Chapter 6

Exploring the Basic APIs Part 1

Aspiring Android developers need to acquire a solid understanding of foundational Java APIs. You have already encountered a few of these APIs, such as the `Object` and `String` classes and the `Throwable` class hierarchy. This chapter introduces you to additional language-oriented (basic) APIs pertaining to math, packages, primitive types, and the garbage collector.

> **NOTE:** Chapter 6 explores basic API classes and interfaces that are located in the `java.lang`, `java.lang.ref`, and `java.math` packages.

Math APIs

Chapter 2 presented Java's `+`, `-`, `*`, `/`, and `%` operators for performing basic arithmetic on primitive type values. Java also provides classes for performing trigonometry and other advanced math operations, representing monetary values accurately, and supporting extremely long integers for use in RSA encryption (`http://en.wikipedia.org/wiki/RSA`) and other contexts.

Math and StrictMath

The `java.lang.Math` class declares `double` constants `E` and `PI` that represent the natural logarithm base value (2.71828...) and the ratio of a circle's circumference to its diameter (3.14159...). `E` is initialized to `2.718281828459045` and `PI` is initialized to `3.141592653589793`. `Math` also declares assorted class methods to perform various math operations. Table 6–1 describes many of these methods.

Table 6–1. *Math Methods*

Method	Description
`double abs(double d)`	Return the absolute value of `d`. There are four special cases: `abs(-0.0) = +0.0`, `abs(+infinity) = +infinity`, `abs(-infinity) = +infinity`, and `abs(NaN) = NaN`.
`float abs(float f)`	Return the absolute value of `f`. There are four special cases: `abs(-0.0) = +0.0`, `abs(+infinity) = +infinity`, `abs(-infinity) = +infinity`, and `abs(NaN) = NaN`.
`int abs(int i)`	Return the absolute value of `i`. There is one special case: the absolute value of `Integer.MIN_VALUE` is `Integer.MIN_VALUE`.
`long abs(long l)`	Return the absolute value of `l`. There is one special case: the absolute value of `Long.MIN_VALUE` is `Long.MIN_VALUE`.
`double acos(double d)`	Return angle `d`'s arc cosine within the range 0 through PI. There are three special cases: `acos(anything > 1) = NaN`, `acos(anything < -1) = NaN`, and `acos(NaN) = NaN`.
`double asin(double d)`	Return angle `d`'s arc sine within the range -PI/2 through PI/2. There are three special cases: `asin(anything > 1) = NaN`, `asin(anything < -1) = NaN`, and `asin(NaN) = NaN`.
`double atan(double d)`	Return angle `d`'s arc tangent within the range -PI/2 through PI/2. There are five special cases: `atan(+0.0) = +0.0`, `atan(-0.0) = -0.0`, `atan(+infinity) = +PI/2`, `atan(-infinity) = -PI/2`, and `atan(NaN) = NaN`.
`double ceil(double d)`	Return the smallest value (closest to negative infinity) that is not less than `d` and is equal to an integer. There are six special cases: `ceil(+0.0) = +0.0`, `ceil(-0.0) = -0.0`, `ceil(anything > -1.0 and < 0.0) = -0.0`, `ceil(+infinity) = +infinity`, `ceil(-infinity) = -infinity`, and `ceil(NaN) = NaN`.
`double cos(double d)`	Return the cosine of angle `d` (expressed in radians). There are three special cases: `cos(+infinity) = NaN`, `cos(-infinity) = NaN`, and `cos(NaN) = NaN`.
`double exp(double d)`	Return Euler's number e raised to the power `d`. There are three special cases: `exp(+infinity) = +infinity`, `exp(-infinity) = +0.0`, and `exp(NaN) = NaN`.
`double floor(double d)`	Return the largest value (closest to positive infinity) that is not greater than `d` and is equal to an integer. There are five special cases: `floor(+0.0) = +0.0`, `floor(-0.0) = -0.0`, `floor(+infinity) = +infinity`, `floor(-infinity) = -infinity`, and `floor(NaN) = NaN`.

Method	Description
`double log(double d)`	Return the natural logarithm (base e) of `d`. There are six special cases: `log(+0.0) = -infinity`, `log(-0.0) = -infinity`, `log(anything < 0) = NaN`, `log(+infinity) = +infinity`, `log(-infinity) = NaN`, and `log(NaN) = NaN`.
`double log10(double d)`	Return the base 10 logarithm of `d`. There are six special cases: `log10(+0.0) = -infinity`, `log10(-0.0) = -infinity`, `log10(anything < 0) = NaN`, `log10(+infinity) = +infinity`, `log10(-infinity) = NaN`, and `log10(NaN) = NaN`.
`double max(double d1, double d2)`	Return the most positive (closest to positive infinity) of `d1` and `d2`. There are four special cases: `max(NaN, anything) = NaN`, `max(anything, NaN) = NaN`, `max(+0.0, -0.0) = +0.0`, and `max(-0.0, +0.0) = +0.0`.
`float max(double f1, double f2)`	Return the most positive (closest to positive infinity) of `f1` and `f2`. There are four special cases: `max(NaN, anything) = NaN`, `max(anything, NaN) = NaN`, `max(+0.0, -0.0) = +0.0`, and `max(-0.0, +0.0) = +0.0`.
`int max(int i1, int i2)`	Return the most positive (closest to positive infinity) of `i1` and `i2`.
`long max(long l1, long l2)`	Return the most positive (closest to positive infinity) of `l1` and `l2`.
`double min(double d1, double d2)`	Return the most negative (closest to negative infinity) of `d1` and `d2`. There are four special cases: `min(NaN, anything) = NaN`, `min(anything, NaN) = NaN`, `min(+0.0, -0.0) = -0.0`, and `min(-0.0, +0.0) = -0.0`.
`float min(float f1, float f2)`	Return the most negative (closest to negative infinity) of `f1` and `f2`. There are four special cases: `min(NaN, anything) = NaN`, `min(anything, NaN) = NaN`, `min(+0.0, -0.0) = -0.0`, and `min(-0.0, +0.0) = -0.0`.
`int min(int i1, int i2)`	Return the most negative (closest to negative infinity) of `i1` and `i2`.
`long min(long l1, long l2)`	Return the most negative (closest to negative infinity) of `l1` and `l2`.
`double random()`	Return a pseudorandom number between 0.0 (inclusive) and 1.0 (exclusive).
`long round(double d)`	Return the result of rounding `d` to a long integer. The result is equivalent to `(long) Math.floor(d+0.5)`. There are seven special cases: `round(+0.0) = +0.0`, `round(-0.0) = +0.0`, `round(anything > Long.MAX_VALUE) = Long.MAX_VALUE`, `round(anything < Long.MIN_VALUE) = Long.MIN_VALUE`,

Method	Description
	`round(+infinity) = Long.MAX_VALUE`, `round(-infinity) = Long.MIN_VALUE`, and `round(NaN) = +0.0`.
`int round(float f)`	Return the result of rounding `f` to an integer. The result is equivalent to `(int) Math.floor(f+0.5)`. There are seven special cases: `round(+0.0) = +0.0`, `round(-0.0) = +0.0`, `round(anything > Integer.MAX_VALUE) = Integer.MAX_VALUE`, `round(anything < Integer.MIN_VALUE) = Integer.MIN_VALUE`, `round(+infinity) = Integer.MAX_VALUE`, `round(-infinity) = Integer.MIN_VALUE`, and `round(NaN) = +0.0`.
`double signum(double d)`	Return the sign of `d` as -1.0 (`d` less than 0.0), 0.0 (`d` equals 0.0), and 1.0 (`d` greater than 0.0). There are five special cases: `signum(+0.0) = +0.0`, `signum(-0.0) = -0.0`, `signum(+infinity) = +1.0`, `signum(-infinity) = -1.0`, and `signum(NaN) = NaN`.
`float signum(float f)`	Return the sign of `f` as -1.0 (`f` less than 0.0), 0.0 (`f` equals 0.0), and 1.0 (`f` greater than 0.0). There are five special cases: `signum(+0.0) = +0.0`, `signum(-0.0) = -0.0`, `signum(+infinity) = +1.0`, `signum(-infinity) = -1.0`, and `signum(NaN) = NaN`.
`double sin(double d)`	Return the sine of angle `d` (expressed in radians). There are five special cases: `sin(+0.0) = +0.0`, `sin(-0.0) = -0.0`, `sin(+infinity) = NaN`, `sin(-infinity) = NaN`, and `sin(NaN) = NaN`.
`double sqrt(double d)`	Return the square root of `d`. There are five special cases: `sqrt(+0.0) = +0.0`, `sqrt(-0.0) = -0.0`, `sqrt(anything < 0) = NaN`, `sqrt(+infinity) = +infinity`, and `sqrt(NaN) = NaN`.
`double tan(double d)`	Return the tangent of angle `d` (expressed in radians). There are five special cases: `tan(+0.0) = +0.0`, `tan(-0.0) = -0.0`, `tan(+infinity) = NaN`, `tan(-infinity) = NaN`, and `tan(NaN) = NaN`.
`double toDegrees (double angrad)`	Convert angle `angrad` from radians to degrees via expression `angrad*180/PI`. There are five special cases: `toDegrees(+0.0) = +0.0`, `toDegrees(-0.0) = -0.0`, `toDegrees(+infinity) = +infinity`, `toDegrees(-infinity) = -infinity`, and `toDegrees(NaN) = NaN`.
`double toRadians (angdeg)`	Convert angle `angdeg` from degrees to radians via expression `angdeg/180*PI`. There are five special cases: `toRadians(+0.0) = +0.0`, `toRadians(-0.0) = -0.0`, `toRadians(+infinity) = +infinity`, `toRadians(-infinity) = -infinity`, and `toRadians(NaN) = NaN`.

Table 6–1 reveals a wide variety of useful math-oriented methods. For example, each `abs()` method returns its argument's *absolute value* (number without regard for sign).

`abs(double)` and `abs(float)` are useful for comparing double precision floating-point and floating-point values safely. For example, `0.3 == 0.1+0.1+0.1` evaluates to false because 0.1 has no exact representation. However, you can compare these expressions with `abs()` and a tolerance value, which indicates an acceptable range of error. For example, `Math.abs(0.3-(0.1+0.1+0.1)) < 0.1` returns true because the absolute difference between `0.3` and `0.1+0.1+0.1` is less than a 0.1 tolerance value.

Previous chapters demonstrated other `Math` methods. For example, Chapter 2 demonstrated `Math`'s `sin()`, `toRadians()`, `cos()`, `round(double)`, and `random()` methods.

