = WrapperType.valueOf(str); // (2)

v = ref;                        // (4a)      ref = v;                         // (1a)
v = ref.typeValue();            // (4b)      ref = WrapperType.valueOf(v);   // (1b)
```

type is:	*WrapperType* is:
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Comments:
(1a) Autoboxing.
(2) Not for `Character` type. Can throw `NumberFormatException`.
(4a) Unboxing.
(5) For numeric wrapper types and `Boolean` only.
Can throw `NumberFormatException` for numeric wrapper types.
(6b) Not for `byte` and `short` primitive types.

Figure 8.2 *Converting Values among Primitive, Wrapper, and* `String` *Types*

Common Wrapper Class Utility Methods

Converting Primitive Values to Wrapper Objects

Autoboxing is a convenient way to wrap a primitive value in an object ((1a) in Figure 8.2 and §2.3, p. 45).

Click here to view code image

```
Character charObj1   = '\n';
Boolean   boolObj1   = true;
Integer   intObj1    = 2020;
Double    doubleObj1 = 3.14;
```

We can also use the following `valueOf(`*type* `v)` method that takes the primitive value of *type* to convert as an argument ((1b) in **Figure 8.2**).

Click here to view code image

```
static WrapperType valueOf(type v)
```

```
Character charObj1   = Character.valueOf('\n');
Boolean   boolObj1   = Boolean.valueOf(true);
Integer   intObj1    = Integer.valueOf(2020);
Double    doubleObj1 = Double.valueOf(3.14);
```

Converting Strings to Wrapper Objects

Each wrapper class (except `Character`) defines the static method `valueOf(String str)` that returns the wrapper object corresponding to the primitive value represented by the `String` object passed as an argument ((2) in **Figure 8.2**). This method for the numeric wrapper types also throws a `NumberFormatException` if the `String` parameter is not a valid number.

Click here to view code image

```
static WrapperType valueOf(String str)
```

Click here to view code image

```
Boolean boolObj4   = Boolean.valueOf("false");
Integer intObj3    = Integer.valueOf("1949");
Double  doubleObj2 = Double.valueOf("3.14");
Double  doubleObj3 = Double.valueOf("Infinity");
```

In addition to the one-argument `valueOf()` method, the integer wrapper classes 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. Note that the string argument does not specify the prefix for the number system notation.

Click here to view code image

```
static IntegerWrapperType valueOf(String str, int base)
                          throws NumberFormatException
```

Click here to view code image

```
Byte    byteObj1  = Byte.valueOf("1010", 2);     // Decimal value 10
Short   shortObj2 = Short.valueOf("12", 8);      // Decimal value 10
Short   shortObj3 = Short.valueOf("012", 8);     // Decimal value 10
Short   shortObj4 = Short.valueOf("\012", 8);    // NumberFormatException
Integer intObj4   = Integer.valueOf("-a", 16);   // Decimal value -10
Integer intObj6   = Integer.valueOf("-0xa", 16); // NumberFormatException
Long    longObj2  = Long.valueOf("-a", 16);      // 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 text representation of the primitive value in the wrapper object ((3) in Figure 8.2).

```
String toString()
```

Click here to view code image

```
String charStr   = charObj1.toString();        // "\n"
String boolStr   = boolObj2.toString();        // "true"
String intStr    = intObj1.toString();         // "2020"
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*, which is passed as an argument ((6a) in Figure 8.2).

Click here to view code image

```
static String toString(type v)
```

Click here to view code image

```
String charStr2   = Character.toString('\n');  // "\n"
String boolStr2   = Boolean.toString(true);    // "true"
String intStr2    = Integer.toString(2020);    // "2020"
String doubleStr2 = Double.toString(3.14);     // "3.14"
```

For integer primitive types, the base is assumed to be 10. For floating-point numbers, the text 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 methods for converting integers to text representations in decimal, binary, octal, and hexadecimal notation (p. 435).

Converting Wrapper Objects to Primitive Values

Unboxing is a convenient way to unwrap the primitive value in a wrapper object ((4a) in Figure 8.2 and §2.3, p. 45).

Click here to view code image

```
char    c = charObj1;                          // '\n'
boolean b = boolObj2;                          // true
int     i = intObj1;                           // 2020
double  d = doubleObj1;                        // 3.14
```

Each wrapper class defines a `type Value()` method that returns the primitive value in the wrapper object ((4b) in Figure 8.2).

```
type typeValue()
```

Click here to view code image

```
char    c = charObj1.charValue();          // '\n'
boolean b = boolObj2.booleanValue();       // true
int     i = intObj1.intValue();            // 2020
double  d = doubleObj1.doubleValue();      // 3.14
```

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

Wrapper Comparison, Equality, and Hash Code

Each wrapper class implements the `Comparable<Type>` interface, which defines the following method:

Click here to view code image

```
int compareTo(Type obj2)
```

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

Click here to view code image

```
// Comparisons based on objects created earlier
Character charObj2   = 'a';
int result1 = charObj1.compareTo(charObj2);      // result1 < 0
int result2 = intObj1.compareTo(intObj3);        // result2 > 0
int result3 = doubleObj1.compareTo(doubleObj2);  // result3 == 0
int result4 = doubleObj1.compareTo(intObj1);     // Compile-time error!
```

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

Click here to view code image

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

Click here to view code image

```
// Comparisons based on objects created earlier
boolean charTest   = charObj1.equals(charObj2);      // false
boolean boolTest   = boolObj2.equals(Boolean.FALSE); // false
boolean intTest    = intObj1.equals(intObj3);        // true
boolean doubleTest = doubleObj1.equals(doubleObj2);  // true
boolean test       = intObj1.equals(Long.valueOf(2020L)); // false. Not same type.
```

The following values are *interned* when they are wrapped during boxing. That is, only *one* wrapper object exists in the program for these primitive values when boxing is applied:

- The `boolean` value `true` or `false`
- A `byte`
- A `char` with a Unicode value in the interval `[\u0000`, `\u007f]` (i.e., decimal interval [0, 127])
- An `int` or `short` value in the interval `[-128, 127]`

If references `w1` and `w2` refer to two wrapper objects that box the *same* value, which fulfills any of the conditions mentioned above, then `w1 == w2` is always `true`. In other words, for the values listed previously, object equality and reference equality give the same result.

Click here to view code image

```
// Reference and object equality
Byte bRef1 = 10;
Byte bRef2 = 10;
System.out.println(bRef1 == bRef2);            // true
System.out.println(bRef1.equals(bRef2));       // true

Integer iRef1 = 1000;
Integer iRef2 = 1000;
System.out.println(iRef1 == iRef2);            // false, values not in [-128, 127]
System.out.println(iRef1.equals(iRef2));       // true
```

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

Click here to view code image

```
int hashCode()
```

```
int index = charObj1.hashCode();                   // 10 ('\n')
```

Numeric Wrapper Classes

The numeric wrapper classes `Byte`, `Short`, `Integer`, `Long`, `Float`, and `Double` are all subclasses of the `abstract` class `Number` (**Figure 8.1**, **p. 425**).

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

Click here to view code image

```
NumericWrapperType.MIN_VALUE
NumericWrapperType.MAX_VALUE
```

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

Click here to view code image

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

Converting Numeric Wrapper Objects to Numeric Primitive Types

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:

Click here to view code image

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

See also (4b) in Figure 8.2.

The following code shows conversion of values in numeric wrapper objects to any numeric primitive type:

Click here to view code image

```
Byte    byteObj2   = (byte)16;                  // Cast mandatory
Integer intObj5    = 42030;
Double  doubleObj4 = 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), 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 str)`, which returns the primitive numeric value represented by the `String` object passed as an 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 ((5) in Figure 8.2).

Click here to view code image

```
static type parseType(String str) throws NumberFormatException
```

Click here to view code image

```
byte   value1 = Byte.parseByte("16");
int    value2 = Integer.parseInt("2020");       // parseInt, not parseInteger
int    value3 = Integer.parseInt("7UP");        // NumberFormatException.
double value4 = Double.parseDouble("3.14");
double value5 = Double.parseDouble("Infinity");
```

For the integer wrapper types, the overloaded static method `parse`*Type*`()` can additionally take a second argument, which can specify the base in which to interpret the string representing the signed integer in the first argument. Note that the string argument does not specify the prefix for the number system notation.

Click here to view code image

```
type parseType(String str, int base) throws NumberFormatException
```

Click here to view code image

```
byte  value6  = Byte.parseByte("1010", 2);        // Decimal value 10
short value7  = Short.parseShort("12", 8);        // Decimal value 10
short value8  = Short.parseShort("012", 8);       // Decimal value 10
short value9  = Short.parseShort("\012", 8);      // NumberFormatException
int   value10 = Integer.parseInt("-a", 16);       // Decimal value -10
int   value11 = Integer.parseInt("-0xa", 16);     // NumberFormatException
long  value12 = Long.parseLong("-a", 16);         // Decimal value -10L
```

Converting Integer Values to Strings in Different Notations

The wrapper classes `Integer` and `Long` provide static methods for converting integers to text representations 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 8.2 demonstrates the use of these methods.

Click here to view code image

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

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

Click here to view code image

```
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 text representation of the *magnitude* of the integer `i` in the specified base.

The second method is equivalent to the method `toString(int i, int base)`, where the base has the value 10, and which returns the text representation as a signed decimal ((6a) in Figure 8.2).

Example 8.2 *Text Representation of Integers*

Click here to view code image

```
public class IntegerRepresentation {
  public static void main(String[] args) {
    int positiveInt = +41;    // 0b101001, 051, 0x29
    int negativeInt = -41;    // 0b11111111111111111111111111010111, -0b101001,
                              // 037777777727, -051, 0xffffffd7, -0x29
    System.out.println("Text representation for decimal value: " + positiveInt);
    integerStringRepresentation(positiveInt);
    System.out.println("Text representation for decimal value: " + negativeInt);
    integerStringRepresentation(negativeInt);
  }

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

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

Output from the program:

Click here to view code image

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

The `Character` Class

The `Character` class defines a myriad of constants, including the following, which represent the minimum and the maximum values of the `char` type (§2.2, p. 42):

Click here to view code image

```
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 the Unicode standard:

Click here to view code image

```
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:

Click here to view code image

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

The `Boolean` Class

In addition to the common utility methods for wrapper classes discussed earlier in this section, the `Boolean` class defines the following wrapper objects to represent the primitive values `true` and `false`, respectively:

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

Converting Strings to Boolean Values

The wrapper class `Boolean` defines the following static method, which returns the `boolean` value `true` only if the `String` argument is equal to the string `"true"`, ignoring the case; otherwise, it returns the `boolean` value `false`. Note that this method does not throw any exceptions, as its numeric counterparts do.

