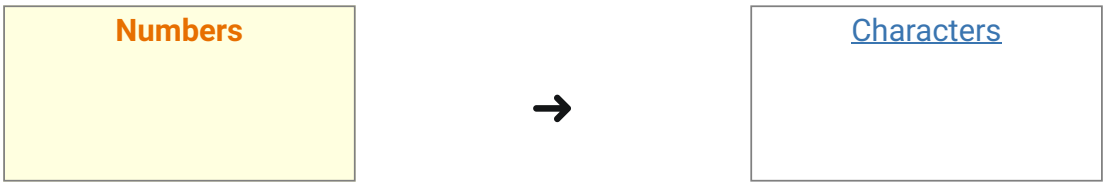


Numbers

Numbers

This section begins with a discussion of the `Number` class in the `java.lang` package, its subclasses, and the situations where you would use instantiations of these classes rather than the primitive number types.

This section also presents the `PrintStream` and `DecimalFormat` classes, which provide methods for writing formatted numerical output.

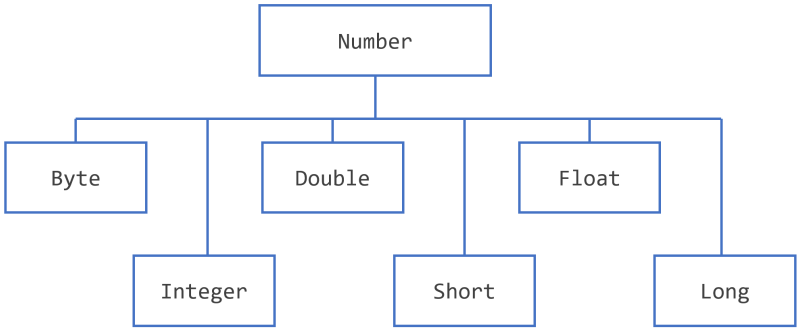
Finally, the `Math` class in `java.lang` is discussed. It contains mathematical functions to complement the operators built into the language. This class has methods for the trigonometric functions, exponential functions, and so forth.

When working with numbers, most of the time you use the primitive types in your code. For example:

```
int i = 500;
float gpa = 3.65f;
byte mask = 0x7f;
```

There are, however, reasons to use objects in place of primitives, and the Java platform provides wrapper classes for each of the primitive data types. These classes "wrap" the primitive in an object. Often, the wrapping is done by the compiler—if you use a primitive where an object is expected, the compiler boxes the primitive in its wrapper class for you. Similarly, if you use a number object when a primitive is expected, the compiler unboxes the object for you. For more information, see the section Autoboxing and Unboxing

All of the numeric wrapper classes are subclasses of the abstract class `Number`:



The Number Class Hierarchy

> *Note: There are four other subclasses of `Number` that are not discussed here. `BigDecimal` and `BigInteger` are used for high-precision calculations. `AtomicInteger` and `AtomicLong` are used for multi-threaded applications.*

There are three reasons that you might use a `Number` object rather than a primitive:

1. As an argument of a method that expects an object (often used when manipulating collections of numbers).
2. To use constants defined by the class, such as `MIN_VALUE` and `MAX_VALUE`, that provide the upper and lower bounds of the data type.
3. To use class methods for converting values to and from other primitive types, for converting to and from strings, and for converting between number systems (decimal, octal, hexadecimal, binary).

The following table lists the instance methods that all the subclasses of the Number class implement.

The following methods convert the value of this `Number` object to the primitive data type returned.

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

The following methods compare this `Number` object to the argument.

- `int compareTo(Byte anotherByte)`
- `int compareTo(Double anotherDouble)`
- `int compareTo(Float anotherFloat)`
- `int compareTo(Integer anotherInteger)`
- `int compareTo(Long anotherLong)`
- `int compareTo(Short anotherShort)`
- `boolean equals(Object obj)`

The method `equals(Object obj)` determines whether this number object is equal to the argument. The methods return `true` if the argument is not `null` and is an object of the same type and with the same numeric value. There are some extra requirements for `Double` and `Float` objects that are described in the Java API documentation.