As Chapter 5's `Lotto649` application revealed, `random()` (which returns a number that appears to be randomly chosen but is actually chosen by a predictable math calculation, and hence is *pseudorandom*) is useful in simulations, games, and wherever an element of chance is needed, but first its return value (0.0 to almost 1.0) must somehow be transformed into a more useful range, perhaps 0 through 49, or maybe -100 through 100. You will find Listing 6–1's `rnd()` method useful for making these transformations.

Listing 6–1. ***Converting `random()`'s return value into something more useful***

```
public static int rnd(int limit)
{
   return (int) (Math.random()*limit);
}
```

`rnd()` transforms `random()`'s 0.0 to almost 1.0 double precision floating-point range to a 0 through `limit - 1` integer range. For example, `rnd(50)` returns an integer ranging from 0 through 49. Also, `-100+rnd(201)` transforms 0.0 to almost 1.0 into -100 through 100 by adding a suitable offset and passing an appropriate `limit` value.

> **CAUTION:** Do not specify `(int) Math.random()*limit` because this expression always evaluates to 0. The expression first casts `random()`'s double precision floating-point fractional value (0.0 through 0.99999. . .) to integer 0 by truncating the fractional part, and then multiplies 0 by `limit`, which results in 0.

Table 6–1 also reveals some curiosities beginning with +infinity, -infinity, +0.0, -0.0, and NaN (Not a Number).

Java's floating-point calculations are capable of returning +infinity, -infinity, +0.0, -0.0, and NaN because Java largely conforms to IEEE 754 (`http://en.wikipedia.org/wiki/IEEE_754`), a standard for floating-point calculations. The following are the circumstances under which these special values arise:

- +infinity returns from attempting to divide a positive number by 0.0. For example, `System.out.println(1.0/0.0);` outputs `Infinity`.
- -infinity returns from attempting to divide a negative number by 0.0. For example, `System.out.println(-1.0/0.0);` outputs `-Infinity`.

- NaN returns from attempting to divide 0.0 by 0.0, attempting to calculate the square root of a negative number, and attempting other strange operations. For example, `System.out.println(0.0/0.0);` and `System.out.println(Math.sqrt(-1.0));` each output `NaN`.
- +0.0 results from attempting to divide a positive number by +infinity. For example, `System.out.println(1.0/(1.0/0.0));` outputs `+0.0`.
- -0.0 results from attempting to divide a negative number by +infinity. For example, `System.out.println(-1.0/(1.0/0.0));` outputs `-0.0`.

Once an operation yields +infinity, -infinity, or NaN, the rest of the expression usually equals that special value. For example, `System.out.println(1.0/0.0*20.0);` outputs `Infinity`. Also, an expression that first yields +infinity or -infinity might devolve into NaN. For example, `1.0/0.0*0.0` yields +infinity (`1.0/0.0`) and then NaN (+infinity*`0.0`).

Another curiosity is `Integer.MAX_VALUE`, `Integer.MIN_VALUE`, `Long.MAX_VALUE`, and `Long.MIN_VALUE`. Each of these items is a primitive wrapper class constant that identifies the maximum or minimum value that can be represented by the class's associated primitive type.

Finally, you might wonder why the `abs()`, `max()`, and `min()` overloaded methods do not include `byte` and `short` versions, as in `byte abs(byte b)` and `short abs(short s)`. There is no need for these methods because the limited ranges of bytes and short integers make them unsuitable in calculations. If you need such a method, check out Listing 6–2.

Listing 6–2. ***Overloaded*** *`byte abs(byte b)`* ***and*** *`short abs(short s)`* ***methods***

```
public static byte abs(byte b)
{
   return (b < 0) ? (byte) -b : b;
}
public static short abs(short s)
{
   return (s < 0) ? (short) -s : s;
}
public static void main(String[] args)
{
   byte b = -2;
   System.out.println(abs(b)); // Output: 2
   short s = -3;
   System.out.println(abs(s)); // Output: 3
}
```

The `(byte)` and `(short)` casts are necessary because `-b` converts `b`'s value from a `byte` to an `int`, and `-s` converts `s`'s value from a `short` to an `int`. In contrast, these casts are not needed with `(b < 0)` and `(s < 0)`, which automatically cast `b`'s and `s`'s values to an `int` before comparing them with `int`-based `0`.

TIP: Their absence from `Math` suggests that `byte` and `short` are not very useful in method declarations. However, these types are useful when declaring arrays whose elements store small values (such as a binary file's byte values). If you declared an array of `int` or `long` to store such values, you would end up wasting heap space (and might even run out of memory).

While searching through the Java documentation for the `java.lang` package, you will probably encounter a class named `StrictMath`. Apart from a longer name, this class appears to be identical to `Math`. The differences between these classes can be summed up as follows:

- `StrictMath`'s methods return exactly the same results on all platforms. In contrast, some of `Math`'s methods might return values that vary ever so slightly from platform to platform.
- Because `StrictMath` cannot utilize platform-specific features such as an extended-precision math coprocessor, an implementation of `StrictMath` might be less efficient than an implementation of `Math`.

For the most part, `Math`'s methods call their `StrictMath` counterparts. Two exceptions are `toDegrees()` and `toRadians()`. Although these methods have identical code bodies in both classes, `StrictMath`'s implementations include reserved word `strictfp` in the method headers:

```
public static strictfp double toDegrees(double angrad)
public static strictfp double toRadians(double angdeg)
```

Wikipedia's "strictfp" entry (`http://en.wikipedia.org/wiki/Strictfp`) mentions that `strictfp` restricts floating-point calculations to ensure portability. This reserved word accomplishes portability in the context of intermediate floating-point representations and overflows/underflows (generating a value too large or small to fit a representation).

NOTE: The previously cited "strictfp" article states that `Math` contains `public static strictfp double abs(double);` and other `strictfp` methods. If you check out this class's source code under Java version 6 update 16, you will not find `strictfp` anywhere in the source code. However, many `Math` methods (such as `sin()`) call their `StrictMath` counterparts, which are implemented in a platform-specific library, and the library's method implementations are strict.

Without `strictfp`, an intermediate calculation is not limited to the IEEE 754 32-bit and 64-bit floating-point representations that Java supports. Instead, the calculation can take advantage of a larger representation (perhaps 128 bits) on a platform that supports this representation.

An intermediate calculation that overflows/underflows when its value is represented in 32/64 bits might not overflow/underflow when its value is represented in more bits.

Because of this discrepancy, portability is compromised. `strictfp` levels the playing field by requiring all platforms to use 32/64 bits for intermediate calculations.

When applied to a method, `strictfp` ensures that all floating-point calculations performed in that method are in strict compliance. However, `strictfp` can be used in a class header declaration (as in `public strictfp class FourierTransform`) to ensure that all floating-point calculations performed in that class are strict.

> **NOTE:** `Math` and `StrictMath` are declared `final` so that they cannot be extended. Also, they declare private empty noargument constructors so that they cannot be instantiated.
>
> `Math` and `StrictMath` are examples of *utility classes* because they exist as placeholders for utility constants and utility (`static`) methods.

BigDecimal

In Chapter 2, I introduced a `CheckingAccount` class with a `balance` field. I declared this field to be of type `int`, and included a comment stating that `balance` represents the number of dollars that can be withdrawn. Alternatively, I could have stated that `balance` represents the number of pennies that can be withdrawn.

Perhaps you are wondering why I did not declare `balance` to be of type `double` or `float`. That way, `balance` could store values such as 18.26 (18 dollars in the whole number part and 26 pennies in the fraction part). I did not declare `balance` to be a `double` or `float` for the following reasons:

- Not all floating-point values that can represent monetary amounts (dollars and cents) can be stored exactly in memory. For example, 0.1 (which you might use to represent 10 cents), has no exact storage representation. If you executed `double total = 0.1; for (int i = 0; i < 50; i++) total += 0.1; System.out.println(total);`, you would observe `5.099999999999998` instead of the correct `5.1` as the output.
- The result of each floating-point calculation needs to be rounded to the nearest cent. Failure to do so introduces tiny errors that can cause the final result to differ from the correct result. Although `Math` supplies a pair of `round()` methods that you might consider using to round a calculation to the nearest cent, these methods round to the nearest integer (dollar).

Listing 6–3's `InvoiceCalc` application demonstrates both problems. However, the first problem is not serious because it contributes very little to the inaccuracy. The more serious problem occurs from failing to round to the nearest cent after performing a calculation.

Listing 6–3. *Floating-point-based invoice calculations leading to confusing results*

```
import java.text.NumberFormat;

class InvoiceCalc
{
   final static double DISCOUNT_PERCENT = 0.1; // 10%
   final static double TAX_PERCENT = 0.05; // 5%
   public static void main(String[] args)
   {
      double invoiceSubtotal = 285.36;
      double discount = invoiceSubtotal*DISCOUNT_PERCENT;
      double subtotalBeforeTax = invoiceSubtotal-discount;
      double salesTax = subtotalBeforeTax*TAX_PERCENT;
      double invoiceTotal = subtotalBeforeTax+salesTax;
      NumberFormat currencyFormat = NumberFormat.getCurrencyInstance();
      System.out.println("Subtotal: " + currencyFormat.format(invoiceSubtotal));
      System.out.println("Discount: " + currencyFormat.format(discount));
      System.out.println("SubTotal after discount: " +
                         currencyFormat.format(subtotalBeforeTax));
      System.out.println("Sales Tax: " + currencyFormat.format(salesTax));
      System.out.println("Total: " + currencyFormat.format(invoiceTotal));
   }
}
```

Listing 6–3 relies on the `NumberFormat` class (located in the `java.text`) package and its `format()` method to format a double precision floating-point value into a currency—I will discuss `NumberFormat` in Chapter 9. When you run `InvoiceCalc`, you will discover the following output:

```
Subtotal: $285.36
Discount: $28.54
SubTotal after discount: $256.82
Sales Tax: $12.84
Total: $269.67
```

This output reveals the correct subtotal, discount, subtotal after discount, and sales tax. In contrast, it incorrectly reveals 269.67 instead of 269.66 as the final total. The customer will not appreciate paying an extra penny, even though 269.67 is the correct value according to the floating-point calculations:

```
Subtotal: 285.36
Discount: 28.536
SubTotal after discount: 256.824
Sales Tax: 12.8412
Total: 269.6652
```

The problem arises from not rounding the result of each calculation to the nearest cent before performing the next calculation. As a result, the 0.024 in 256.824 and 0.0012 in 12.84 contribute to the final value, causing `NumberFormat`'s `format()` method to round this value to 269.67.