Click here to view code image

```
static boolean parseBoolean(String str)
```

Click here to view code image

```
boolean b1 = Boolean.parseBoolean("TRUE");      // true.
boolean b2 = Boolean.parseBoolean("true");      // true.
boolean b3 = Boolean.parseBoolean("false");     // false.
boolean b4 = Boolean.parseBoolean("FALSE");     // false.
boolean b5 = Boolean.parseBoolean("not true");  // false.
boolean b6 = Boolean.parseBoolean("null");      // false.
boolean b7 = Boolean.parseBoolean(null);        // false.
```

Review Questions

8.1 Which of the following statements is true?

Select the one correct answer.

a. If the references `x` and `y` denote two different objects, the expression `x.equals(y)` is always `false`.

b. If the references `x` and `y` denote two different objects, the expression `(x.hashCode() == y.hashCode())` is always `false`.

c. The `hashCode()` method in the `Object` class is declared as `final`.

d. The `equals()` method in the `Object` class is declared as `final`.

e. All arrays have a method named `clone`.

8.2 What will be the result of compiling and running the following program?

Click here to view code image

```
public class RQ200A70 {
  public static void main(String[] args) {
    Integer i = Integer.valueOf(-10);
    Integer j = Integer.valueOf(-10);
    Integer k = -10;
    System.out.print((i==j) + "|");
    System.out.print(i.equals(j) + "|");
    System.out.print((i==k) + "|");
    System.out.print(i.equals(k));
  }
}
```

Select the one correct answer.

a. `false|true|false|true`

b. `true|true|true|true`

c. `false|true|true|true`

d. `true|true|false|true`

e. None of the above

8.3 Which statement is true about primitives and wrapper classes?

Select the one correct answer.

a. Java automatically converts primitives to wrapper objects with no performance penalty.

b. Java automatically converts wrapper objects to primitives with no performance penalty.

c. Wrapper references can be assigned the `null` value.

d. A variable of a primitive type can be assigned the `null` value.

e. Auto-unboxing of an uninitialized numeric wrapper reference results in a primitive value of zero.

8.4 Given the following code:

Click here to view code image

```
public class Test {
  public static void main(String[] args) {
    Integer i1 = 10;
    Integer i2 = 10;
    int i3 = 10;
    Integer x1 = i1*i2*i3;
    Integer x2 = i1*i2*i3;
    int x3 = i1*i2*i3;
    String result = (i1 == i2) ? "A" : "";
    result += (i1 == i3) ? "B" : "";
    result += (x1 == x2) ? "C" : "";
    result += (x1 == x3) ? "D" : "";
    System.out.print(result);
  }
}
```

What is the result?

Select the one correct answer.

a. `ABCD`

b. `ABD`

c. `AC`

d. `BD`

e. The program will fail to compile.

8.4 The `String` Class

Handling character sequences is supported primarily by the `String` and `String-Builder` classes. This section discusses the `String` class that provides support for creating, initializing, and manipulating immutable character strings. The next section discusses support for mutable strings provided by the `StringBuilder` class (p. 464).

Internal Representation of Strings

The following character encoding schemes are commonly used for encoding character data on computers:

- LATIN-1: Also known as ISO 8859-1. LATIN-1 is a fixed-length encoding that uses 1 byte to represent a character in the range 00 to FF—that is, characters that can be represented by 8 bits. This encoding suffices for most Western European languages.
- UTF-16: This encoding scheme is a variable-length scheme that uses either 2 bytes or 4 bytes to represent a character in the range 0000 to 10FFFF. This encoding suffices for most languages in the world. However, the `char` type in Java only represents values in the UTF-16 range 0000 to FFFF—that is, characters that can be represented by 2 bytes.

Internally, the character sequence in a `String` object is stored as *an array of* `byte`. If *all* characters in the string can be stored as a *single* byte per character, they are all encoded in the array with the LATIN-1 encoding scheme—1 byte per character. If any character in the sequence requires more than 1 byte, they are all encoded in the array with the UTF-16 encoding scheme—2 bytes per character. To keep track of which encoding is used for the characters in the internal `byte` array, the `String` class has a `private final` encoding-flag field named `coder` which the string methods can consult to correctly interpret the bytes in the internal array. What encoding to use is detected when the string is created. With this strategy of *compact strings*, storage is not wasted as would be the case if all strings were to be encoded in the UTF-16 encoding scheme.

Creating and Initializing Strings

Immutability

The `String` class implements *immutable* character strings, which are read-only once the string has been created and initialized. Objects of the `String` class are thus *thread-safe*, as the state of a `String` object cannot be corrupted through concurrent access by multiple threads. Operations on a `String` object that modify the characters return a new `String` object. The `StringBuilder` class implements mutable strings (p. 464).

String Internment

The easiest way to create a `String` object is to use a string literal:

Click here to view code image

```
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 that is 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 preceding assignment. A string literal can be used to invoke methods on its `String` object:

Click here to view code image

```
int strLength = "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 character sequence. The `String` class maintains a private *string pool* where such strings are interned.

Click here to view code image

```
String str2 = "You cannot change me!";      // Interned string assigned.
```

Both `String` references `str1` and `str2` denote the same interned `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:

Click here to view code image

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

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

Click here to view code image

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

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

Click here to view code image

```
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 as `final` so that no subclass can override this behavior. Internally using both compact strings and string internment optimizes memory allocation and performance for strings, which is fully transparent for programmers.

String Constructors

The `String` class has numerous constructors to create and initialize `String` objects based on various types of arguments. Here we present a few selected constructors:

```
String()
```

Creates a new `String` object, whose content is the *empty string*, `""`.

```
String(String str)
```

Creates a new `String` object, whose contents are the same as those of the `String` object passed as the argument.

Click here to view code image

```
String(char[] value)
String(char[] value, int offset, int count)
```

Create a new `String` object, whose contents are copied from a `char` array. The second constructor allows extraction of a certain number of characters (`count`) from a given `offset` in the specified array.

Click here to view code image

```
String(StringBuilder builder)
```

This constructor allows interoperability with the `StringBuilder` class.

Note that using a constructor creates a brand-new `String` object; using a constructor does *not* intern the string. In the following code, the `String` object denoted by `str4` is different from the interned `String` object passed as an argument:

Click here to view code image

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

Constructing `String` objects can also be done from arrays of bytes, arrays of characters, and string builders:

Click here to view code image

```
byte[] bytes = {97, 98, 98, 97};
char[] characters = {'a', 'b', 'b', 'a'};
StringBuilder strBuilder = new StringBuilder("abba");
//...
String byteStr  = new String(bytes);      // Using array of bytes: "abba"
String charStr  = new String(characters); // Using array of chars: "abba"
String buildStr = new String(strBuilder); // Using string builder: "abba"
```

In Example 8.3, note that the reference `str1` does not denote the same `String` object as the 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 the creation of 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.

The `String` method `intern()` allows the contents of a `String` object to be interned. If a string with the same contents is not already in the string pool, it is added; otherwise, the already interned string is returned. The following relationship holds between any two strings `strX` and `strY`:

Click here to view code image

```
strX.intern() == strY.intern() is true if and only if strX.equals(strY) is true.
```

In Example 8.3, as 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, the call to the `intern()` method on any of these `String` objects will return the same canonical string from the pool. The method call `str4.intern()` will find that a string with its contents (`"You cannot change me"`) already exists in the string pool, and will also return this string from the pool. Thus the object reference comparison `str1.intern() == str4.intern()` will return `true`, but `str4 == str4.intern()` is still `false`, as the `String` object denoted by `str4` is not interned and therefore does not denote the same `String` object as the one that is interned.

```
String intern()
```

If the string pool already contains a string that is equal to this `String` object (determined by the `equals(Object)` method), then the string from the pool is returned. Otherwise, this `String` object is added to the pool and a reference to this `String` object is returned.

Example 8.3 *String Construction and Internment*

Click here to view code image

```java
// File: StringConstruction.java
class Auxiliary {
  static String str1 = "You cannot change me!";          // Interned
}
//______________________________________________________________________________
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

    System.out.println("str1.intern() == str4.intern(): " +
                       (str1.intern() == str4.intern()));// (12) true
    System.out.println("str4 == str4.intern(): " +
                       (str4 == str4.intern()));         // (13) false
  }
}
```

Output from the program:

Click here to view code image

```
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
str1.intern() == str4.intern(): true
str4 == str4.intern(): false
```

The `CharSequence` Interface

This interface defines a readable sequence of `char` values. It is implemented by the `String` and `StringBuilder` classes. Many methods in these classes accept arguments of this interface type, and specify it as their return type. This interface facilitates interoperability between these classes. It defines the following methods:

```
int length()
```

Returns the number of `char` values in this sequence.

```
default boolean isEmpty()
```

Returns `true` if this character sequence is empty. The default implementation returns the result of `this.length() == 0`. See also the `String.isBlank()` method (p. 453).

```
char charAt(int index)
```

A character at a particular index in a sequence can be read using the `charAt()` method. The first character is at index `0` and the last one at index 1 less than the number of characters in the string. If the index value is not valid, an `IndexOutOfBoundsException` is thrown.

Click here to view code image

```
CharSequence subSequence(int start, int end)
```

Returns a new `CharSequence` that is a subsequence of this sequence. Characters from the current sequence are read from the index `start` to the index `end-1`, inclusive.

```
String toString()
```

Returns a string containing the characters in this sequence in the same order as this sequence.

Click here to view code image

```
static int compare(CharSequence cs1, CharSequence cs2)
```

Compares two `CharSequence` instances lexicographically. It returns a negative value, 0, or a positive value if the first sequence is lexicographically less than, equal to, or greater than the second, respectively.

```
default IntStream chars()
```

This default method returns an `IntStream` of `char` values from this sequence (§16.4, p. 901).

Reading Characters from a String

The following methods can be used for character-related operations on a string:

Click here to view code image

```
char charAt(int index)              From the CharSequence interface (p. 444).

char[] toCharArray()
```

Returns a new character array, with length equal to the length of this string, that contains the characters in this string.

Click here to view code image

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

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 (`dst`), 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.

Click here to view code image

```
IntStream chars()                   From the CharSequence interface (p. 444).

int length()                        From the CharSequence interface (p. 444).

boolean isEmpty()                   From the CharSequence interface (p. 444).
```

Reading Lines from a String

The following method can be used to extract lines from a string:

```
Stream<String> lines()
```

Returns a stream of lines extracted from this string, separated by a *line terminator* (§16.4, p. 902). A line is defined as a sequence of characters terminated by a line terminator.

A line terminator is one of the following: a line feed character `"\n"`, a carriage return character `"\r"`, or a carriage return followed immediately by a line feed `"\r\n"`.

Example 8.4 uses some of these methods for reading characters from strings 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 8.4 *Reading Characters from a* `String`

Click here to view code image

```
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() - 3];       // 3 characters less.
    str.getChars( 0,            7, destination, 0);         // (5) "You can"
    str.getChars(10, str.length(), destination, 7);         // (6) " change me!"
                                                            // "not" not copied.
    // Print the character array.
    for (int i = 0; i < destination.length; i++) {
      System.out.print(destination[i]);
    }
    System.out.println();
  }
}
```

Output from the program:

Click here to view code image

```
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 8.4**, 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:

Click here to view code image

```
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 Unicode values.

Click here to view code image

```
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:

Click here to view code image

```
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)
```

The `String` class implements the `Comparable<String>` interface (**§14.4**, **p. 761**). The `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

---

Here are some examples of string comparisons:

Click here to view code image

```
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 = "abba";
String str2 = "aha";

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

Character Case in a String

Click here to view code image

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

Note that the original string is returned if none of the characters needs its 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(),` §18.1, p. 1096), 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.

Examples of case in strings:

Click here to view code image

```
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.toUpperCase();  // 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 new 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 §2.8, p. 63. 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:

Click here to view code image

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

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 advertise"`. 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 new `String` object, whose content is `"Just advertise here."` 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 advertise"` is inevitable at (3).