Each `Number` class contains other methods that are useful for converting numbers to and from strings and for converting between number systems. The following table lists these methods in the `Integer` class. Methods for the other `Number` subclasses are similar:

Method	Description
`static Integer decode(String s)`	Decodes a string into an integer. Can accept string representations of decimal, octal, or hexadecimal numbers as input.
`static int parseInt(String s)`	Returns an integer (decimal only).
`static int parseInt(String s, int radix)`	Returns an integer, given a string representation of decimal, binary, octal, or hexadecimal (radix equals 10, 2, 8, or 16 respectively) numbers as input.
`String toString()`	Returns a `String` object representing the value of this `Integer`.
`static String toString(int i)`	Returns a `String` object representing the specified integer.
`static Integer valueOf(int i)`	Returns an `Integer` object holding the value of the specified primitive.

Method	Description
`static Integer valueOf(String s)`	Returns an `Integer` object holding the value of the specified string representation.
`static Integer valueOf(String s, int radix)`	Returns an `Integer` object holding the integer value of the specified string representation, parsed with the value of radix. For example, if s = "333" and radix = 8, the method returns the base-ten integer equivalent of the octal number 333.

Formatting Numeric Print Output

Earlier you saw the use of the `print` and `println` methods for printing strings to standard output `System.out`. Since all numbers can be converted to strings, you can use these methods to print out an arbitrary mixture of strings and numbers. The Java programming language has other methods, however, that allow you to exercise much more control over your print output when numbers are included.

The Printf and Format Methods

The `java.io` package includes a `PrintStream` class that has two formatting methods that you can use to replace `print` and `println`. These methods, `format` and `printf`, are equivalent to one another. The familiar `System.out` that you have been using happens to be a `PrintStream` object, so you can invoke `PrintStream` methods on `System.out`. Thus, you can use `format` or `printf` anywhere in your code where you have previously been using `print` or `println`. For example,

```
System.out.format(.....);
```

The syntax for these two `java.io.PrintStream` methods is the same:

```
public PrintStream format(String format, Object... args)
```

where `format` is a string that specifies the formatting to be used and args is a list of the variables to be printed using that formatting. A simple example would be

```
System.out.format("The value of " + "the float variable is " +
     "%f, while the value of the " + "integer variable is %d, " +
     "and the string is %s", floatVar, intVar, stringVar);
```

The first parameter, `format`, is a format string specifying how the objects in the second parameter, `args`, are to be formatted. The `format` string contains plain text as well as format specifiers, which are special characters that format the arguments of `Object...` args. (The notation `Object...` args is called *varargs*, which means that the number of arguments may vary.)

Format specifiers begin with a percent sign (`%`) and end with a converter. The converter is a character indicating the type of argument to be formatted. In between the percent sign (`%`) and the converter you can have optional flags and specifiers. There are many converters, flags, and specifiers, which are documented in `java.util.Formatter`.

Here is a basic example:

```
int i = 461012;
System.out.format("The value of i is: %d%n", i)
```

The `%d` specifies that the single variable is a decimal integer. The `%n` is a platform-independent newline character. The output is:

```
The value of i is: 461012
```

The `printf` and `format` methods are overloaded. Each has a version with the following syntax:

```
public PrintStream format(Locale l, String format, Object... args)
```

To print numbers in the French system (where a comma is used in place of the decimal place in the English representation of floating point numbers), for example, you would use:

```
System.out.format(Locale.FRANCE,
    "The value of the float " + "variable is %f, while the " +
    "value of the integer variable " + "is %d, and the string is %s%n",
    floatVar, intVar, stringVar);
```

An Example

The following table lists some of the converters and flags that are used in the sample program, `TestFormat.java`, that follows the table.