Java provides a solution to both problems in the form of a `java.math.BigDecimal` class. This immutable class (a `BigDecimal` instance cannot be modified) represents a signed decimal number (such as 23.653) of arbitrary *precision* (number of digits) with an associated *scale* (an integer that specifies the number of digits after the decimal point).

BigDecimal declares three convenience constants: ONE, TEN, and ZERO. Each constant is the BigDecimal equivalent of 1, 10, and 0 with a zero scale.

> **CAUTION:** BigDecimal declares several ROUND_-prefixed constants. These constants are largely obsolete and should be avoided, along with the public BigDecimal divide(BigDecimal divisor, int scale, int roundingMode) and public BigDecimal setScale(int newScale, int roundingMode) methods, which are still present so that dependent legacy code continues to compile.

BigDecimal also declares a variety of useful constructors and methods. A few of these constructors and methods are described in Table 6–2.

Table 6–2. *BigDecimal Constructors and Methods*

Method	Description
BigDecimal(int val)	Initialize the BigDecimal instance to val's digits. Set the scale to 0.
BigDecimal(String val)	Initialize the BigDecimal instance to the decimal equivalent of val. Set the scale to the number of digits after the decimal point, or 0 if no decimal point is specified. This constructor throws java.lang.NullPointerException when val is null, and java.lang.NumberFormatException when val's string representation is invalid (contains letters, for example).
BigDecimal abs()	Return a new BigDecimal instance that contains the absolute value of the current instance's value. The resulting scale is the same as the current instance's scale.
BigDecimal add(BigDecimal augend)	Return a new BigDecimal instance that contains the sum of the current value and the argument value. The resulting scale is the maximum of the current and argument scales. This method throws NullPointerException when augend is null.
BigDecimal divide(BigDecimal divisor)	Return a new BigDecimal instance that contains the quotient of the current value divided by the argument value. The resulting scale is the difference of the current and argument scales. It might be adjusted when the result requires more digits. This method throws NullPointerException when divisor is null, or java.lang.ArithmeticException when divisor represents 0 or the result cannot be represented exactly.
BigDecimal max(BigDecimal val)	Return either this or val, whichever BigDecimal instance contains the larger value. This method throws NullPointerException when val is null.

Method	Description
`BigDecimal min(BigDecimal val)`	Return either `this` or `val`, whichever `BigDecimal` instance contains the smaller value. This method throws `NullPointerException` when `val` is `null`.
`BigDecimal multiply(BigDecimal multiplicand)`	Return a new `BigDecimal` instance that contains the product of the current value and the argument value. The resulting scale is the sum of the current and argument scales. This method throws `NullPointerException` when `multiplicand` is `null`.
`BigDecimal negate()`	Return a new `BigDecimal` instance that contains the negative of the current value. The resulting scale is the same as the current scale.
`int precision()`	Return the precision of the current `BigDecimal` instance.
`BigDecimal remainder(BigDecimal divisor)`	Return a new `BigDecimal` instance that contains the remainder of the current value divided by the argument value. The resulting scale is the difference of the current scale and the argument scale. It might be adjusted when the result requires more digits. This method throws `NullPointerException` when `divisor` is `null`, or `ArithmeticException` when `divisor` represents 0.
`int scale()`	Return the scale of the current `BigDecimal` instance.
`BigDecimal setScale(int newScale, RoundingMode roundingMode)`	Return a new `BigDecimal` instance with the specified scale and rounding mode. If the new scale is greater than the old scale, additional zeros are added to the unscaled value. In this case no rounding is necessary. If the new scale is smaller than the old scale, trailing digits are removed. If these trailing digits are not zero, the remaining unscaled value has to be rounded. For this rounding operation, the specified rounding mode is used. This method throws `NullPointerException` when `roundingMode` is `null`, and `ArithmeticException` when `roundingMode` is set to `RoundingMode.ROUND_UNNECESSARY` but rounding is necessary based on the current scale.
`BigDecimal subtract(BigDecimal subtrahend)`	Return a new `BigDecimal` instance that contains the current value minus the argument value. The resulting scale is the maximum of the current and argument scales. This method throws `NullPointerException` when `subtrahend` is `null`.
`String toString()`	Return a string representation of this `BigDecimal`. Scientific notation is used when necessary.

Table 6–2 refers to `RoundingMode`, which is an enum containing various rounding mode constants. These constants are described in Table 6–3.

Table 6–3. *`RoundingMode` Constants*

Constant	Description
`CEILING`	Round toward positive infinity.
`DOWN`	Round toward zero.
`FLOOR`	Round toward negative infinity.
`HALF_DOWN`	Round toward the "nearest neighbor" unless both neighbors are equidistant, in which case round down.
`HALF_EVEN`	Round toward the "nearest neighbor" unless both neighbors are equidistant, in which case round toward the even neighbor.
`HALF_UP`	Round toward "nearest neighbor" unless both neighbors are equidistant, in which case round up. (This is the rounding mode commonly taught at school.)
`UNNECESSARY`	Rounding is not necessary because the requested operation produces the exact result.
`UP`	Positive values are rounded toward positive infinity and negative values are rounded toward negative infinity.

The best way to get comfortable with `BigDecimal` is to try it out. Listing 6–4 uses this class to correctly perform the invoice calculations that were presented in Listing 6–3.

Listing 6–4. *`BigDecimal`-based invoice calculations not leading to confusing results*

```
class InvoiceCalc
{
   public static void main(String[] args)
   {
      BigDecimal invoiceSubtotal = new BigDecimal("285.36");
      BigDecimal discountPercent = new BigDecimal("0.10");
      BigDecimal discount = invoiceSubtotal.multiply(discountPercent);
      discount = discount.setScale(2, RoundingMode.HALF_UP);
      BigDecimal subtotalBeforeTax = invoiceSubtotal.subtract(discount);
      subtotalBeforeTax = subtotalBeforeTax.setScale(2, RoundingMode.HALF_UP);
      BigDecimal salesTaxPercent = new BigDecimal("0.05");
      BigDecimal salesTax = subtotalBeforeTax.multiply(salesTaxPercent);
      salesTax = salesTax.setScale(2, RoundingMode.HALF_UP);
      BigDecimal invoiceTotal = subtotalBeforeTax.add(salesTax);
      invoiceTotal = invoiceTotal.setScale(2, RoundingMode.HALF_UP);
      System.out.println("Subtotal: " + invoiceSubtotal);
      System.out.println("Discount: " + discount);
      System.out.println("SubTotal after discount: " + subtotalBeforeTax);
      System.out.println("Sales Tax: " + salesTax);
      System.out.println("Total: " + invoiceTotal);
   }
}
```

Listing 6–4's `main()` method first creates `BigDecimal` objects `invoiceSubtotal` and `discountPercent` that are initialized to `285.36` and `0.10`, respectively. It multiplies `invoiceSubtotal` by `discountPercent` and assigns the `BigDecimal` result to `discount`.

At this point, `discount` contains 28.5360. Apart from the trailing zero, this value is the same as that generated by `invoiceSubtotal*DISCOUNT_PERCENT` in Listing 6–3. The value that should be stored in `discount` is 28.54. To correct this problem before performing another calculation, `main()` calls `discount`'s `setScale()` method with these arguments:

- `2`: Two digits after the decimal point
- `RoundingMode.HALF_UP`: The conventional approach to rounding

After setting the scale and proper rounding mode, `main()` subtracts `discount` from `invoiceSubtotal`, and assigns the resulting `BigDecimal` instance to `subtotalBeforeTax`. `main()` calls `setScale()` on `subtotalBeforeTax` to properly round its value before moving on to the next calculation.

`main()` next creates a `BigDecimal` object named `salesTaxPercent` that is initialized to `0.05`. It then multiplies `subtotalBeforeTax` by `salesTaxPercent`, assigning the result to `salesTax`, and calls `setScale()` on this `BigDecimal` object to properly round its value.

Moving on, `main()` adds `salesTax` to `subtotalBeforeTax`, saving the result in `invoiceTotal`, and rounds the result via `setScale()`. The values in these objects are sent to the standard output device via `System.out.println()`, which calls their `toString()` methods to return string representations of the `BigDecimal` values.

When you run this new version of `InvoiceCalc`, you will discover the following output:

```
Subtotal: 285.36
Discount: 28.54
SubTotal after discount: 256.82
Sales Tax: 12.84
Total: 269.66
```

> **CAUTION:** `BigDecimal` declares a `public BigDecimal(double val)` constructor that you should avoid using if at all possible. This constructor initializes the `BigDecimal` instance to the value stored in `val`, making it possible for this instance to reflect an invalid representation when the `double` cannot be stored exactly. For example, `BigDecimal(0.1)` results in `0.1000000000000000055511151231257827021181583404541015625` being stored in the instance. In contrast, `BigDecimal("0.1")` stores `0.1` exactly.

BigInteger

`BigDecimal` stores a signed decimal number as an unscaled value with a 32-bit integer scale. The unscaled value is stored in an instance of the `java.math.BigInteger` class.

`BigInteger` is an immutable class that represents a signed integer of arbitrary precision. It stores its value in *two's complement format* (all bits are flipped—1s to 0s and 0s to

1s—and 1 is added to the result to be compatible with the two's complement format used by Java's byte integer, short integer, integer, and long integer types).

> **NOTE:** Check out Wikipedia's "Two's complement" entry (`http://en.wikipedia.org/wiki/Two%27s_complement`) to learn more about two's complement.

`BigInteger` declares three convenience constants: `ONE`, `TEN`, and `ZERO`. Each constant is the `BigInteger` equivalent of 1, 10, and 0.

`BigInteger` also declares a variety of useful constructors and methods. A few of these constructors and methods are described in Table 6–4.

Table 6–4. *`BigInteger` Constructors and Methods*

Method	Description
`BigInteger(byte[] val)`	Initialize the `BigInteger` instance to the integer that is stored in the `val` array, with `val[0]` storing the integer's most significant (leftmost) eight bits. This constructor throws `NullPointerException` when `val` is `null`, and `NumberFormatException` when `val.length` equals 0.
`BigInteger(String val)`	Initialize the `BigInteger` instance to the integer equivalent of `val`. This constructor throws `NullPointerException` when `val` is `null`, and `NumberFormatException` when `val`'s string representation is invalid (contains letters, for example).
`BigInteger abs()`	Return a new `BigInteger` instance that contains the absolute value of the current instance's value.
`BigInteger add(BigInteger augend)`	Return a new `BigInteger` instance that contains the sum of the current value and the argument value. This method throws `NullPointerException` when `augend` is `null`.
`BigInteger divide(BigInteger divisor)`	Return a new `BigInteger` instance that contains the quotient of the current value divided by the argument value. This method throws `NullPointerException` when `divisor` is `null`, and `ArithmeticException` when `divisor` represents 0 or the result cannot be represented exactly.
`BigInteger max(BigInteger val)`	Return either `this` or `val`, whichever `BigInteger` instance contains the larger value. This method throws `NullPointerException` when `val` is `null`.
`BigInteger min(BigInteger val)`	Return either `this` or `val`, whichever `BigInteger` instance contains the smaller value. This method throws `NullPointerException` when `val` is `null`.