Some more examples of string concatenation follow:

Click here to view code image

```
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 assignments 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):

Click here to view code image

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

Repeating Strings

The `String` class provides the `repeat()` method that creates a string which is the result of concatenating the current string with itself a specified number of times. The code below illustrates the operation with the `String banner`, which is repeated three times. The method returns a new string, and the original string is intact.

Click here to view code image

```
String banner = "Let's ace that Exam! ";
String bigBanner = banner.repeat(3);
System.out.println("|" + bigBanner + "|");
```

The code prints:

Click here to view code image

```
|Let's ace that Exam! Let's ace that Exam! Let's ace that Exam! |
```

```
String repeat(int count)
```

Returns a string whose value is the concatenation of this string repeated `count` number of times. If this string is empty (i.e., its length is zero) then the empty string is returned.

Joining of `CharSequence` Objects

One operation commonly performed on a sequence of strings is to format them so that each string is separated from the next one by a delimiter. For example, given the following sequence of strings:

```
"2014"
"January"
"11"
```

we wish to format them so that individual strings are separated by the delimiter "/":

```
"2014/January/11"
```

The following static methods in the `String` class can be used for this purpose:

Click here to view code image

```
static String join(CharSequence delimiter, CharSequence... elements)
static String join(CharSequence delimiter,
                   Iterable<? extends CharSequence> elements)
```

Both static methods return a new `String` composed of copies of the `CharSequence` elements joined together with a copy of the specified `CharSequence` delimiter. Thus the resulting string is composed of text representations of the elements separated by the text representation of the specified delimiter.

If either `delimiter` is `null` or `elements` is `null`, a `NullPointerException` is thrown. If both are `null`, the method call is ambiguous.

If an element in `elements` is `null`, the string `"null"` is added as its text representation.

Note that both the individual strings and the delimiter string are of type `CharSequence`. The examples in this section use `String` and `StringBuilder` objects that implement the `CharSequence` interface (p. 444).

An `Iterable` provides an iterator to iterate over its elements. The following examples use an `ArrayList` (§12.5, p. 657) that implements the `Iterable` interface. The second `join()` method is then able to iterate over the `Iterable` using the iterator. This method will accept only an `Iterable` whose elements are of type `CharSequence` or are subtypes of `CharSequence`.

The first example shows joining of `String` objects. The first `join()` method is called in this case.

Click here to view code image

```
// (1) Joining individual String objects:
String dateStr = String.join("/", "2014", "January", "11");
System.out.println(dateStr);                  // 2014/January/11
```

The second example shows joining of elements in a `StringBuilder` array. Again the first `join()` method is called, with the array being passed as the second parameter.

Click here to view code image

```
// (2) Joining elements in a StringBuilder array:
StringBuilder left          = new StringBuilder("Left");
StringBuilder right         = new StringBuilder("Right");
StringBuilder[] strBuilders = { left, right, left };
String march = String.join("-->", strBuilders);
System.out.println(march);                    // Left-->Right-->Left
```

The third example shows joining of elements in an `ArrayList` of `StringBuilder`. The second `join()` method is called, with the `ArrayList` being passed as the second parameter. Note that some of the elements of the `ArrayList` are `null`.

Click here to view code image

```
// (3) Joining elements in a StringBuilder list:
ArrayList<StringBuilder> sbList = new ArrayList<>();
sbList.add(right); sbList.add(null); sbList.add(left); sbList.add(null);
String resultStr = "[" + String.join(", ", sbList) + "]";
System.out.println(resultStr);                // [Right, null, Left, null]
```

The last example shows joining of elements in an `ArrayList` of `CharSequence`. Again the second `join()` method is called, with the `ArrayList` being passed as the second parameter. Note that elements of the `ArrayList` are `String` and `StringBuilder` objects that are also of type `CharSequence`.

Click here to view code image

```
// (4) Joining elements in a CharSequence list:
ArrayList<CharSequence> charSeqList = new ArrayList<>();
charSeqList.add(right); charSeqList.add(left);     // Add StringBuilder objects.
charSeqList.add("Right"); charSeqList.add("Left"); // Add String objects.
String resultStr2 = "<" + String.join("; ", charSeqList) + ">";
System.out.println(resultStr2);               // <Right; Left; Right; Left>
```

Searching for Characters and Substrings in Strings

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.

Click here to view code image

```
int indexOf(int ch)
int indexOf(int ch, int fromIndex)
```

The first method finds the index of the first occurrence of the argument character in a string. The second method 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, resulting in the value `-1` being returned.

Click here to view code image

```
int indexOf(String str)
int indexOf(String str, int fromIndex)
```

The first method finds the start index of the first occurrence of the substring argument in a string. The second method 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.

Click here to view code image

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

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

Click here to view code image

```
String replace(char oldChar, char newChar)
String replace(CharSequence target, CharSequence replacement)
```

The first method returns a new `String` object that is the result of replacing all occurrences of the `oldChar` in the current string with the `newChar`. The current string is returned if no occurrences of the `oldChar` can be found.

The second method returns a new `String` object that is the result of replacing all occurrences of the character sequence `target` in the current string with the character sequence `replacement`. The current string is returned if no occurrences of the `target` can be found.

The following methods can be used to test whether a string satisfies a given criterion:

Click here to view code image

```
boolean contains(CharSequence cs)
```

Returns `true` if the current string contains the specified character sequence, and `false` otherwise.

Click here to view code image

```
boolean startsWith(String prefix)
```

Returns `true` if the current string starts with the character sequence specified by parameter `prefix`, and `false` otherwise.

Click here to view code image

```
boolean startsWith(String prefix, int index)
```

Returns `true` if the substring of the current string at the specified `index` starts with the character sequence specified by parameter `prefix`, and `false` otherwise.

Click here to view code image

```
boolean endsWith(String suffix)
```

Returns `true` if the current string ends with the character sequence specified by parameter `suffix`, and `false` otherwise.

Examples of search and replace methods in the `String` class:

Click here to view code image

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

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

String newStr    = funStr.replace('J', 'W');      // "Wava Wives"
String newBanner = banner.replace("One", "No");   // "No man, No vote"
boolean found1   = banner.contains("One");        // true
boolean found2   = newBanner.contains("One");     // false

String song = "Start me up!";
//              012345677890
boolean found3    = song.startsWith("Start");     // true
boolean notFound1 = song.startsWith("start");     // false
boolean found4    = song.startsWith("me", 6);     // true
boolean found5    = song.endsWith("up!");         // true
boolean notFound2 = song.endsWith("up");          // false
```

Extracting Substrings from Strings

The `String` class provides methods to trim and strip strings, and also extract substrings.

```
boolean isBlank()
```

Returns `true` if the string is empty or contains only *whitespace*; otherwise, it returns `false`. See also the method `isEmpty()` in the `CharSequence` interface (p. 444).

```
String strip()
String stripLeading()
String stripTrailing()
```

Return a string whose value is this string, with all leading and trailing *whitespace* removed, or with all leading whitespace removed, or with all trailing whitespace removed, respectively. If this `String` object represents an empty string, or if all characters in this string are whitespace, then an empty string is returned. See also the method `stripIndent()` (p. 444).

```
String trim()
```

This method can be used to create a string where all characters with values less than or equal to the space character `'\u0020'` have been removed from the front (leading) and the end (trailing) of a string. It is recommended to use the strip methods to remove leading and trailing whitespace in a string.

Click here to view code image

```
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, an `IndexOutOfBoundsException` is thrown.

Click here to view code image

```
CharSequence subSequence(int start, int end) From the CharSequence interface
(p. 444)
```

This method behaves the same way as the `substring(int start, int end)` method.

The `Character.isWhitespace()` method can be used to determine whether a character is whitespace.

Click here to view code image

```
System.out.println(Character.isWhitespace('\t'));  // true
System.out.println(Character.isWhitespace('\n'));  // true
System.out.println(Character.isWhitespace('a'));   // false
```

Examples of blank strings:

Click here to view code image

```
System.out.println("".isBlank());        // true
System.out.println(" \t  ".isBlank());   // true
```

Examples of stripping a string:

Click here to view code image

```
String utopia = "\t\n  Java Nation \n\t  ";
System.out.println(utopia.strip().equals("Java Nation"));               // true
System.out.println(utopia.stripLeading().equals("Java Nation \n\t  ")); // true
System.out.println(utopia.stripTrailing().equals("\t\n  Java Nation")); // true
```

Examples of extracting substrings:

Click here to view code image

```
String utopia2 = "\t\n  Java Nation \n\t  ";
utopia2 = utopia2.trim();                      // "Java Nation"
utopia2 = utopia2.substring(5);                // "Nation"
String radioactive = utopia2.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:

Click here to view code image

```
String toString()                 From the CharSequence interface (p. 444).
```

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

Click here to view code image

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

All of these methods return a string representing the given parameter value. A call to the method with the parameter `obj` is equivalent to `obj.toString()` when `obj` is not `null`; otherwise, the `"null"` string is returned. 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.

Click here to view code image

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

The static `valueOf()` method, which accepts a primitive value as an argument, is equivalent to the static `toString()` method in the corresponding wrapper class for each of the primitive data types ((6a) and (6b) in §8.3, p. 430). Note that there are no `valueOf()` methods that accept a `byte` or a `short`.

Examples of string conversions:

Click here to view code image

```
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"
```

Transforming a String

The `transform()` method allows a function to be applied to a string to compute a result. The built-in functional interface `Function<T, R>`, and writing lambda expressions to implement such functions, are covered in the following sections: §13.8, p. 712, and §13.2, p. 679, respectively.

Click here to view code image

```
<R> R transform(Function<? super String,? extends R> f)
```

The specified function `f` is applied to this string to produce a result of type `R`.

The example below illustrates using the `transform()` method to reverse the characters in a string via a `StringBuilder` (p. 464). Its versatility becomes apparent when the string contains lines of text or a text block that needs to be processed (p. 458).

Click here to view code image

```
String message = "Take me to your leader!";
String tongueSpeake = message.transform(s ->
    new StringBuilder(s).reverse().toString().toUpperCase());
System.out.println(tongueSpeake); // !REDAEL RUOY OT EM EKAT
```

Indenting Lines in a String

The `indent()` method allows indentation of lines in a string to be adjusted.

```
String indent(int n)
```

Adjusts the indentation of each line of this string based on the value of `n`, and normalizes line termination characters.

This string is conceptually separated into lines using the `String.lines()` method (p. 445). Each line is then adjusted depending on the value of `n`, and then terminated with a line feed (`"\n"`). The resulting lines are then concatenated and the final string returned.

If `n > 0` then `n` spaces are inserted at the beginning of each line. This also applies to an empty line.

If `n == 0` then the line remains unchanged. However, line terminators are still normalized.