Method	Description
`BigInteger multiply(BigInteger multiplicand)`	Return a new `BigInteger` instance that contains the product of the current value and the argument value. This method throws `NullPointerException` when `multiplicand` is `null`.
`BigInteger negate()`	Return a new `BigInteger` instance that contains the negative of the current value.
`BigInteger remainder(BigInteger divisor)`	Return a new `BigInteger` instance that contains the remainder of the current value divided by the argument value. This method throws `NullPointerException` when `divisor` is `null`, and `ArithmeticException` when `divisor` represents 0.
`BigInteger subtract(BigInteger subtrahend)`	Return a new `BigInteger` instance that contains the current value minus the argument value. This method throws `NullPointerException` when `subtrahend` is `null`.
`String toString()`	Return a string representation of this `BigInteger`.

The best way to get comfortable with `BigInteger` is to try it out. Listing 6–5 uses this class in a `factorial()` method comparison context.

Listing 6–5. *Comparing `factorial()` methods*

```
class FactComp
{
   public static void main(String[] args)
   {
      System.out.println(factorial(12));
      System.out.println();
      System.out.println(factorial(20L));
      System.out.println();
      System.out.println(factorial(170.0));
      System.out.println();
      System.out.println(factorial(new BigInteger("170")));
      System.out.println();
      System.out.println(factorial(25.0));
      System.out.println();
      System.out.println(factorial(new BigInteger("25")));
   }
   public static int factorial(int n)
   {
      if (n == 0)
         return 1;
      else
         return n*factorial(n-1);
   }
   public static long factorial(long n)
   {
      if (n == 0)
         return 1;
      else
         return n*factorial(n-1);
```

```
   }
   public static double factorial(double n)
   {
      if (n == 1.0)
         return 1.0;
      else
         return n*factorial(n-1);
   }
   public static BigInteger factorial(BigInteger n)
   {
      if (n.equals(BigInteger.ZERO))
         return BigInteger.ONE;
      else
         return n.multiply(factorial(n.subtract(BigInteger.ONE)));
   }
}
```

Listing 6–5 compares four versions of the recursive `factorial()` method. This comparison reveals the largest argument that can be passed to each of the first three methods before the returned factorial value becomes meaningless, because of limits on the range of values that can be accurately represented by the numeric type.

The first version is based on `int` and has a useful argument range of 0 through 12. Passing any argument greater than 12 results in a factorial that cannot be represented accurately as an `int`.

You can increase the useful range of `factorial()`, but not by much, by changing the parameter and return types to `long`. After making these changes, you will discover that the upper limit of the useful range is 20.

To further increase the useful range, you might create a version of `factorial()` whose parameter and return types are `double`. This is possible because whole numbers can be represented exactly as `double`s. However, the largest useful argument that can be passed is 170.0. Anything higher than this value results in `factorial()` returning +infinity.

It is possible that you might need to calculate a higher factorial value, perhaps in the context of calculating a statistics problem involving combinations or permutations. The only way to accurately calculate this value is to use a version of `factorial()` based on `BigInteger`.

When you run the previous application, it generates the following output:

```
479001600

2432902008176640000

7.257415615307994E306

7257415615307998967396728211129263114716991681296451376543577798900561843401706157852350
7492426174595114909912378385207766660225654427530253289007732075109024004302800582956039
6661259965825710439855829425756896631343961226257109494680671120556888045719334021266145
28000000000000000000000000000000000000000000

1.5511210043330986E25

15511210043330985984000000
```

The first three values represent the highest factorials that can be returned by the `int`-based, `long`-based, and `double`-based `factorial()` methods. The fourth value represents the `BigInteger` equivalent of the highest `double` factorial.

Notice that the `double` method fails to accurately represent 170! (! is the math symbol for factorial). Its precision is simply too small. Although the method attempts to round the smallest digit, rounding does not always work—the number ends in 7994 instead of 7998. Rounding is only accurate up to argument 25.0, as the last two output lines reveal.

> **NOTE:** RSA encryption, `BigDecimal`, and factorial are practical examples of `BigInteger`'s usefulness. However, you can also use `BigInteger` in unusual ways. For example, my February 2006 *JavaWorld* article titled "Travel Through Time with Java" (`http://www.javaworld.com/javaworld/jw-02-2006/jw-0213-funandgames.html`), a part of my Java Fun and Games series, used `BigInteger` to store an image as a very large integer. The idea was to experiment with `BigInteger` methods to look for images of people and places that existed in the past, will exist in the future, or might never exist. If this craziness appeals to you, check out my article.

Package Information

The `java.lang.Package` class provides access to information about a package (see Chapter 4 for an introduction to packages). This information includes version information about the implementation and specification of a Java package, the name of the package, and an indication of whether or not the package has been *sealed* (all classes that are part of the package are archived in the same JAR file).

> **NOTE:** Chapter 1 introduces JAR files.

Table 6–5 describes some of `Package`'s methods.

Table 6–5. *`Package` Methods*

Method	Description
`String getImplementationTitle()`	Return the title of this package's implementation, which might be null. The format of the title is unspecified.
`String getImplementationVendor()`	Return the name of the vendor or organization that provides this package's implementation. This name might be null. The format of the name is unspecified.
`String getImplementationVersion()`	Return the version number of this package's implementation, which might be null. This version string must be a sequence of positive decimal integers separated by periods and might have leading zeros.

Method	Description
`String getName()`	Return the name of this package in standard dot notation; for example, `java.lang`.
`static Package getPackage(String packageName)`	Return the `Package` object that is associated with the package identified as `packageName`, or null when the package identified as `packageName` cannot be found. This method throws `NullPointerException` when `packageName` is `null`.
`static Package[] getPackages()`	Return an array of all `Package` objects that are accessible to this method's caller.
`String getSpecificationTitle()`	Return the title of this package's specification, which might be null. The format of the title is unspecified.
`String getSpecificationVendor()`	Return the name of the vendor or organization that provides the specification that is implemented by this package. This name might be null. The format of the name is unspecified.
`String getSpecificationVersion()`	Return the version number of the specification of this package's implementation, which might be null. This version string must be a sequence of positive decimal integers separated by periods, and might have leading zeros.
`boolean isCompatibleWith(String desired)`	Check this package to determine if it is compatible with the specified version string, by comparing this package's specification version with the `desired` version. Return true when this package's specification version number is greater than or equal to the desired version number (this package is compatible); otherwise, return false. This method throws `NullPointerException` when `desired` is `null`, and `NumberFormatException` when this package's version number or the `desired` version number is not in dotted form.
`boolean isSealed()`	Return true when this package has been sealed; otherwise, return false.

I have created a `PackageInfo` application that demonstrates most of Table 6–5's `Package` methods. Listing 6–6 presents this application's source code.

Listing 6–6. *Obtaining information about a package*

```
public class PackageInfo
{
   public static void main(String[] args)
   {
      if (args.length == 0)
      {
         System.err.println("usage: java PackageInfo packageName [version]");
         return;
      }
```

```
      Package pkg = Package.getPackage(args[0]);
      if (pkg == null)
      {
         System.err.println(args[0] + " not found");
         return;
      }
      System.out.println("Name: " + pkg.getName());
      System.out.println("Implementation title: " +
                         pkg.getImplementationTitle());
      System.out.println("Implementation vendor: " +
                         pkg.getImplementationVendor());
      System.out.println("Implementation version: " +
                         pkg.getImplementationVersion());
      System.out.println("Specification title: " +
                         pkg.getSpecificationTitle());
      System.out.println("Specification vendor: " +
                         pkg.getSpecificationVendor());
      System.out.println("Specification version: " +
                         pkg.getSpecificationVersion());
      System.out.println("Sealed: " + pkg.isSealed());
      if (args.length > 1)
         System.out.println("Compatible with " + args[1] + ": " +
                            pkg.isCompatibleWith(args[1]));
   }
}
```

To use this application, specify at least a package name on the command line. For example, `java PackageInfo java.lang` returns the following output under Java version 6:

```
Name: java.lang
Implementation title: Java Runtime Environment
Implementation vendor: Sun Microsystems, Inc.
Implementation version: 1.6.0_16
Specification title: Java Platform API Specification
Specification vendor: Sun Microsystems, Inc.
Specification version: 1.6
Sealed: false
```

`PackageInfo` also lets you determine if the package's specification is compatible with a specific version number. A package is compatible with its predecessors.

For example, `java PackageInfo java.lang 1.6` outputs `Compatible with 1.6: true`, whereas `java PackageInfo java.lang 1.8` outputs `Compatible with 1.8: false`.

You can also use `PackageInfo` with your own packages, which you learned to create in Chapter 4. For example, that chapter presented a `logging` package.

Copy `PackageInfo.class` into the directory containing the `logging` package directory (which contains the compiled classfiles), and execute `java PackageInfo logging`.

`PackageInfo` responds by displaying the following output:

```
logging not found
```

This error message is presented because `getPackage()` requires at least one classfile to be loaded from the package before it returns a `Package` object describing that package.

The only way to eliminate the previous error message is to load a class from the package. Accomplish this task by merging Listing 6–7 into Listing 6–6.

Listing 6–7. *Dynamically loading a class from a classfile*

```
if (args.length == 3)
try
{
   Class.forName(args[2]);
}
catch (ClassNotFoundException cnfe)
{
   System.err.println("cannot load " + args[2]);
   return;
}
```

This code fragment, which must precede `Package pkg = Package.getPackage(args[0]);`, loads the classfile named by the revised `PackageInfo` application's third command-line argument.

Run the new `PackageInfo` application via `java PackageInfo logging 1.5 logging.File` and you will observe the following output—this command line identifies `logging`'s `File` class as the class to load:

```
Name: logging
Implementation title: null
Implementation vendor: null
Implementation version: null
Specification title: null
Specification vendor: null
Specification version: null
Sealed: false
Exception in thread "main" java.lang.NumberFormatException: Empty version
 string
       at java.lang.Package.isCompatibleWith(Unknown Source)
       at PackageInfo.main(PackageInfo.java:43)
```

It is not surprising to see all of these null values because no package information has been added to the `logging` package. Also, `NumberFormatException` is thrown from `isCompatibleWith()` because the `logging` package does not contain a specification version number in dotted form (it is null).