If `n < 0` then at most `n` whitespace characters are removed from the beginning of each line, depending on the number of leading whitespace characters in the line. Each whitespace character is treated as a single character; specifically, the tab character `"\t"` is considered a single character and is not expanded.

The example below illustrates how indentation is adjusted depending on the value passed to the `indent()` method. The string `cmds` constructed at (0) has four lines, where the third line is an empty line. The first, second, and last lines are indented by one, two, and three spaces, respectively. The first line is terminated with the return character (`\r`), and the last line is not terminated at all. The other lines are terminated normally with a newline character (`\n`). Keep also in mind that the string `cmds` is immutable.

Click here to view code image

```
String cmds = " Attention!\r  Quick march!\n\n   Eyes left!"; // (0)
String str1 = cmds.indent(0);                                 // (1)
String str2 = cmds.indent(3);                                 // (2)
String str3 = cmds.indent(-1);                                // (3)
String str4 = cmds.indent(-2);                                // (4)
String str5 = cmds.indent(-3);                                // (5)
```

The lines numbered (1) to (5) below show the resulting string from the method calls above from (1) to (5), respectively, applied to the string `cmds` at (0) above. The `jshell` tool can readily be used to execute the method calls above to examine the resulting strings. Note that termination of all lines is normalized with the newline character (`\n`).

Click here to view code image

```
(0) cmds ==> " Attention!\r  Quick march!\n\n   Eyes left!"
(1) str1 ==> " Attention!\n  Quick march!\n\n   Eyes left!\n"
(2) str2 ==> "    Attention!\n     Quick march!\n   \n      Eyes left!\n"
(3) str3 ==> "Attention!\n Quick march!\n\n  Eyes left!\n"
(4) str4 ==> "Attention!\nQuick march!\n\n Eyes left!\n"
(5) str5 ==> "Attention!\nQuick march!\n\nEyes left!\n"
```

In Table 8.1, the columns correspond to the resulting strings from (1) to (5) above. The last row in Table 8.1 shows the output when the resulting strings are printed.

The underscore (`_`) in each cell in the last row is a visual marker that indicates the position after printing the resulting string. Note that the empty line at (0) above is indented (marked visually as `sss`), as shown in column (2). A value less than `-3` does not change the result shown in column (5).

Table 8.1 *Indenting Lines in a String*

(1)	**(2)**	**(3)**	**(4)**	**(5)**
`n = 0`	`n = 3`	`n = -1`	`n = -2`	`n = -3`
`123456789`	`123456789`	`123456789`	`123456789`	`123456789`
` Attention!`	`    Attention!`	`Attention!`	`Attention!`	`Attention!`
`  Quick march!`	`        Quick march!`	`Quick march!`	`Quick march!`	`Quick marc`
	`sss`			
`   Eyes left!`	`        Eyes left!`	` Eyes left!`	`Eyes left!`	`Eyes left!`
`_`	`_`	`_`	`_`	`_`

Formatted Strings

We have used the `System.out.printf()` method to format values and print them to the terminal window (§1.9, p. 24). To just create the string with the formatted values, but not print the formatted result, we can use the following static method from the `String` class. It accepts the same arguments as the `printf()` method, and uses the same format specifications (Table 1.2, p. 26).

Click here to view code image

```
static String format(String format, Object... args)
```

Returns a string with the result of formatting the values in the variable arity parameter `args` according to the `String` parameter `format`. The format string contains format specifications that determine how each subsequent value in the variable arity parameter `args` will be formatted.

Any error in the format string will result in a runtime exception.

Click here to view code image

```
String formatted(Object... args)
```

Formats the supplied arguments using this string as the format string. It is equivalent to `String.format(this, args)`.

The following call to the `format()` method creates a formatted string with the three values formatted according to the specified format string:

Click here to view code image

```
String formattedStr = String.format("Formatted values|%5d|%8.3f|%5s|",
                                    2020, Math.PI, "Hi");
System.out.println(formattedStr);  // Formatted values| 2020|   3.142|   Hi|
formattedStr = formattedStr.toUpperCase();
System.out.println(formattedStr);  // FORMATTED VALUES| 2020|   3.142|   HI|
```

Alternatively, we can use the `formatted()` method:

Click here to view code image

```
String formattedStr1 = "Formatted values|%5d|%8.3f|%5s|"
                             .formatted(2020, Math.PI, "Hi");
System.out.println(formattedStr1); // Formatted values| 2020|   3.142|   Hi|
```

For formatting strings in a language-neutral way, the `MessageFormat` class can be considered (§18.7, p. 1139). Other miscellaneous methods exist in the `String` class for pattern matching (`matches()`), splitting strings (`split()`), and converting a string to an array of bytes (`get-Bytes()`). The method `hashCode()` can be used to compute a hash value based on the charac-ters in the string. Please consult the Java SE Platform API documentation for more details.

Text Blocks

Constructing string literals that span multiple lines can be tedious using string concatenation and line terminators. Text blocks provide a better solution—avoiding having to escape tab, newline, and double-quote characters in the text, and in addition preserving indentation of a multiline string.

Basic Text Blocks

The string `sql1` below represents a three-line SQL query. It is constructed using a *text block*, resulting in an object of type `String`. We will use it as an example to illustrate constructing text blocks.

Click here to view code image

```
String sql1 = """
SELECT *
FROM Programmers
WHERE Language = 'Java';
""";
```

The string literal resulting from entering the text block in the `jshell` tool:

Click here to view code image

```
sql1 ==> "SELECT *\nFROM Programmers\nWHERE Language = 'Java';\n";
```

Printing the text block `sql1` above will give the following result, where the underscore (`_`) is used as a visual marker to indicate the position after printing the resulting string:

Click here to view code image

```
SELECT *
FROM Programmers
WHERE Language = 'Java';
_
```

A text block starts with three double quotes (`"""`) and ends with three double quotes (`"""`). The *opening delimiter* of a text block consists of the opening three double quotes (followed by any spaces, tabs, or form feed characters) that must be terminated by a line terminator. The *closing delimiter* of a text block consists of three double quotes.

The *content of a text block* begins with the sequence of characters that immediately *begins after the line terminator* of the opening delimiter and *ends with the first double quote* of the closing delimiter. Note that the line terminator of the opening delimiter is *not* part of the text block's content.

The following attempt to construct a text block will result in a compile-time error because of erroneous characters (i.e., the `//` comment) that follow the opening three double quotes of the text block.

Click here to view code image

```
String badBlock = """      // (1) Compile-time error!
Not a good start.
""";
```

The text block below results in an empty string (`""`):

Click here to view code image

```
String emptyBlock = """
""";
```

Using Escape Sequences in Text Blocks

We do not need to end each text line with the `\n` escape sequence, as the text block retains the line terminators entered directly into the text. However, using the `\n` escape sequence will be interpreted as literally inserting a newline character in the text. The query `sql2` below is equivalent to the query `sql1` above, and will result in the same string. (For escape sequences, see Table 2.10, p. 38.)

Click here to view code image

```
String sql2 = """
SELECT *
FROM Programmers\nWHERE Language = 'Java';
""";
```

The string literal resulting from entering the text block in the `jshell` tool:

Click here to view code image

```
sql2 ==> "SELECT *\nFROM Programmers\nWHERE Language = 'Java';\n"
```

However, ending a line explicitly in the text block with a newline character will result in an empty line being inserted into the resulting string, as can be seen in the code below:

Click here to view code image

```
String sql3 = """
SELECT *
FROM Programmers\n
WHERE Language = 'Java';
""";
```

The string literal resulting from entering the text block in the `jshell` tool:

Click here to view code image

```
sql3 ==> "SELECT *\nFROM Programmers\n\nWHERE Language = 'Java';\n"
```

Printing the text block `sql3` above will give the following result, where the underscore (`_`) is used as a visual marker to indicate the position after printing the resulting string:

Click here to view code image

```
SELECT *
FROM Programmers

WHERE Language = 'Java';
_
```

In the examples so far, the closing delimiter of a text block was specified on the last line by itself, resulting in the *last line of text* in the text block being terminated by the line terminator that was entered directly. If the last line of text should not be terminated, the closing delimiter can be used at the end of this line as shown below.

Click here to view code image

```
String sql4 = """
SELECT *
FROM Programmers
WHERE Language = 'Java';"""; // No line terminator. Closing delimiter ends block.
```

The string literal resulting from entering the text block in the `jshell` tool shows no line terminator for the last line of the query:

Click here to view code image

```
sql4 ==> "SELECT *\nFROM Programmers\nWHERE Language = 'Java';"
```

T`he` print statement below does not print a newline after the last line:

Click here to view code image

```
System.out.print(sql4);
SELECT *
FROM Programmers
WHERE Language = 'Java';
```

Other escape sequences (Table 2.10, p. 38) can also be used in a text block. In contrast to the `\n` escape sequence, the |*Line terminator* escape sequence, allowed only in a text block, escapes the line terminator, thus preventing the termination of the current line so that it is joined with the next line.

Click here to view code image

```
String sql5 = """
SELECT * \
FROM Programmers \
WHERE Language = 'Java';
""";
```

The string literal resulting from entering the text block in the `jshell` tool shows that the three lines were joined and no characters were substituted for the |*Line terminator* escape sequence:

Click here to view code image

```
sql5 ==> "SELECT * FROM Programmers WHERE Language = 'Java';\n"
```

The result from printing the string `sql5` is shown below, where the underscore (`_`) is used as a visual marker to indicate the position after printing the resulting string.

Click here to view code image

```
SELECT * FROM Programmers WHERE Language = 'Java';
_
```

To include a backslash (`\`) in any context other than line continuation will require escaping the backslash, analogous to using it in a string literal.

By default, any trailing whitespace on a line is removed and line termination normalized. In some cases, it might be necessary to retain trailing whitespace. This can be achieved by using the `\s` escape sequence. Replacing the last trailing space that should be retained with this escape sequence will preserve any trailing whitespace *before* the `\s` escape sequence, including the space replaced by the escape sequence. At (1) below, *four* trailing spaces are retained.

```
String sql6 = """
SELECT *
```

Click here to view code image

```
FROM Programmers   \s     """;                    // (1) No line termination.
// sql6 ==> "SELECT *\nFROM Programmers    "

String sql7 = "WHERE Language = 'Java';\n";
System.out.print(sql6 + sql7);
```

Output from the code, showing the retained spaces, where the underscore (_) is used as a visual marker to indicate the position after printing the resulting string:

Click here to view code image

```
SELECT *
FROM Programmers    WHERE Language = 'Java';
_
```

Double quotes (") are treated like any other character and can be used without escaping them in a text block, except when a three double quote sequence is used in the text. It is enough to escape the first double quote in a three double quote sequence, but all three double quotes can also be escaped individually in the sequence. It is also not required that double quotes should be balanced. The code below illustrates using double quotes in a text block.

Click here to view code image

```
String fact = """
The sequence \"""
has "special" meaning in a text block.
""";
// fact ==> "The sequence \"\"\"\nhas \"special\" meaning in a text block.\n"
System.out.print(fact);
```

Output from the code, where the underscore (_) is used as a visual marker to indicate the position after printing the resulting string:

Click here to view code image

```
The sequence """
has "special" meaning in a text block.
_
```

A text block can be used to define the *format string* for value substitution in formatted text (p. 457). Below, the text block defines the format string used by the `formatted()` method for substituting values of its arguments.

Click here to view code image

```
String query = """
SELECT *
FROM %s
WHERE %s = '%s';
""".formatted("Customers", "Country", "Norway");
System.out.print(query);
```

Output from the print statement, where the underscore (_) is used as a visual marker to indicate the position after printing the resulting string:

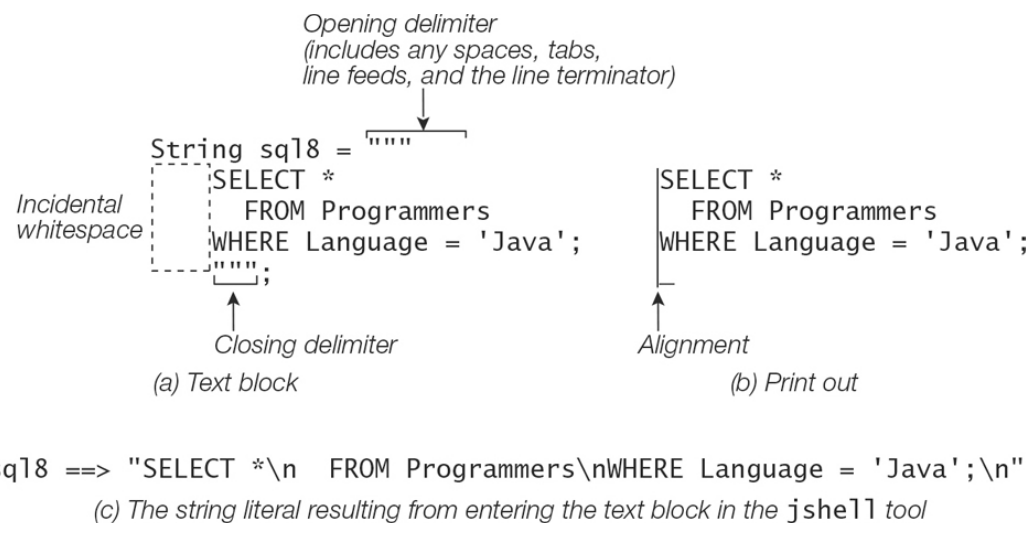
Click here to view code image

```
SELECT *
FROM Customers
WHERE Country = 'Norway';
_
```

Incidental Whitespace

So far the lines in the text blocks have all been left-justified. This need not be the case. Indentation can be used if desired. However, one needs to be careful as *incidental whitespace* from *each* line is removed by the compiler. Incidental whitespace is indentation that is common to all lines in the text block. It is equal to the number of leading whitespace characters in the *least indented* lines in the text block. The least indented lines end up with no indentation. The line containing the closing delimiter also plays a part in determining the incidental whitespace to be removed from all lines. Regardless of how much incidental whitespace is removed from the other lines in the text box, *all* leading whitespace preceding the closing delimiter is always removed, resulting in an empty line.

Whitespace that is not incidental is referred to as *essential whitespace*.



Opening delimiter
(includes any spaces, tabs,
line feeds, and the line terminator)
Incidental
whitespace
Closing delimiter
(a) Text block
Alignment
(b) Print out
(c) The string literal resulting from entering the text block in the jshell tool

```
String sql8 = """
    SELECT *
      FROM Programmers
    WHERE Language = 'Java';
    """;
```

```
SELECT *
  FROM Programmers
WHERE Language = 'Java';
_
```

```
sql8 ==> "SELECT *\n  FROM Programmers\nWHERE Language = 'Java';\n"
```

Figure 8.3 *Incidental Whitespace in Text Blocks*

In **Figure 8.3a**, we see that the first and third lines, and the line with the closing delimiter, are the ones that are least indented in the text block. The number of leading whitespace characters in each of these lines is four. The indentation in the second line is six positions. The least number (i.e., four) of leading whitespace characters common to all lines is the incidental whitespace that is removed from *each* line. **Figure 8.3b** shows how the contents of the text block will be printed. The underscore (`_`) is a visual marker that indicates the position after printing the text block, showing that a newline is printed for the last line of the text block. Removing the incidental whitespace preserves the relative indentation of the lines in the text block. **Figure 8.3c** shows the string literal resulting from entering the text block in the `jshell` tool.

As stated earlier, the whitespace preceding the closing delimiter is significant in determining incidental whitespace. **Table 8.2** shows a few more examples to illustrate how incidental whitespace is calculated and removed in a text block. The last two rows show the text block declaration and the result of printing the text block, respectively. The underscore (`_`) in each cell in the last row is a visual marker that indicates the position after printing the text block, showing that no newline is printed for the last line of the text blocks in columns (4) and (5). The numbered remarks below refer to the column numbers in **Table 8.2**. The string literal resulting from entering each text block in the `jshell` tool is also shown.

(1) The least indentation (0) is determined by the last line with the closing delimiter. No incidental whitespace is removed.
Click here to view code image

```
tb1 ==> "    KEEP\n      IT\n         SIMPLE\n"
```

(2) The least indentation (2) is determined by the last line with the closing delimiter.
Click here to view code image

```
tb2 ==> "  KEEP\n    IT\n      SIMPLE\n"
```

(3) The least indentation (4) is determined by the line containing the word `IT`. However, *all* leading whitespace in the *last* line is removed, not just four spaces as in the other lines—we can see the result in the string literal below:
Click here to view code image

```
tb3 ==> "  KEEP\nIT\n  SIMPLE\n"
```

(4) The least indentation (2) is determined by the last line containing the word `SIMPLE` and the closing delimiter. Note that the last line is not terminated.
Click here to view code image

```
tb4 ==> "  KEEP\n    IT\nSIMPLE"
```

(5) The least indentation (4) is determined by the lines containing the words `KEEP` and `IT`, respectively. Note that in the last line only incidental whitespace is removed, and the line is not terminated.
Click here to view code image

```
tb5 ==> "KEEP\nIT\n  SIMPLE"
```

Table 8.2 *Incidental Whitespace in Text Blocks*

(1)	**(2)**	**(3)**	**(4)**	**(5)**
Least indent is 0	**Least indent is 2**	**Least indent is 4**	**Least indent is 2**	**Least indent is**
`123456789012345`	`123456789012345`	`123456789012345`	`123456789012345`	`12345678901234`

```
String tb1 =          String tb2 =          String tb3 =         String tb4 =         String tb5
"""                   """                   """                  """                  """
    KEEP                  KEEP                    KEEP               KEEP                 KEEP
      IT                    IT                  IT                     IT                 IT
        SIMPLE                SIMPLE              SIMPLE           SIMPLE""";               SIMPL
""";                    """;                         """;
```

```
    KEEP                KEEP                  KEEP                 KEEP               KEEP
      IT                  IT                IT                       IT               IT
        SIMPLE              SIMPLE            SIMPLE               SIMPLE               SIMPLE

_                     _                     _                    _                    _
```

A blank line (i.e., one that is empty or contains only whitespace) in a text block is replaced by an empty line—that is, any whitespace it contains is removed.

Incidental whitespace in a text block is processed internally as if by the execution of the `String.stripIndent()` method. Calling this method on a text block with one or more lines has *no* effect on the content of the text block, but it will strip any whitespace from a text block that has *no line structure*.

The reader is encouraged to use the `jshell` tool to examine the string literal resulting from entering the text blocks presented in this section.

```
String stripIndent()
```

Returns a string whose value is this string, with incidental whitespace removed from the beginning and end of every line.

8.5 The `StringBuilder` Class

Although there is a close relationship between objects of the `String` and `StringBuilder` classes, these are two independent `final` classes, both directly extending the `Object` class. Hence, `String` references cannot be stored (or cast) to `StringBuilder` references, and vice versa. However, both classes implement the `CharSequence` interface (**p. 444**) and the `Comparable` interface (**§14.4**, **p. 761**).

Since the `StringBuilder` class does not override the `equals()` and `hashCode()` methods from the `Object` class, the contents of string builders should be converted to `String` objects for equality comparison and to compute a hash value.

The `StringBuilder` class has many operations analogous to the ones in the `String` class. In addition, it provides support for manipulating *mutable* strings (**p. 467**).

Constructing String Builders

Mutability

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

The legacy class `StringBuffer` (**Figure 8.1**, **p. 425**) also implements *mutable* sequences of characters. It support the same operations as the `StringBuilder` class, but the `StringBuffer` class is *thread-safe*. Certain operations on a string buffer are synchronized so that when accessed concurrently by multiple threads, these operations are safe to perform without corrupting the state of the string buffer (**§22.4**, **p. 1387**). Note that a `String` object is also thread-safe—because it is immutable, a thread cannot change its state. String builders are preferred when heavy modification of character sequences is involved and synchronization of operations, which also carries a performance penalty, is not important. Although the rest of this section focuses on string builders, it is equally applicable to string buffers.

String Builder Constructors

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

Click here to view code image

```
StringBuilder(String str)
StringBuilder(CharSequence charSeq)
```

The contents of the new `StringBuilder` object are the same as the contents of the `String` object or the character sequence passed as an argument. The initial capacity of the string builder is set to the length of the argument sequence, plus room for 16 more characters.

Click here to view code image

```
StringBuilder(int initialCapacity)
```

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

```
StringBuilder()
```

This constructor also creates a new `StringBuilder` object with no content. The initial capacity of the string builder is set to 16 characters.

Examples of `StringBuilder` object creation and initialization:

Click here to view code image

```
StringBuilder strBuilder1 = new StringBuilder("Phew!");  // "Phew!", capacity 21
StringBuilder strBuilder2 = new StringBuilder(10);       // "", capacity 10
StringBuilder strBuilder3 = new StringBuilder();         // "", capacity 16
```

Reading Characters from String Builders

The following methods can be used for reading characters in a string builder:

Click here to view code image

```
char charAt(int index)                From the CharSequence interface (p. 444).

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

Copies characters from the current string builder into the destination character array. Characters from the current string builder are read from index `srcBegin` to index `srcEnd-1`, inclusive. They are copied into the destination array (`dst`), 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.

Click here to view code image

```
IntStream chars()                     From the CharSequence interface (p. 444).

int length()                          From the CharSequence interface (p. 444).
```

The following is an example of reading the contents of a string builder:

Click here to view code image

```
StringBuilder strBuilder = new StringBuilder("Java");      // "Java", capacity 20
char charFirst = strBuilder.charAt(0);                     // 'J'
char charLast  = strBuilder.charAt(strBuilder.length()-1); // 'a'
```

Searching for Substrings in String Builders

The methods for searching substrings in string builders are analogous to the ones in the `String` class (p. 451).

Click here to view code image

```
int indexOf(String substr)
int indexOf(String substr, int fromIndex)

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

Examples of searching for substrings in a `StringBuilder`:

Click here to view code image

```
StringBuilder banner2 = new StringBuilder("One man, One vote");
//                                         01234567890123456
int subInd1a = banner2.indexOf("One");          // 0
int subInd1b = banner2.indexOf("One", 3);       // 9
int subInd2a = banner2.lastIndexOf("One");      // 9
int subInd2b = banner2.lastIndexOf("One", 10); // 9
int subInd2c = banner2.lastIndexOf("One", 8);  // 0
int subInd2d = banner2.lastIndexOf("One", 2);  // 0
```

Extracting Substrings from String Builders

The `StringBuilder` class provides the following methods to extract substrings, which are also provided by the `String` class (p. 453).