Perhaps the simplest way to place package information into the `logging` package is to create a `logging.jar` file in a similar manner to the example shown in Chapter 4. But first, you must create a small text file that contains the package information. You can choose any name for the file. Listing 6–8 reveals my choice of `manifest.mf`.

Listing 6–8. *`manifest.mf` containing the package information*

```
Implementation-Title: Logging Implementation
Implementation-Vendor: Jeff Friesen
Implementation-Version: 1.0a
Specification-Title: Logging Specification
Specification-Vendor: Jeff "JavaJeff" Friesen
Specification-Version: 1.0
Sealed: true
```

NOTE: Make sure to press the Return/Enter key at the end of the final line (`Sealed: true`). Otherwise, you will probably observe `Sealed: false` in the output because this entry will not be stored in the `logging` package by the JDK's `jar` tool—`jar` is a bit quirky.

Execute the following command line to create a JAR file that includes `logging` and its files, and whose *manifest*, a special file named `MANIFEST.MF` that stores information about the contents of a JAR file, contains the contents of Listing 6–8:

```
jar cfm logging.jar manifest.mf logging
```

This command line creates a JAR file named `logging.jar` (via the `c` [create] and `f` [file] options). It also merges the contents of `manifest.mf` (via the `m` [manifest] option) into `MANIFEST.MF`, which is stored in the package's `META-INF` directory.

NOTE: To learn more about a JAR file's manifest, read the "JAR Manifest" section of the JDK documentation's "JAR File Specification" page (`http://java.sun.com/javase/6/docs/technotes/guides/jar/jar.html#JAR%20Manifest`).

Assuming that the `jar` tool presents no error messages, execute the following Windows-oriented command line (or a command line suitable for your platform) to run `PackageInfo` and extract the package information from the `logging` package:

```
java -cp logging.jar;. PackageInfo logging 1.0 logging.File
```

This time, you should see the following output:

```
Name: logging
Implementation title: Logging Implementation
Implementation vendor: Jeff Friesen
Implementation version: 1.0a
Specification title: Logging Specification
Specification vendor: Jeff "JavaJeff" Friesen
Specification version: 1.0
Sealed: true
Compatible with 1.0: true
```

Primitive Wrapper Classes

The `java.lang` package includes `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, and `Short`. These classes are known as *primitive wrapper classes* because their instances wrap themselves around values of primitive types.

NOTE: The primitive wrapper classes are also known as *value classes*.

Java provides these eight primitive wrapper classes for two reasons:

- The collections framework (discussed Chapter 8) provides lists, sets, and maps that can only store objects; they cannot store primitive values. You store a primitive value in a primitive wrapper class instance and store the instance in the collection.
- These classes provide a good place to associate useful constants (such as `MAX_VALUE` and `MIN_VALUE`) and class methods (such as `Integer`'s `parseInt()` methods and `Character`'s `isDigit()`, `isLetter()`, and `toUpperCase()` methods) with the primitive types.

This section introduces you to each of these primitive wrapper classes and a class named `Number`.

Boolean

`Boolean` is the smallest of the primitive wrapper classes. This class declares three constants, including `TRUE` and `FALSE`, which denote precreated `Boolean` objects.

`Boolean` also declares a pair of constructors for initializing a `Boolean` object:

- `Boolean(boolean value)` initializes the `Boolean` object to `value`.
- `Boolean(String s)` converts `s`'s text to a true or false value and stores this value in the `Boolean` object.

The second constructor compares `s`'s value with `true`. Because the comparison is case-insensitive, any combination of these four letters (such as `true`, `TRUE`, or `tRue`) results in true being stored in the object. Otherwise, the constructor stores false in the object.

`Boolean`'s constructors are complemented by `boolean booleanValue()`, which returns the wrapped Boolean value.

`Boolean` also declares or overrides the following methods:

- `int compareTo(Boolean b)` compares the current `Boolean` object with `b` to determine their relative order. The method returns 0 when the current object contains the same Boolean value as `b`, a positive value when the current object contains true and `b` contains false, and a negative value when the current object contains false and `b` contains true.
- `boolean equals(Object o)` compares the current `Boolean` object with `o` and returns true when `o` is not null, `o` is of type `Boolean`, and both objects contain the same Boolean value.
- `static boolean getBoolean(String name)` returns true when a system property (discussed in Chapter 7) identified by `name` exists and is equal to true.

- `int hashCode()` returns a suitable hash code that allows `Boolean` objects to be used with hash-based collections (discussed in Chapter 8).
- `static boolean parseBoolean(String s)` parses `s`, returning true if `s` equals `"true"`, `"TRUE"`, `"True"`, or any other combination of these letters. Otherwise, this method returns false. (*Parsing* breaks a sequence of characters into meaningful components, known as *tokens*.)
- `String toString()` returns `"true"` when the current `Boolean` instance contains true; otherwise, this method returns `"false"`.
- `static String toString(boolean b)` returns `"true"` when `b` contains true; otherwise, this method returns `"false"`.
- `static Boolean valueOf(boolean b)` returns `TRUE` when `b` contains true or `FALSE` when `b` contains false.
- `static Boolean valueOf(String s)` returns `TRUE` when `s` equals `"true"`, `"TRUE"`, `"True"`, or any other combination of these letters. Otherwise, this method returns `FALSE`.

> **CAUTION:** Newcomers to the `Boolean` class often think that `getBoolean()` returns a `Boolean` object's true/false value. However, `getBoolean()` returns the value of a Boolean-based system property—I discuss system properties in Chapter 7. If you need to return a `Boolean` object's true/false value, use the `booleanValue()` method instead.

It is often better to use `TRUE` and `FALSE` than to create `Boolean` objects. For example, suppose you need a method that returns a `Boolean` object containing true when the method's `double` argument is negative, or false when this argument is zero or positive. You might declare your method like the `isNegative()` method shown in Listing 6–9.

Listing 6–9. ***An `isNegative()` method with unnecessary `Boolean` object creation***

```
public Boolean isNegative(double d)
{
   return new Boolean(d < 0);
}
```

Although this method is concise, it unnecessarily creates a `Boolean` object. When the method is called frequently, many `Boolean` objects are created that consume heap space. When heap space runs low, the garbage collector runs and slows down the application, which impacts performance.

Listing 6–10 reveals a better way to code `isNegative()`.

Listing 6–10. ***A refactored `isNegative()` method not creating `Boolean` objects***

```
public Boolean isNegative(double d)
{
   return (d < 0) ? Boolean.TRUE : Boolean.FALSE;
}
```

This method avoids creating `Boolean` objects by returning either the precreated `TRUE` or `FALSE` object.

> **TIP:** You should strive to create as few objects as possible. Not only will your applications have smaller memory footprints, they will perform better because the garbage collector will not be required to run as often.

Character

`Character` is the largest of the primitive wrapper classes, containing many constants, a constructor, many methods, and a pair of nested classes (`Subset` and `UnicodeBlock`).

> **NOTE:** `Character`'s complexity derives from Java's support for Unicode (`http://en.wikipedia.org/wiki/Unicode`). For brevity, I ignore much of `Character`'s Unicode-related complexity, which is beyond the scope of this chapter.

`Character` declares a single `Character(char value)` constructor, which you use to initialize a `Character` object to `value`. This constructor is complemented by `char charValue()`, which returns the wrapped character value.

When you start writing applications, you might codify expressions such as `ch >= '0' && ch <= '9'` (test `ch` to see if it contains a digit) and `ch >= 'A' && ch <= 'Z'` (test `ch` to see if it contains an uppercase letter). You should avoid doing so for three reasons:

- It is too easy to introduce a bug into the expression. For example, `ch > '0' && ch <= '9'` introduces a subtle bug that does not include `'0'` in the comparison.
- The expressions are not very descriptive of what they are testing.
- The expressions are biased toward Latin digits (0–9) and letters (A–Z and a–z). They do not take into account digits and letters that are valid in other languages. For example, `'\u0beb'` is a character literal representing one of the digits in the Tamil language.

`Character` declares several comparison and conversion utility methods that address these concerns. These methods include the following:

- `static boolean isDigit(char ch)` returns true when `ch` contains a digit (typically 0 through 9, but also digits in other languages).
- `static boolean isLetter(char ch)` returns true when `ch` contains a letter (typically A–Z or a–z, but also letters in other languages).
- `static boolean isLetterOrDigit(char ch)` returns true when `ch` contains a letter or digit (typically A–Z, a–z, or 0–9, but also letters or digits in other languages).

- `static boolean isLowerCase(char ch)` returns true when `ch` contains a lowercase letter.
- `static boolean isUpperCase(char ch)` returns true when `ch` contains an uppercase letter.
- `static boolean isWhitespace(char ch)` returns true when `ch` contains a whitespace character (typically a space, a horizontal tab, a carriage return, or a line feed).
- `static char toLowerCase(char ch)` returns the lowercase equivalent of `ch`'s uppercase letter; otherwise, this method returns `ch`'s value.
- `static char toUpperCase(char ch)` returns the uppercase equivalent of `ch`'s lowercase letter; otherwise, this method returns `ch`'s value.

For example, `isDigit(ch)` is preferable to `ch >= '0' && ch <= '9'` because it avoids a source of bugs, is more readable, and returns true for non-Latin digits (such as `'\u0beb'`) as well as Latin digits.

Float and Double

`Float` and `Double` store floating-point and double precision floating-point values in `Float` and `Double` objects, respectively. These classes declare the following constants:

- `MAX_VALUE` identifies the maximum value that can be represented as a `float` or `double`.
- `MIN_VALUE` identifies the minimum value that can be represented as a `float` or `double`.
- `NaN` represents `0.0F/0.0F` as a `float` and `0.0/0.0` as a `double`.
- `NEGATIVE_INFINITY` represents -infinity as a `float` or `double`.
- `POSITIVE_INFINITY` represents +infinity as a `float` or `double`.

`Float` and `Double` also declare the following constructors for initializing their objects:

- `Float(float value)` initializes the `Float` object to `value`.
- `Float(double value)` initializes the `Float` object to the `float` equivalent of `value`.
- `Float(String s)` converts `s`'s text to a floating-point value and stores this value in the `Float` object.
- `Double(double value)` initializes the `Double` object to `value`.
- `Double(String s)` converts `s`'s text to a double precision floating-point value and stores this value in the `Double` object.

Float's constructors are complemented by float floatValue(), which returns the wrapped floating-point value. Similarly, Double's constructors are complemented by double doubleValue(), which returns the wrapped double precision floating-point value.