Click here to view code image

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

CharSequence subSequence(int start, int end)
```
From the `CharSequence` interface (p. 444).

Examples of extracting substrings:

Click here to view code image

```
StringBuilder bookName = new StringBuilder("Java Gems by Anonymous");
//                                          01234567890123456789012
String title  = bookName.substring(0,9);   // "Java Gems"
String author = bookName.substring(13);    // "Anonymous"
```

Constructing Strings from String Builders

The `StringBuilder` class overrides the `toString()` method from the `Object` class (see also the `CharSequence` interface, p. 444). It returns the contents of a string builder as a `String` object.

Click here to view code image

```
String toString()                    From the CharSequence interface (p. 444).
```

Click here to view code image

```
StringBuilder strBuilder = new StringBuilder("Build or not to build.");
String fromBuilder = strBuilder.toString();     // "Build or not to build."
```

Comparing String Builders

String builders implement the `Comparable<StringBuilder>` interface and can be compared lexicographically, analogous to `String` objects (p. 447).

Click here to view code image

```
int compareTo(StringBuilder anotherSB)
```

Here are some examples of string builder comparisons:

Click here to view code image

```
StringBuilder sb1 = new StringBuilder("abba");
StringBuilder sb2 = new StringBuilder("aha");

int compValue1 = sb1.compareTo(sb2);         // negative value => sb1 < sb2
int compValue2 = sb2.compareTo(sb1);         // positive value => sb2 > sb1
boolean isEqual = sb1.compareTo(sb2) == 0;                // false
boolean flag    = sb1.toString().equals(sb2.toString()); // false
```

Modifying String Builders

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

Note that the methods in this subsection return the reference value of the modified string builder, making it convenient to chain calls to these methods.

Appending Characters to a String Builder

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

Click here to view code image

```
StringBuilder 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 builder.

Click here to view code image

```
StringBuilder append(String str)
StringBuilder append(CharSequence charSeq)
StringBuilder append(CharSequence charSeq, int start, int end)
StringBuilder append(char[] charArray)
StringBuilder append(char[] charArray, int offset, int length)
StringBuilder append(char c)
```

Allow characters from various sources to be appended to the end of the current string builder.

Click here to view code image

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

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 builder.

Inserting Characters in a String Builder

The overloaded method `insert()` can be used to *insert* characters at a *given offset* in a string builder.

Click here to view code image

```
StringBuilder insert(int offset, Object obj)
StringBuilder insert(int dstOffset, CharSequence seq)
StringBuilder insert(int dstOffset, CharSequence seq, int start, int end)
StringBuilder insert(int offset, String str)
StringBuilder insert(int offset, char[] charArray)
StringBuilder insert(int offset, type c)
```

In the methods above, *type* can be `char`, `boolean`, `int`, `long`, `float`, or `double`. The second argument is converted to a string, if necessary, by applying the static method `String.valueOf()`. The `offset` argument specifies where the characters are to be inserted in the string builder, and must be greater than or equal to 0. Note that the `offset` specifies the number of characters from the start of the string builder.

Replacing Characters in a String Builder

The following methods can be used to replace characters in a string builder:

Click here to view code image

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

Changes the character at a specified index in the string builder. An `IndexOutOfBoundsException` is thrown if the index is not valid. This method does not change the length of the string builder, and does *not* return a value.

Click here to view code image

```
StringBuilder replace(int start, int end, String replacement)
```

Replaces the characters in a subsequence of the string builder with the characters in the specified `String`. The subsequence is defined by the `start` index (inclusive) and the `end` index (exclusive). It returns the modified string builder. If the `start` index is not valid (i.e., `start` index is negative, greater than `length()`, or greater than `end` index.), a `StringIndexOutOfBoundsException` is thrown.

Deleting Characters in a String Builder

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

Click here to view code image

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

The first method deletes a character at a specified index in the string builder, contracting the string builder by one character. The second method deletes a substring, which is specified by the `start` index (inclusive) and the `end` index (exclusive), contracting the string builder accordingly. If `start` index is equal to `end` index, no changes are made.

Reversing Characters in a String Builder

The following method in the class `StringBuilder` reverses the contents of a string builder:

```
StringBuilder reverse()
```

Examples of using methods that modify string builders:

Click here to view code image

```
StringBuilder builder = new StringBuilder("banana split");   // "banana split"
builder.delete(4,12);                                        // "bana"
builder.append(42);                                          // "bana42"
builder.insert(4,"na");                                      // "banana42"
builder.reverse();                                           // "24ananab"
builder.deleteCharAt(builder.length()-1);                    // "24anana"
builder.replace(0, 2, "b");                                  // "banana"
builder.append('s');                                         // "bananas"
```

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

Click here to view code image

```
builder.delete(4,12).append(42).insert(4,"na").reverse()
       .deleteCharAt(builder.length()-1).replace(0, 2, "b")
       .append('s');                                      // "bananas"
```

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

Click here to view code image

```
((((((builder.delete(4,12)).append(42)).insert(4,"na")).reverse())
      .deleteCharAt(builder.length()-1)).replace(0, 2, "b"))
      .append('s');                                       // "bananas"
```

Each method call returns the reference value of the modified string builder, which is then used to invoke the next method. The string builder remains denoted by the reference `builder`.

The compiler uses string builders to implement string concatenation with the `+` operator in `String`-valued non-constant expressions. The following code illustrates this optimization:

Click here to view code image

```
String theChosen = "U";
String str1 = 4 + theChosen + "Only";           // (1) Non-constant expression.
```

The assignment statement at (1) is equivalent to the following code using a string builder:

Click here to view code image

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

The code at (2) does not create any temporary `String` objects when concatenating several strings, since a single `StringBuilder` object is modified and finally converted to a `String` object having the string content `"4UOnly"`.

Controlling String Builder Capacity

The following methods are exclusive to the `StringBuilder` class and can be used to control various capacity-related aspects of a string builder:

```
int capacity()
```

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

Click here to view code image

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

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

```
void trimToSize()
```

Attempts to reduce the storage used for the character sequence. It may affect the capacity of the string builder.

Click here to view code image

```
void setLength(int newLength)
```

Ensures that the actual number of characters—that is, the length of the string builder—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 affects the capacity of the string builder only if the value of the parameter `newLength` is greater than the current capacity.

Example 8.5 illustrates the various capacity-related methods of the `StringBuilder` class. It is instructive to go through the output to see how these methods affect the length and the capacity of a string builder.

Example 8.5 *Controlling String Builder Capacity*

Click here to view code image

```
public class StringBuilderCapacity {
  public static void main(String[] args) {
    StringBuilder builder = new StringBuilder("No strings attached!");
    System.out.println("Builder contents: " + builder);
    System.out.println("Builder length:   " + builder.length());
    System.out.println("Builder capacity: " + builder.capacity());
    System.out.println("Ensure capacity of 40");
    builder.ensureCapacity(40);
    System.out.println("Builder capacity: " + builder.capacity());

    System.out.println("Trim to size");
    builder.trimToSize();
    System.out.println("Builder length:   " + builder.length());
    System.out.println("Builder capacity: " + builder.capacity());

    System.out.println("Set length to 10");
    builder.setLength(10);
    System.out.println("Builder length:   " + builder.length());
    System.out.println("Builder contents: " + builder);
    System.out.println("Set length to 0");
    builder.setLength(0);
    System.out.println("Builder is empty: " + (builder.length() == 0));
  }
}
```

Probable output from the program:

Click here to view code image

```
Builder contents: No strings attached!
Builder length:   20
Builder capacity: 36
Ensure capacity of 40
```

```
Builder capacity: 74
Trim to size
Builder length:   20
Builder capacity: 20
Set length to 10
Builder length:   10
Builder contents: No strings
Set length to 0
Builder is empty: true
```

Review Questions

8.5 Which expression will extract the substring `"kap"`, given the following declaration?

```
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)`

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

Click here to view code image

```
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.

b. The program will print `str3str1str2` at runtime.

c. The program will print `str3` at runtime.

d. The program will print `str3str1` at runtime.

e. The program will print `str3str2` at runtime.

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

Click here to view code image

```
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 program will fail to compile.

b. The program will print `false false` at runtime.

c. The program will print `false true` at runtime.

d. The program will print `true false` at runtime.

e. The program will print `true true` at runtime.

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

Click here to view code image

```
public class Uppity {
  public static void main(String[] args) {
    String str1 = "lower", str2 = "LOWER", str3 = "UPPER";
    str1.toUpperCase();
    str1.replace("LOWER","UPPER");
    System.out.println((str1.equals(str2)) + " " + (str1.equals(str3)));
  }
}
```

Select the one correct answer.

a. The program will print `false true`.

b. The program will print `false false`.

c. The program will print `true false`.

d. The program will print `true true`.

e. The program will fail to compile.

f. The program will compile, but it will throw an exception at runtime.

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

Click here to view code image

```
public class FunCharSeq {
  private static void putO(String s1) {
    s1 = s1.trim();
    s1 += "O";
  }

  public static void main(String[] args) {
    String s1 = " W ";
    putO(s1);
    s1.concat("W");
    System.out.println("|" + s1 + "|");
  }
}
```

Select the one correct answer.

a. `|WOW|`

b. `| W W|`

c. `|WO|`

d. `| W |`

e. The program will fail to compile.

f. The program will compile, but it will throw an exception at runtime.

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

Click here to view code image

```
public class Uppity {
  public static void main(String[] args) {
    String str1 = "lower", str2 = "LOWER", str3 = "UPPER";
    str1.toUpperCase();
    str1.replace("LOWER","UPPER");
    System.out.println((str1.equals(str2)) + " " + (str1.equals(str3)));
  }
}
```

Select the one correct answer.

a. The program will print `false true`.

b. The program will print `false false`.

c. The program will print `true false`.

d. The program will print `true true`.

e. The program will fail to compile.

f. The program will compile, but it will throw an exception at runtime.

8.11 What will the following code print when run?

Click here to view code image

```
String s = "This is hard";
s = "-" + s.substring(s.indexOf(' '), s.indexOf(' ', s.indexOf(' ') + 1) + 1)
```

```
    .strip() + "-";
System.out.println(s);
```

Select the one correct answer.

a. `- is -`

b. `-is-`

c. `-is -`

d. `- is-`

e. `-s is h-`

f. `-sish-`

8.12 Given the following code:

Click here to view code image

```
String txt = """
  a
    b
  c
""";
int from = 0;
int to = txt.indexOf('\n');
String line = null;
while(to < txt.length()-1) {
  to = txt.indexOf('\n', from);
  line = txt.substring(from, to);
  System.out.print(line.length());
  from = to+1;
}
```

The first line of the text block has two leading spaces, the second line has four leading spaces, and the third line has two leading spaces. There are no leading spaces on the line with the closing delimiter of the text block.

What is the result?

Select the one correct answer.

a. `353`

b. `3530`

c. `1510`

d. `131`

8.13 Which text block produces a single line of text with the following exact characters when printed?