Float declares several utility methods in addition to floatValue(). These methods include the following:

- static int floatToIntBits(float value) converts value to a 32-bit integer.
- static boolean isInfinite(float f) returns true when f's value is +infinity or -infinity. A related public boolean isInfinite() method returns true when the current Float object's value is +infinity or -infinity.
- static boolean isNaN(float f) returns true when f's value is NaN. A related public boolean isNaN() method returns true when the current Float object's value is NaN.
- static float parseFloat(String s) parses s, returning the floating-point equivalent of s's textual representation of a floating-point value or throwing NumberFormatException when this representation is invalid (contains letters, for example).

Double declares several utility methods in addition to doubleValue(). These methods include the following:

- static long doubleToLongBits(double value) converts value to a long integer.
- static boolean isInfinite(double d) returns true when d's value is +infinity or -infinity. A related boolean isInfinite() method returns true when the current Double object's value is +infinity or -infinity.
- static boolean isNaN(double d) returns true when d's value is NaN. A related public boolean isNaN() method returns true when the current Double object's value is NaN.
- static double parseDouble(String s) parses s, returning the double precision floating-point equivalent of s's textual representation of a double precision floating-point value or throwing NumberFormatException when this representation is invalid.

The floatToIntBits() and doubleToIntBits() methods are used in implementations of the equals() and hashCode() methods that must take float and double fields into account. floatToIntBits() and doubleToIntBits() allow equals() and hashCode() to respond properly to the following situations:

- `equals()` must return true when `f1` and `f2` contain `Float.NaN` (or `d1` and `d2` contain `Double.NaN`). If `equals()` was implemented in a manner similar to `f1.floatValue() == f2.floatValue()` (or `d1.doubleValue() == d2.doubleValue()`), this method would return false because NaN is not equal to anything, including itself.
- `equals()` must return false when `f1` contains +0.0 and `f2` contains -0.0 (or vice versa), or `d1` contains +0.0 and `d2` contains -0.0 (or vice versa). If `equals()` was implemented in a manner similar to `f1.floatValue() == f2.floatValue()` (or `d1.doubleValue() == d2.doubleValue()`), this method would return true because `+0.0 == -0.0` returns true.

These requirements are needed for hash-based collections (discussed in Chapter 8) to work properly. Listing 6–11 shows how they impact `Float`'s and `Double`'s `equals()` methods.

Listing 6–11. ***Demonstrating*** `Float`***'s*** `equals()` ***method in a NaN context and*** `Double`***'s*** `equals()` ***method in a +/-0.0 context***

```
public static void main(String[] args)
{
   Float f1 = new Float(Float.NaN);
   System.out.println(f1.floatValue());
   Float f2 = new Float(Float.NaN);
   System.out.println(f2.floatValue());
   System.out.println(f1.equals(f2));
   System.out.println(Float.NaN == Float.NaN);
   System.out.println();
   Double d1 = new Double(+0.0);
   System.out.println(d1.doubleValue());
   Double d2 = new Double(-0.0);
   System.out.println(d2.doubleValue());
   System.out.println(d1.equals(d2));
   System.out.println(+0.0 == -0.0);
}
```

Run this application. The following output proves that `Float`'s `equals()` method properly handles NaN and `Double`'s `equals()` method properly handles +/-0.0:

```
NaN
NaN
true
false

0.0
-0.0
false
true
```

> **TIP:** If you want to test a `float` or `double` value for equality with +infinity or -infinity (but not both), do not use `isInfinite()`. Instead, compare the value with `NEGATIVE_INFINITY` or `POSITIVE_INFINITY` via `==`. For example, `f == Float.NEGATIVE_INFINITY`.

You will find `parseFloat()` and `parseDouble()` useful in many contexts. For example, Listing 6–12 uses `parseDouble()` to parse command-line arguments into `double`s.

Listing 6–12. *Parsing command-line arguments into double precision floating-point values*

```
public static void main(String[] args)
{
   if (args.length != 3)
   {
      System.err.println("usage: java Calc value1 op value2");
      System.err.println("op is one of +, -, *, or /");
      return;
   }
   try
   {
      double value1 = Double.parseDouble(args[0]);
      double value2 = Double.parseDouble(args[2]);
      if (args[1].equals("+"))
         System.out.println(value1+value2);
      else
      if (args[1].equals("-"))
         System.out.println(value1-value2);
      else
      if (args[1].equals("*"))
         System.out.println(value1*value2);
      else
      if (args[1].equals("/"))
         System.out.println(value1/value2);
      else
         System.err.println("invalid operator: " + args[1]);
   }
   catch (NumberFormatException nfe)
   {
      System.err.println("Bad number format: " + nfe.getMessage());
   }
}
```

Specify `java Calc 10E+3 + 66.0` to try out the `Calc` application. This application responds by outputting `10066.0`. If you specified `java Calc 10E+3 + A` instead, you would observe `Bad number format: For input string: "A"` as the output, which is in response to the second `parseDouble()` method call's throwing of a `NumberFormatException` object.

Although `NumberFormatException` describes an unchecked exception, and although unchecked exceptions are often not handled because they represent coding mistakes, `NumberFormatException` does not fit this pattern in this example. The exception does not arise from a coding mistake; it arises from someone passing an illegal numeric argument to the application, which cannot be avoided through proper coding.

Integer, Long, Short, and Byte

Integer, Long, Short, and Byte store 32-bit, 64-bit, 16–bit, and 8-bit integer values in Integer, Long, Short, and Byte objects, respectively.

Each class declares MAX_VALUE and MIN_VALUE constants that identify the maximum and minimum values that can be represented by its associated primitive type.

These classes also declare the following constructors for initializing their objects:

- Integer(int value) initializes the Integer object to value.
- Integer(String s) converts s's text to a 32-bit integer value and stores this value in the Integer object.
- Long(long value) initializes the Long object to value.
- Long(String s) converts s's text to a 64-bit integer value and stores this value in the Long object.
- Short(short value) initializes the Short object to value.
- Short(String s) converts s's text to a 16–bit integer value and stores this value in the Short object.
- Byte(byte value) initializes the Byte object to value.
- Byte(String s) converts s's text to an 8-bit integer value and stores this value in the Byte object.

Integer's constructors are complemented by int intValue(), Long's constructors are complemented by long longValue(), Short's constructors are complemented by short shortValue(), and Byte's constructors are complemented by byte byteValue(). These methods return wrapped integers.

These classes declare various useful integer-oriented methods. For example, Integer declares the following class methods for converting a 32-bit integer to a String according to a specific representation (binary, hexadecimal, octal, and decimal):

- static String toBinaryString(int i) returns a String object containing i's binary representation. For example, Integer.toBinaryString(255) returns a String object containing 11111111.
- static String toHexString(int i) returns a String object containing i's hexadecimal representation. For example, Integer.toHexString(255) returns a String object containing ff.
- static String toOctalString(int i) returns a String object containing i's octal representation. For example, toOctalString(64) returns a String object containing 377.

- `static String toString(int i)` returns a `String` object containing `i`'s decimal representation. For example, `toString(255)` returns a `String` object containing 255.

It is often convenient to prepend zeros to a binary string so that you can align multiple binary strings in columns. For example, you might want to create an application that displays the following aligned output:

```
11110001
+
00000111
--------
11111000
```

Unfortunately, `toBinaryString()` does not let you accomplish this task. For example, `Integer.toBinaryString(7)` returns a `String` object containing 111 instead of 00000111. Listing 6–13's `toAlignedBinaryString()` method addresses this oversight.

Listing 6–13. *Aligning binary strings*

```
public static void main(String[] args)
{
   System.out.println(toAlignedBinaryString(7, 8));
   System.out.println(toAlignedBinaryString(255, 16));
   System.out.println(toAlignedBinaryString(255, 7));
}
static String toAlignedBinaryString(int i, int numBits)
{
   String result = Integer.toBinaryString(i);
   if (result.length() > numBits)
      return null; // cannot fit result into numBits columns
   int numLeadingZeros = numBits-result.length();
   String zerosPrefix = "";
   for (int j = 0; j < numLeadingZeros; j++)
      zerosPrefix += "0";
   return zerosPrefix + result;
}
```

The `toAlignedBinaryString()` method takes two arguments: the first argument specifies the 32-bit integer that is to be converted into a binary string, and the second argument specifies the number of bit columns in which to fit the string.

After calling `toBinaryString()` to return `i`'s equivalent binary string without leading zeros, `toAlignedBinaryString()` verifies that the string's digits can fit into the number of bit columns specified by `numBits`. If they do not fit, this method returns null. (You will learn about `length()` and other `String` methods in Chapter 7.)

Moving on, `toAlignedBinaryString()` calculates the number of leading `"0"`s to prepend to `result`, and then uses a for loop to create a string of leading zeros. This method ends by returning the leading zeros string prepended to the result string.

Although using the compound string concatenation with assignment operator (+=) in a loop to build a string looks okay, it is very inefficient because intermediate `String` objects are created and thrown away. However, I employed this inefficient code so that I can contrast it with the more efficient code that I present in Chapter 7.

When you run this application, it generates the following output:

```
00000111
0000000011111111
null
```

Number

Each of `Float`, `Double`, `Integer`, `Long`, `Short`, and `Byte` provides the other classes' *x*`Value()` methods in addition to its own *x*`Value()` method. For example, `Float` provides `doubleValue()`, `intValue()`, `longValue()`, `shortValue()`, and `byteValue()` in addition to `floatValue()`.

All six methods are members of `java.lang.Number`, which is the abstract superclass of `Float`, `Double`, `Integer`, `Long`, `Short`, and `Byte`—`Number`'s `floatValue()`, `doubleValue()`, `intValue()`, and `longValue()` methods are abstract. `Number` is also the superclass of `BigDecimal` and `BigInteger` (and some concurrency-related classes; see Chapter 9).

`Number` exists to simplify iterating over a collection of `Number` subclass objects. For example, you can declare a variable of `List<Number>` type and initialize it to an instance of `ArrayList<Number>`. You can then store a mixture of `Number` subclass objects in the collection, and iterate over this collection by calling a subclass method polymorphically.

References API

Chapter 2 introduced you to garbage collection, where you learned that the garbage collector removes an object from the heap when there are no more references to the object.

Chapter 3 introduced you to `Object`'s `finalize()` method, where you learned that the garbage collector calls this method before removing an object from the heap. This method gives the object an opportunity to perform cleanup.

This section continues from where Chapters 2 and 3 left off by introducing you to Java's References API. This API makes it possible for an application to interact with the garbage collector in limited ways.

The section first acquaints you with some basic terminology. It then introduces you to the API's `Reference` and `ReferenceQueue` classes, followed by the API's `SoftReference`, `WeakReference`, and `PhantomReference` classes.

Basic Terminology

When an application runs, its execution reveals a *root set of references*, a collection of local variables, parameters, class fields, and instance fields that currently exist and that contain (possibly null) references to objects. This root set changes over time as the application runs. For example, parameters disappear after a method returns.

Many garbage collectors identify this root set when they run. They use the root set to determine if an object is *reachable* (referenced, also known as *live*) or *unreachable* (not referenced). The garbage collector cannot collect reachable objects. Instead, it can only collect objects that, starting from the root set of references, cannot be reached.

> **NOTE:** Reachable objects include objects that are indirectly reachable from root-set variables, which means objects that are reachable through live objects that are directly reachable from those variables. An object that is unreachable by any path from any root-set variable is eligible for garbage collection.

Beginning with Java version 1.2, reachable objects were classified as strongly reachable, softly reachable, weakly reachable, and phantom reachable. Unlike strongly reachable objects, softly, weakly, and phantom reachable objects can be garbage collected.

The following list describes these four kinds of reachability in terms of reference strength, from strongest to weakest:

- An object is *strongly reachable* when it is reachable by a thread without the thread having to traverse References API objects—the thread follows a *strong reference* in a root-set variable. A newly created object (such as the object referenced by `d` in `Double d = new Double(1.0);`) is strongly reachable by the thread that created it. (I discuss threads in Chapter 7.)
- An object is *softly reachable* when it is not strongly reachable but can be reached by traversing a *soft reference* (a reference to the object where the reference is stored in a `SoftReference` object). The strongest reference to this object is a soft reference. When heap memory runs low, the garbage collector typically clears the soft references of the oldest softly reachable objects and removes those objects after finalizing them (by calling `finalize()`).
- An object is *weakly reachable* when it is not strongly or softly reachable but can be reached by traversing a *weak reference* (a reference to the object where the reference is stored in a `WeakReference` object). The strongest reference to this object is a weak reference. The garbage collector clears weak references to weakly reachable objects and throws away these objects (after finalizing them) the next time it runs, even when memory is plentiful.
- An object is *phantom reachable* when it is neither strongly, softly, nor weakly reachable, it has been finalized, and the garbage collector is ready to reclaim its memory. Furthermore, it is referred to by some *phantom reference* (a reference to the object where the reference is stored in a `PhantomReference` object). The strongest reference to this object is a phantom reference.

NOTE: Apart from the garbage collector being less eager to clean up the softly reachable object, a soft reference is exactly like a weak reference. Also, a weak reference is not strong enough to keep an object in memory.

The object whose reference is stored in a `SoftReference`, `WeakReference`, or `PhantomReference` object is known as a *referent*.

Reference and ReferenceQueue

The References API consists of five classes located in the `java.lang.ref` package. Central to this package are `Reference` and `ReferenceQueue`.

`Reference` is the abstract superclass of this package's concrete `SoftReference`, `WeakReference`, and `PhantomReference` subclasses.

`ReferenceQueue` is a concrete class whose instances describe queue data structures. When you associate a `ReferenceQueue` instance with a `Reference` subclass object (`Reference` object, for short), the `Reference` object is added to the queue when the referent to which its encapsulated reference refers becomes garbage.

NOTE: You associate a `ReferenceQueue` object with a `Reference` object by passing the `ReferenceQueue` object to an appropriate `Reference` subclass constructor.

`Reference` is declared as generic type `Reference<T>`, where `T` identifies the referent's type. This class provides the following methods:

- `void clear()` assigns null to the stored reference; the `Reference` object on which this method is called is not *enqueued* (inserted) into its associated reference queue (if there is an associated reference queue). (The garbage collector clears references directly; it does not call `clear()`. Instead, this method is called by applications.)
- `boolean enqueue()` adds the `Reference` object on which this method is called to the associated reference queue. This method returns true when this `Reference` object has become enqueued; otherwise, this method returns false—this `Reference` object was already enqueued or was not associated with a queue when created. (The garbage collector enqueues `Reference` objects directly; it does not call `enqueue()`. Instead, this method is called by applications.)
- `T get()` returns this `Reference` object's stored reference. The return value is null when the stored reference has been cleared, either by the application or by the garbage collector.

- `boolean isEnqueued()` returns true when this `Reference` object has been enqueued, either by the application or by the garbage collector. Otherwise, this method returns false—this `Reference` object was not associated with a queue when created.

> **NOTE:** `Reference` also declares constructors. Because these constructors are package-private, only classes in the `java.lang.ref` package can subclass `Reference`. This restriction is necessary because instances of `Reference`'s subclasses must work closely with the garbage collector.

`ReferenceQueue` is declared as generic type `ReferenceQueue<T>`, where `T` identifies the referent's type. This class declares the following constructor and methods:

- `ReferenceQueue()` initializes a new `ReferenceQueue` instance.
- `Reference<? extends T> poll()` polls this queue to check for an available `Reference` object. If one is available, the object is removed from the queue and returned. Otherwise, this method returns immediately with a null value.
- `Reference<? extends T> remove()` removes the next `Reference` object from the queue and returns this object. This method waits indefinitely for a `Reference` object to become available, and throws `java.lang.InterruptedException` when this wait is interrupted.
- `Reference<? extends T> remove(long timeout)` removes the next `Reference` object from the queue and returns this object. This method waits until a `Reference` object becomes available or until `timeout` milliseconds have elapsed—passing 0 to `timeout` causes the method to wait indefinitely. If `timeout`'s value expires, the method returns null. This method throws `java.lang.IllegalArgumentException` when `timeout`'s value is negative, or `InterruptedException` when this wait is interrupted.

SoftReference

The `SoftReference` class describes a `Reference` object whose referent is softly reachable. In addition to inheriting `Reference`'s methods and overriding `get()`, this generic class provides the following constructors for initializing a `SoftReference` object:

- `SoftReference(T r)` encapsulates `r`'s reference. The `SoftReference` object behaves as a soft reference to `r`. No `ReferenceQueue` object is associated with this `SoftReference` object.

- `SoftReference(T r, ReferenceQueue<? super T> q)` encapsulates `r`'s reference. The `SoftReference` object behaves as a soft reference to `r`. The `ReferenceQueue` object identified by `q` is associated with this `SoftReference` object. Passing `null` to `q` indicates a soft reference without a queue.

`SoftReference` is useful for implementing caches, such as a cache of images. An image cache keeps images in memory (because it takes time to load them from disk) and ensures that duplicate (and possibly very large) images are not stored in memory.

The image cache contains references to image objects that are already in memory. If these references were strong, the images would remain in memory. You would then need to figure out which images are no longer needed and remove them from memory so that they can be garbage collected.

Having to manually remove images duplicates the work of a garbage collector. However, if you wrap the references to the image objects in `SoftReference` objects, the garbage collector will determine when to remove these objects (typically when heap memory runs low) and perform the removal on your behalf.

Listing 6–14 shows how you might use `SoftReference` to maintain a cache of images.

Listing 6–14. *Maintaining a cache of images*

```
class Image
{
   private byte[] image;
   private Image(String name)
   {
      image = new byte[1024*100];
   }
   static Image getImage(String name)
   {
      return new Image(name);
   }
}
public class ImageCache
{
   final static int NUM_IMAGES = 200;
   @SuppressWarnings("unchecked")
   public static void main(String[] args)
   {
      String[] imageNames = new String[NUM_IMAGES];
      for (int i = 0; i < imageNames.length; i++)
         imageNames[i] = new String("image" + i + ".gif");

      SoftReference<Image>[] cache = new SoftReference[imageNames.length];
      for (int i = 0; i < cache.length; i++)
         cache[i] = new SoftReference<Image>(Image.getImage(imageNames[i]));

      for (int i = 0; i < cache.length; i++)
      {
         Image im = cache[i].get();
         if (im == null)
         {
```

```
                System.out.println(imageNames[i] + " not in cache");
                im = Image.getImage(imageNames[i]);
                cache[i] = new SoftReference<Image>(im);
            }
            System.out.println("Drawing image");
            im = null; // Remove strong reference to image.
        }
    }
}
```

This listing declares an `Image` class that simulates a loaded image. Each instance is created by calling the `getImage()` class method, and the instance's private `image` array occupies 100KB of memory.

The `main()` method first creates an array of `String` objects that contain image filenames. The technique employed in creating this array is inefficient. You will discover an efficient alternative in Chapter 7.

`main()` next creates an array of `SoftReference` objects that serves as a cache for `Image` objects. This array is initialized to `SoftReference` objects; each `SoftReference` object is initialized to an `Image` object's reference.

`main()` now enters the application's main loop. It iterates over the cache, retrieving each `Image` object or null when the garbage collector has cleared the soft reference to the `Image` object (so that it can make room in the heap).

If the reference assigned to `im` is not null, the `Image` object has not been made unreachable and subsequent code can draw the image on the screen. The `im = null;` assignment removes the strong reference to the `Image` object from the `im` root-set variable.

> **NOTE:** The `im = null;` assignment is not necessary in this application because either `im` is immediately overwritten by `get()`'s return value in the next loop iteration, or the loop and the application ends. Because `im`'s value might hang around for a while in a refactored and longer-lived version of this application, and the garbage collector would not be able to remove the associated `Image` object from the heap because that object would be strongly reachable, I've included this assignment to show you how to get rid of `im`'s value.

When the reference assigned to `im` is null, the `Image` object has been made unreachable and has probably been removed from the heap. In this case, the `Image` object must be re-created and stored in a new `SoftReference` object that is stored in the cache.

Here is a small portion of the output that I observed—you may have to adjust the application's code to observe similar output:

```
image162.gif not in cache
Drawing image
image163.gif not in cache
Drawing image
Drawing image
```

Regarding the last line of output, its `Drawing image` message implies that `image164.gif` is still in the cache. In other words, the associated `Image` object is still reachable.

> **NOTE:** If you observe an unending repetition of the `Drawing image` message, perhaps your Java virtual machine's heap space is larger than the heap space used by my virtual machine when I ran this application on my Windows XP platform. If your virtual machine's heap space is large enough, soft references will not be cleared and you will end up with an infinite loop of output. To correct this situation, you might want to increase the size of `Image`'s `image` array (perhaps from `1024*100` to `1024*500`) and (possibly) assign a larger value to `NUM_IMAGES` (perhaps `500`).