```
"a""b"
```

Select the one correct answer.

a. `String txt = """"a""b\"""";`

b.

```
String txt = """"a""b"
""";
```

c.

```
String txt = """
"a""b"""";
```

d.

```
String txt = """
"a""b\"""";
```

e.

```
String txt = """
"a""b"
""";
```

f.

```
String txt = """
"a\""b"\""";
```

8.14 Which of the following statements are true about text blocks? Select the two correct answers.

a. Content of a text block starts immediately after the line terminator of the opening delimiter.

b. Content of a text block ends immediately after the line terminator of the closing delimiter.

c. A text block is a subtype of the `String` class.

d. All leading whitespace is removed from each line in the text block.

e. All trailing whitespace is removed from each line in the text block.

f. Incidental whitespace is not removed from each line in the text block.

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

Click here to view code image

```
public class MyClass {
  public static void main(String[] args) {
    String str = "hello";
    StringBuilder sb = new StringBuilder(str);
    sb.reverse();
    if (str == sb) System.out.println("a");
```

```
      if (str.equals(sb)) System.out.println("b");
      if (sb.equals(str)) System.out.println("c");
    }
  }
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile, but it will throw an exception at runtime.

c. The program will compile, but it will not print anything.

d. The program will compile, and will print `abc`.

e. The program will compile, and will print `bc`.

f. The program will compile, and will print `a`.

g. The program will compile, and will print `b`.

h. The program will compile, and will print `c`.

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

Click here to view code image

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

Select the one correct answer.

a. The code will fail to compile because there is no method named `setLength` in the `StringBuilder` class.

b. The code will fail to compile because the `StringBuilder` reference `sb` is not a legal argument to the `println()` method.

c. The program will throw a `StringIndexOutOfBoundsException` at runtime.

d. The program will print `Have a nice day` at runtime.

e. The program will print `Have a` at runtime.

f. The program will print `ce day` at runtime.

8.17 Which statement is true about the following code, where the argument string `"  1234  "` has two leading and two trailing spaces?

Click here to view code image

```
  public class RQ_8_24 {
    public static void main(String[] args) {
```

```
    StringBuilder sb = new StringBuilder("  1234  ");
    sb.trimToSize();
    sb.append("!");
    sb.reverse();
    sb.setLength(5);
    System.out.println("|" + sb + "|");
  }
}
```

Select the one correct answer.

a. The program will print `|4321!|`.

b. The program will print `|!1234|`.

c. The program will print `|! 43|`.

d. The program will print `|1234!|`.

e. The program will print `|!4321|`.

f. The program will fail to compile.

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

Click here to view code image

```
public class PeskyCharSeq {
  public static void main (String[] args) {
    StringBuilder sb1 = new StringBuilder("WOW");
    StringBuilder sb2 = new StringBuilder(sb1);
    System.out.println((sb1==sb2) + " " + sb1.equals(sb2));
  }
}
```

Select the one correct answer.

a. The program will print `false true`.

b. The program will print `false false`.

c. The program will print `true false`.

d. The program will print `true true`.

e. The program will fail to compile.

f. The program will compile, but it will throw an exception at runtime.

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

Click here to view code image

```
public class MoreCharSeq {
  public static void main (String[] args) {
    String s1 = "WOW";
    StringBuilder s2 = new StringBuilder(s1);
    String s3 = new String(s2);
    System.out.println((s1.hashCode() == s2.hashCode()) + " " +
                       (s1.hashCode() == s3.hashCode()));
  }
}
```

Select the one correct answer.

a. The program will print `false true`.

b. The program will print `false false`.

c. The program will print `true false`.

d. The program will print `true true`.

e. The program will fail to compile.

f. The program will compile, but it will throw an exception at runtime.

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

Click here to view code image

```
public class Appendage {
  private static void putO(StringBuilder s1) {
    s1.append("O");
  }

  public static void main(String[] args) {
    StringBuilder s1 = new StringBuilder("W");
    putO(s1);
    s1.append("W!");
    System.out.println(s1);
  }
}
```

Select the one correct answer.

a. The program will print `WW!`.

b. The program will print `WOW!`.

c. The program will print `W`.

d. The program will print `WO`.

e. The program will fail to compile.

f. The program will compile, but it will throw an exception at runtime.

8.21 What will the following code print when run?

Click here to view code image

```
StringBuilder text = new StringBuilder();
text.append("12");
text.insert(1, "34");
text.delete(1, 1);
text.replace(0, 1, "");
System.out.println(text);
```

Select the one correct answer.

a. `2`

b. `12`

c. `32`

d. `34`

e. `42`

f. `134`

g. `234`

h. `312`

i. `342`

j. `412`

8.22 What will the following code print when run?

Click here to view code image

```
StringBuilder text = new StringBuilder();
text.append("42");
text.delete(1,2);
System.out.println(text.toString() + (text.capacity() + text.length()));
```

Select the one correct answer.

a. `416`

b. `417`

c. `4161`

d. `4171`

e. `410`

f. `411`

g. `4101`

h. `4111`

8.6 The `Math` Class

The `final` class `java.lang.Math` defines a set of `static` methods to support common mathematical functions, including functions for rounding numbers, finding the maximum and minimum of two numbers, calculating logarithms and exponentiation, performing exact arithmetic, generating pseudorandom numbers, and much more. The `Math` class is a utility class and cannot be instantiated.

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

Click here to view code image

```
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.

Click here to view code image

```
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.

Click here to view code image

```
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:

Click here to view code image

```
long   ll = Math.abs(2022L);             // 2022L
double dd = Math.abs(-Math.PI);          // 3.141592653589793

double d1 = Math.min(Math.PI, Math.E);   // 2.718281828459045
long   m1 = Math.max(1984L, 2022L);      // 2022L
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` in order to assign it to an `int` variable.

Click here to view code image

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

Returns the *smallest* (*closest to negative infinity*) `double` value that is *greater than or equal to* the argument `d`, and is equal to a mathematical integer.

Click here to view code image

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

Returns the *largest* (*closest to positive infinity*) `double` value that is *less than or equal to* the argument `d`, and is equal to a mathematical integer. Note that `Math.ceil(d)` is exactly the value of `-Math.floor(-d)`.

Click here to view code image

```
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 8.3**, **p. 480**.

Click here to view code image

```
double upPI    = Math.ceil(3.14);              // 4.0
double downPI  = Math.floor(3.14);             // 3.0
long   roundPI = Math.round(3.14);             // 3L

double upNegPI    = Math.ceil(-3.14);          // -3.0
double downNegPI  = Math.floor(-3.14);         // -4.0
long   roundNegPI = Math.round(-3.14);         // -3L
```

Table 8.3 *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

Click here to view code image

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

Returns the value of `d1` raised to the power of `d2` (i.e., `d1` ^{d2}).

Click here to view code image

```
static double exp(double d)
```

Returns the exponential number *e* raised to the power of `d` (i.e., e^{d}).

Click here to view code image

```
static double log(double d)
```

Returns the natural logarithm (base *e*) of `d` (i.e., log_e `d`).

Click here to view code image

```
static double sqrt(double d)
```

Returns the square root of `d` (i.e., `d` $^{0.5}$). For a `NaN` or a negative argument, the result is a `NaN`.

Some examples of exponential functions:

Click here to view code image

```
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(25.0);               // 5.0
```

Exact Integer Arithmetic Functions

Integer arithmetic operators (like `+`, `-`, `*`) do not report *overflow errors*—that is, the integer values *wrap around* (§2.8, p. 59). However, if it is important to detect overflow errors, the `Math` class provides methods that report overflow errors by throwing an `ArithmeticException` when performing integer arithmetic operations. The relevant methods have the postfix `"Exact"` in their name, and are shown below.

The code snippets below illustrate exact integer arithmetic operations provided by the `Math` class, where we assume the field `System.out` is statically imported. The results computed at (1a), (2a), and (3a) are incorrect due to overflow, but the standard arithmetic operators do not report overflow errors. (1b), (2b), and (3b) use exact arithmetic operations and throw an `ArithmeticException` when an overflow is detected. (1c), (2c), and (3c) compute correct results as there is no overflow in performing the operation.

Click here to view code image

```
out.println(Integer.MAX_VALUE + 1);                  // (1a) -2147483648
out.println(Math.addExact(Integer.MAX_VALUE, 1));    // (1b) ArithmeticException
out.println(Math.addExact(1_000_000, 1_000));        // (1c) 1001000
out.println(Integer.MAX_VALUE * 100);                // (2a) -100
out.println(
     Math.multiplyExact(Integer.MAX_VALUE, 100)      // (2b) ArithmeticException
);
out.println(Math.multiplyExact(1_000_000, 1_000));   // (2c) 1000000000

out.println((int)Long.MAX_VALUE);                    // (3a) -1
out.println(Math.toIntExact(Long.MAX_VALUE));        // (3b) ArithmeticException
out.println(Math.toIntExact(1_000_000));             // (3c) 1000000
```

Selected exact integer arithmetic methods in the `Math` class are shown below.

Click here to view code image

```
static int absExact(int a)
static long absExact(long a)
```

Return the mathematical absolute value of an `int` or a `long` value if it is exactly representable as an `int` or a `long`, throwing `ArithmeticException` if the argument is `Integer.MIN_VALUE` or `Long.MIN_VALUE`, as these argument values would overflow the positive `int` or `long` range, respectively.

Click here to view code image

```
static int  addExact(int x, int y)
static long addExact(long x, long y)
static int  subtractExact(int x, int y)
static long subtractExact(long x, long y)
```

Return the sum or difference of the given arguments, throwing `Arithmetic-Exception` if the result overflows an `int` or a `long`, respectively.

Click here to view code image

```
static int  multiplyExact(int x, int y)
static long multiplyExact(long x, int y)
static long multiplyExact(long x, long y)
```

Return the product of the arguments, throwing `ArithmeticException` if the result overflows an `int` in the first method or a `long` in the last two methods.

Click here to view code image

```
static int  incrementExact(int a)
static long incrementExact(long a)
static int  decrementExact(int a)
static long decrementExact(long a)
```

Return the argument incremented or decremented by 1, throwing `Arithmetic-Exception` if the result overflows an `int` or a `long`, respectively.

Click here to view code image

```
static int  negateExact(int a)
static long negateExact(long a)
```

Return the negation of the argument, throwing `ArithmeticException` if the result overflows an `int` or a `long`, respectively.

Click here to view code image

```
static int toIntExact(long value)
```

Returns the value of the `long` argument as an `int`, throwing `ArithmeticException` if the value overflows an `int`.

Pseudorandom Number Generator

The class `Math` provides the `random()` method for generating pseudorandom numbers of type `double` that are in the open interval `[0.0, 1.0)`. The `Random` class (p. 482) should also be considered, as it is more versatile and easier to use.

```
static double 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.

We can simulate a dice roll as follows:

Click here to view code image

```
int diceValue = 1 + (int)(Math.random() * 6.0);  // A dice roll in range [1 .. 6]
```

The dice value will always be in the interval `[1, 6]`, depending on the pseudorandom number returned by the `Math.random()` method, as we can see from the sample dice rolls below.