WeakReference

The `WeakReference` class describes a `Reference` object whose referent is weakly reachable. In addition to inheriting `Reference`'s methods, this generic class provides the following constructors for initializing a `WeakReference` object:

- `WeakReference(T r)` encapsulates `r`'s reference. The `WeakReference` object behaves as a weak reference to `r`. No `ReferenceQueue` object is associated with this `WeakReference` object.
- `WeakReference(T r, ReferenceQueue<? super T> q)` encapsulates `r`'s reference. The `WeakReference` object behaves as a weak reference to `r`. The `ReferenceQueue` object identified by `q` is associated with this `WeakReference` object. Passing `null` to `q` indicates a weak reference without a queue.

`WeakReference` is useful for preventing memory leaks related to hashmaps. A memory leak occurs when you keep adding objects to a hashmap and never remove them. The objects remain in memory because the hashmap stores strong references to them.

Ideally, the objects should only remain in memory when they are strongly referenced from elsewhere in the application. When an object's last strong reference (apart from hashmap strong references) disappears, the object should be garbage collected.

This situation can be remedied by storing weak references to hashmap entries so they are discarded when no strong references to their keys exist. Java's `WeakHashmap` class (discussed in Chapter 8) accomplishes this task.

PhantomReference

The `PhantomReference` class describes a `Reference` object whose referent is phantom reachable. In addition to inheriting `Reference`'s methods and overriding `get()`, this generic class provides a single constructor for initializing a `PhantomReference` object:

- `PhantomReference(T r, ReferenceQueue<? super T> q)` encapsulates `r`'s reference. The `PhantomReference` object behaves as a phantom reference to `r`. The `ReferenceQueue` object identified by `q` is associated with this `PhantomReference` object. Passing `null` to `q` makes no sense because `get()` is overridden to return null and the `PhantomReference` object will never be enqueued.

Unlike `WeakReference` and `SoftReference` objects, which are enqueued onto their reference queues when their referents become weakly reachable (before finalization), or sometime after their referents become softly reachable (before finalization), `PhantomReference` objects are enqueued after their referents have been reclaimed.

Although you cannot access a `PhantomReference` object's referent (its `get()` method returns null), this class is useful because enqueuing the `PhantomReference` object tells you exactly when the referent has been removed. Perhaps you want to delay creating a large object until another large object has been removed (to avoid a thrown `java.lang.OutOfMemoryError` object).

`PhantomReference` is also useful as a substitute for *resurrection* (making an unreachable object reachable). Because there is no way to access the referent (`get()` returns null), which is no longer in memory when the `PhantomReference` object is enqueued, the object can be cleaned up during the first garbage collection cycle in which that object was discovered to be phantom reachable. You can then clean up related resources after receiving notification via the `PhantomReference` object's reference queue.

> **NOTE:** Resurrection occurs in the `finalize()` method when you assign `this` to a root-set variable. For example, you might specify `r = this;` within `finalize()` to assign the unreachable object identified as `this` to a class field named `r`.

In contrast, the garbage collector requires at least two garbage collection cycles to determine if an object that overrides `finalize()` can be garbage collected. When the first cycle detects that the object is eligible for garbage collection, it calls `finalize()`. Because this method might have resurrected the object, a second garbage collection cycle is needed to determine if resurrection has happened.

> **CAUTION:** Resurrection has been used to implement object pools that recycle the same objects when these objects are expensive (time-wise) to create (database connection objects are an example). Because resurrection exacts a severe performance penalty, and because the `PhantomReference` class makes resurrection unnecessary, you should avoid using resurrection in your applications.

Listing 6–15 shows how you might use `PhantomReference` to detect the removal of a large object.

Listing 6–15. *Detecting a large object's removal*

```
class LargeObject
{
   private byte[] memory = new byte[1024*1024*50]; // 50 megabytes
}
public class LargeObjectDemo
{
   public static void main(String[] args)
   {
      ReferenceQueue<LargeObject> rq;
      rq = new ReferenceQueue<LargeObject>();
      PhantomReference<LargeObject> pr;
      pr = new PhantomReference<LargeObject>(new LargeObject(), rq);
      int counter = 0;
      int[] x;
      while (rq.poll() == null)
      {
         System.out.println("waiting for large object to be removed");
         if (counter++ == 10)
            x = new int[1024*1024];
      }
      System.out.println("large object removed");
   }
}
```

Listing 6–15 declares a `LargeObject` class whose private `memory` array occupies 50MB. If your Java implementation throws `OutOfMemoryError` when you run this application, you might need to reduce the size of this array.

The `main()` method first creates a `ReferenceQueue` object that describes a queue onto which a subsequently created `PhantomReference` object that contains a `LargeObject` reference will be enqueued.

`main()` next creates the `PhantomReference` object, passing a reference to a newly created `LargeObject` object and a reference to the previously created `ReferenceQueue` object to the constructor.

After initializing a `counter` variable (which determines how many loop iterations pass before another large object is created), and after introducing a local variable named `x` that will hold a strong reference to another large object, `main()` enters a polling loop.

The polling loop begins by calling `poll()` to detect the removal of the `LargeObject` object from memory. As long as this method returns null, meaning that the `LargeObject` object is still in memory, the loop outputs a message and increments `counter`.

When `counter`'s value reaches 10, `x` is assigned an `int`-based array containing one million integer elements. Because the reference stored in `x` is strong, this array will not be garbage collected (before the application ends).

On my platform, assigning this array's reference to `x` is sufficient for the garbage collector to destroy the `LargeObject` object. Its `PhantomReference` object is enqueued onto the `rq`-referenced `ReferenceQueue`; `poll()` returns the `PhantomReference` object.

Depending on your implementation of the virtual machine, you might or might not observe the `large object removed` message. If you do not see this message, you might need to increase the size of array `x`, making sure that `OutOfMemoryError` is not thrown.

When I run this application on my platform, I observe the following output—you may have to adjust the application's code to observe similar output:

```
waiting for large object to be removed
waiting for large object to be removed
waiting for large object to be removed
waiting for large object to be removed
waiting for large object to be removed
waiting for large object to be removed
waiting for large object to be removed
waiting for large object to be removed
waiting for large object to be removed
waiting for large object to be removed
waiting for large object to be removed
large object removed
```

> **NOTE:** For a more useful example of `PhantomReference`, and for more in-depth knowledge of garbage collection, check out Keith D Gregory's "Java Reference Objects" blog post (`http://www.kdgregory.com/index.php?page=java.refobj`).

EXERCISES

The following exercises are designed to test your understanding of Java's basic APIs:

1. What constants does `Math` declare?
2. Why is `Math.abs(Integer.MIN_VALUE)` equal to `Integer.MIN_VALUE`?
3. What does `Math`'s `random()` method accomplish?
4. Identify the five special values that can arise during floating-point calculations.
5. How do `Math` and `StrictMath` differ?
6. What is the purpose of `strictfp`?
7. What is `BigDecimal` and why might you use this class?
8. Which `RoundingMode` constant describes the form of rounding commonly taught at school?
9. What is `BigInteger`?
10. What is the purpose of `Package`'s `isSealed()` method?
11. True or false: `getPackage()` requires at least one classfile to be loaded from the package before it returns a `Package` object describing that package.
12. Identify the two main uses of the primitive wrapper classes.

13. Why should you avoid coding expressions such as `ch >= '0' && ch <= '9'` (test `ch` to see if it contains a digit) or `ch >= 'A' && ch <= 'Z'` (test `ch` to see if it contains an uppercase letter)?

14. Identify the four kinds of reachability.

15. What is a referent?

16. Which of the References API's classes is the equivalent of `Object`'s `finalize()` method?

17. Before the era of graphics screens, developers sometimes used a text-based screen to display graphics shapes. For example, a circle might be displayed as follows:

```
          *
      *********
     **       **
   **           **
   *             *
  *               *
 **               **
 *                 *
 *                 *
 *                 *
**                 **
 *                 *
 *                 *
 *                 *
 **               **
  *               *
   *             *
   **           **
     **       **
      *********
          *
```

NOTE: This shape appears elliptical instead of circular because each asterisk's displayed height is greater than its displayed width. If the height and width matched, the shape would appear circular.

Create a `Circle` application that generates and displays the previous circle shape. Start by creating a two-dimensional `screen` array of 22 rows by 22 columns. Initialize each array element to the space character (indicating a clear screen). For each integer angle from 0 to 360, compute the x and y coordinates by multiplying a radius value of 10 by each of the cosine and sine of the angle. Add 11 to the x value and 11 to the y value to center the circle shape within the `screen` array. Assign an asterisk to the array at the resulting (x, y) coordinates. After the loop completes, output the array to the standard output device.

18. A *prime number* is a positive integer greater than 1 that is evenly divisible only by 1 and itself. Create a `PrimeNumberTest` application that determines if its solitary integer argument is prime or not prime, and outputs a suitable message. For example, `java PrimeNumberTest 289` should output the message `289 is not prime`. A simple way to check for primality is to loop from 2 through the square root of the integer argument, and use the remainder operator in the loop to determine if the argument is divided evenly by the loop index. For example, because `6%2` yields a remainder of 0 (2 divides evenly into 6), integer 6 is not a prime number.

Summary

The `java.lang.Math` class supplements the basic math operations (`+`, `-`, `*`, `/`, and `%`) with advanced operations (such as trigonometry). The companion `java.lang.StrictMath` class ensures that all of these operations yield the same values on all platforms.

Money must never be represented by floating-point and double precision floating-point variables because not all monetary values can be represented exactly. In contrast, the `java.math.BigDecimal` class lets you accurately represent and manipulate these values.

`BigDecimal` relies on the `java.math.BigInteger` class for representing its unscaled value. A `BigInteger` instance describes an integer value that can be of arbitrary length (subject to the limits of the virtual machine's memory).

The `java.lang.Package` class provides access to package information. This information includes version information about the implementation and specification of a Java package, the package's name, and an indication of whether the package is sealed or not.

Instances of the `java.lang` package's `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, and `Short` primitive wrapper classes wrap themselves around values of primitive types. These classes are useful for storing primitive values in collections.

The References API makes it possible for an application to interact with the garbage collector in limited ways. This API's `java.lang.ref` package contains classes `Reference`, `ReferenceQueue`, `SoftReference`, `WeakReference`, and `PhantomReference`.

`SoftReference` is useful for implementing image caches, `WeakReference` is useful for preventing memory leaks related to hashmaps, and `PhantomReference` is useful for learning when an object has died so its resources can be cleaned up.

Your exploration of Java's basic APIs is far from finished. Chapter 7 continues to focus on basic APIs by discussing the Reflection API, string management, the `System` class, and the low-level Threading API.