Click here to view code image

```
Math.random():    {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9}
Multiply by 6:    {0.0, 0.6, 1.2, 1.8, 2.4, 3.0, 3,6, 4.2, 4.8, 5.4}
Convert to int:   {0,   0,   1,   1,   2,   3,   3,   4,   4,   5}
Add offset 1:     {1,   1,   2,   2,   3,   4,   4,   5,   5,   6}
```

8.7 The `Random` Class

In computer games and simulations we often need to generate a sequence of random numbers. For a dice game it is necessary to simulate rolling the dice—that is, generating a dice roll value between 1 and 6. Ideally, the probability of a given value is the same (one-sixth) for each value on a die. The numbers generated this way are called *random numbers*. To generate random numbers with the help of a program is not possible, as a program can only compute values, not pick them randomly. To guarantee equal probability for all cases is very difficult, so we have to settle for *pseudorandom numbers*—that is, a sequence of numbers that closely approximates a sequence of random numbers. A lot of research has gone into finding mathematical formulae that compute good approximations to random numbers.

The `java.util.Random` class implements the `java.util.random.RandomGenerator` interface that defines the common protocol for pseudorandom number generators (PRNGs) in Java.

Generating *numeric streams of pseudorandom values* using the `Random` class is covered with the discussion on creating streams (§16.4, p. 900).

The following constructors can be used to create a new random number generator, optionally specifying a *seed* value (p. 484).

```
Random()
Random(long seed)
```

The `Random` class implements PRNGs for different primitive data types. The appropriate *next* method can be called on a `Random` object to obtain the next pseudorandom value.

Click here to view code image

```
int     nextInt()
int     nextInt(int bound)
long    nextLong()
float   nextFloat()
double  nextDouble()
boolean nextBoolean()
```

Return the next pseudorandom, uniformly distributed value that is either `int`, `int` between `0` and `bound` (exclusive), `long`, `float` between `0.0f` and `1.0f` (exclusive), `double` between `0.0d` and `1.0d` (exclusive), or `true`/`false`, respectively.

Here we will concentrate on a pseudorandom generator for `int` values, but first we need to create an object of the `Random` class:

Click here to view code image

```
Random generator = new Random();
```

We can then call the method `nextInt()` repeatedly on this object every time we need a new random `int` value:

Click here to view code image

```
int number = generator.nextInt();
```

Each call to the method will return a random integer in the interval [-2^{31}, $2^{31} - 1$], which is the range of the `int` data type.

Determining the Range

We are often interested in generating random numbers in a particular range. For example, the following code will return a random number in the interval `[0, 10]`:

Click here to view code image

```
number = generator.nextInt(11);      // Random integer in the interval [0, 10]
```

If the bound value is `n`, the integer returned is in the interval `[0, n-1]`. By supplying a new bound value to the `nextInt()` method, we can change the upper bound of the original interval.

If we want to shift the interval, we can add (or subtract) an offset from the value returned by the `nextInt()` method. In the code below, values generated in the original interval `[0, 10]` will now lie in the interval `[2, 12]`—that is, the offset `2` maps the interval `[0, 10]` onto the interval `[2, 12]`:

Click here to view code image

```
number = 2 + generator.nextInt(11);  // Random integer in the interval [2, 12]
```

If we want the values in the interval to have a distance greater than 1, we can multiply the value generated by a distance value:

Click here to view code image

```
number = 2 + 3*generator.nextInt(5); // Random integer in the set {2, 5, 8, 11, 14}
```

With a distance value of `3`, the expression `3*generator.nextInt(5)` always returns a value in the set `{0, 3, 6, 9, 12}`, and an offset of `2` ensures that the variable `number` is assigned a value from the set `{2, 5, 8, 11, 14}`.

We can simulate a dice roll as follows:

Click here to view code image

```
int diceValue = 1 + generator.nextInt(6);  // A dice value in the interval [1, 6]
```

The expression `generator.nextInt(6)` always returns a value in the set `{0, 1, 2, 3, 4, 5}`, and an offset of `1` ensures that the variable `diceValue` is assigned a value from the set `{1, 2, 3, 4, 5, 6}`.

Generating the Same Sequence of Pseudorandom Numbers

The way in which we have used the pseudorandom number generator up to now cannot guarantee the same sequence of pseudorandom numbers each time the program is run. This is because the pseudorandom number generator is based on the time of the system clock. This is obviously different each time the program is run. If we want to generate the same sequence of pseudorandom numbers each time the program is run, we can specify a *seed* in the call to the `Random` constructor:

Click here to view code image

```
Random persistentGenerator = new Random(31);
```

In the declaration above, the seed is the prime number 31. The seed is usually a prime number (i.e., a number that is only divisible by itself or one), as such numbers are highly suitable for implementing good, viable pseudorandom number generators.

8.8 Using Big Numbers

There is a need for greater precision in arithmetic calculations than the precision offered by arithmetic operators performing calculations on finite-precision values of numeric primitive types. This is especially true for calculations involving financial or monetary values, where accumulation of rounding errors and precision is paramount. The classes `BigInteger` and `BigDecimal` in the `java.math` package support arbitrary-precision integers and arbitrary-precision decimal numbers, respectively, and provide methods for arbitrary-precision computations (**Table 8.4**). Both classes extend the `java.lang.Number` abstract class (**Figure 8.1**).

Table 8.4 *Big Number Classes*

Big number classes in the `java.math` package	Description
`BigDecimal extends Number`	A class that represents immutable, arbitrary-precision, signed decimal numbers
`BigInteger extends Number`	A class that represents immutable, arbitrary-precision signed integers

A `BigDecimal` is represented as an arbitrary-precision *unscaled integer value* and a 32-bit *integer scale*. For a non-negative unscaled value, the scale represents the number of digits to the right of the decimal point. For a negative unscaled value, the number represented is (`unscaled value * 10`$^{-\text{scale}}$). For example:

Click here to view code image

```
The BigDecimal  3.14   has unscaled value =  314   and scale = 2.
The BigDecimal -3.1415 has unscaled value = -31415 and scale = 4.
```

The `BigDecimal` class can represent very large and very small decimal numbers with very high precision. The class provides methods that allow control over scale and rounding behavior for arithmetic operations. In addition, the class provides support for converting to different representations and comparing `BigDecimal` values.

Formatting, parsing, and rounding of `BigDecimal` values is covered with formatting of numbers and currency (§18.5, p. 1116).

A `BigInteger` represents an arbitrary-precision signed integer value in base 10 and two's-complement notation, which is internally implemented by an arbitrary-size `int` array. The `BigInteger` class has methods analagous to the `BigDecimal` class for arithmetic calculations, conversions, and comparison.

Here we will only provide a few examples of using the `BigDecimal` class. The API for the `BigDecimal` and `BigInteger` classes in the `java.math` package should be consulted, both for details on features presented here and additional features provided for high-precision manipulation of big numbers.

`BigDecimal` Constants

The class `BigDecimal` provides the following ready-made constants corresponding to the decimal values 0, 1, and 10, that are represented with a scale of 0:

```
BigDecimal.ZERO
BigDecimal.ONE
BigDecimal.TEN
```

Constructing `BigDecimal` Numbers

Selected constructors of the `BigDecimal` class are shown below.

```
BigDecimal(int value)
BigDecimal(long value)
BigDecimal(double value)
```

Create a `BigDecimal` with the decimal number representation of the specified value. Note that these constructors can result in a loss of precision if the specified numerical value does not have an exact representation.

Click here to view code image

```
BigDecimal(String strValue)
```

Creates a `BigDecimal` from the string representation of a numerical value. Recommended as the preferred way of creating `BigDecimal` numbers.

The following `valueOf()` methods also create `BigDecimal` numbers:

Click here to view code image

```
static BigDecimal valueOf(long value)
static BigDecimal valueOf(double value)
```

Create a `BigDecimal` from the specified numerical `value`. The second method uses the string representation of the `double value` obtained from the `Double.toString(double)` method to create a `BigDecimal`.

The following code creates `BigDecimal` values and also shows the string representation of the `BigDecimal` numbers created. It shows that the constructor `BigDecimal(String)` or the static `valueOf()` methods are preferable when creating `BigDecimal` values, especially for decimal values that cannot be represented as exact `double` values.

Click here to view code image

```
BigDecimal dTobd   = new BigDecimal(0.7);        // 0.69999999999999995559 10...
BigDecimal strTobd = new BigDecimal("0.7");      // 0.7
BigDecimal valTobd = BigDecimal.valueOf(0.7);    // 0.7
```

Computing with `BigDecimal` Numbers

Selected methods that perform common arithmetic operations with `BigDecimal` numbers are given below.

The limitations of arithmetic operators on decimal values are shown by the code below. An incorrect result may be computed if the representation of a decimal value is inexact in the format of the `double` primitive type.

Click here to view code image

```
double d1 = 0.70;
double d2 = 0.10;
System.out.println(d1 + d2);

Output: 0.7999999999999999
```

However, using `BigDecimal` values gives the correct result:

Click here to view code image

```
BigDecimal bd1b = new BigDecimal("0.70");
BigDecimal bd2b = new BigDecimal("0.10");
System.out.println(bd1b.add(bd2b));

Output: 0.80
```

Finally, here is a simple arithmetic calculation using `BigDecimal` numbers to compute the total cost of 10^3 items, priced at $2.99 each and 25% sales tax.

Click here to view code image

```
BigDecimal price     = new BigDecimal("2.99");
BigDecimal tax       = new BigDecimal("0.25");
BigDecimal quantity  = BigDecimal.TEN.pow(3);
BigDecimal totalCost = price.add(price.multiply(tax)).multiply(quantity);
```

```
System.out.println(totalCost);

Output:  3737.5000
```

It is worth keeping in mind that `BigDecimal` numbers are immutable, and not to fall into the pitfall shown in the code below:

Click here to view code image

```
BigDecimal sum = BigDecimal.ZERO;
sum.add(BigDecimal.ONE);           // sum is not updated! Returns new BigDecimal.
```

Selected methods for computing with `BigDecimal` numbers:

Click here to view code image

```
BigDecimal add(BigDecimal val)
BigDecimal subtract(BigDecimal val)
BigDecimal multiply(BigDecimal val)
BigDecimal divide(BigDecimal val)
BigDecimal remainder(BigDecimal val)
```

Return a `BigDecimal` whose value represents the result of performing the operation `(this + val)`, `(this - val)`, `(this * val)`, `(this / val)`, or `(this % val)`, respectively.

```
BigDecimal abs()
```

Returns a `BigDecimal` whose value is the absolute value of this `BigDecimal`.

```
BigDecimal negate()
```

Returns a `BigDecimal` whose value is `(-this)`.

```
BigDecimal pow(int n)
```

Returns a `BigDecimal` whose value is `(this`^{n}`)`.

Review Questions

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

Click here to view code image

```
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.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 at (1).

b. The line at (2).

c. The line at (3).

d. The line at (4).

e. The line at (5).

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

Click here to view code image

```
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

8.25 Which of the following 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.