Converter	Flag	Explanation
d		A decimal integer.
f		A float.
n		A new line character appropriate to the platform running the application. You should always use `%n`, rather than `\n`.
tB		A date & time conversion—locale-specific full name of month.
td, te		A date & time conversion—2-digit day of month. td has leading zeroes as needed, te does not.
ty, tY		A date & time conversion—ty = 2-digit year, tY = 4-digit year.
tl		A date & time conversion—hour in 12-hour clock.
tM		A date & time conversion—minutes in 2 digits, with leading zeroes as necessary.
tp		A date & time conversion—locale-specific am/pm (lower case).
tm		A date & time conversion—months in 2 digits, with leading zeroes as necessary.
tD		A date & time conversion—date as %tm%td%ty
	08	Eight characters in width, with leading zeroes as necessary.
	+	Includes sign, whether positive or negative.
	,	Includes locale-specific grouping characters.
	-	Left-justified.
	.3	Three places after decimal point.
	10.3	Ten characters in width, right justified, with three places after decimal point.

The following program shows some of the formatting that you can do with format. The output is shown within double quotes in the embedded comment:

```
import java.util.Calendar;
import java.util.Locale;

public class TestFormat {

    public static void main(String[] args) {
      long n = 461012;
      System.out.format("%d%n", n);       //  -->  "461012"
      System.out.format("%08d%n", n);     //  -->  "00461012"
      System.out.format("%+8d%n", n);     //  -->  " +461012"
      System.out.format("%,8d%n", n);     // -->  " 461,012"
      System.out.format("%+,8d%n%n", n); //  -->  "+461,012"

      double pi = Math.PI;

      System.out.format("%f%n", pi);        // -->  "3.141593"
      System.out.format("%.3f%n", pi);      // -->  "3.142"
      System.out.format("%10.3f%n", pi);    // -->  "     3.142"
      System.out.format("%-10.3f%n", pi);   // -->  "3.142"
      System.out.format(Locale.FRANCE,
                        "%-10.4f%n%n", pi); // -->  "3,1416"

      Calendar c = Calendar.getInstance();
      System.out.format("%tB %te, %tY%n", c, c, c); // -->  "May 29, 2006"

      System.out.format("%tl:%tM %tp%n", c, c, c);  // -->  "2:34 am"

      System.out.format("%tD%n", c);    // -->  "05/29/06"
    }
}
```

> *Note: The discussion in this section covers just the basics of the `format` and `printf` methods. Further detail can be found in the Basic I/O section of this tutorial, in the "Formatting" page. Using the `String.format()` to create strings is covered in `Strings`.*

The DecimalFormat Class

You can use the `java.text.DecimalFormat` class to control the display of leading and trailing zeros, prefixes and suffixes, grouping (thousands) separators, and the decimal

separator. `DecimalFormat` offers a great deal of flexibility in the formatting of numbers, but it can make your code more complex.

The example that follows creates a `DecimalFormat` object, `myFormatter`, by passing a pattern string to the `DecimalFormat` constructor. The `format` method, which `DecimalFormat` inherits from `NumberFormat`, is then invoked by `myFormatter`—it accepts a double value as an argument and returns the formatted number in a string.

Here is a sample program that illustrates the use of `DecimalFormat`:

```
import java.text.*;

public class DecimalFormatDemo {

   static public void customFormat(String pattern, double value ) {
      DecimalFormat myFormatter = new DecimalFormat(pattern);
      String output = myFormatter.format(value);
      System.out.println(value + "  " + pattern + "  " + output);
   }

   static public void main(String[] args) {

      customFormat("###,###.###", 123456.789);
      customFormat("###.##", 123456.789);
      customFormat("000000.000", 123.78);
      customFormat("$###,###.###", 12345.67);
   }
}
```

The output is:

```
123456.789  ###,###.###  123,456.789
123456.789  ###.##  123456.79
123.78  000000.000  000123.780
12345.67  $###,###.###  $12,345.67
```

The following table explains each line of output.

Value	Pattern	Output	Explanation
123456.789	###,###.###	123,456.789	The pound sign (#) denotes a digit, the comma is a placeholder for the grouping separator, and the period is a placeholder for the decimal separator.

Value	Pattern	Output	Explanation
123456.789	###.##	123456.79	The `value` has three digits to the right of the decimal point, but the pattern has only two. The format method handles this by rounding up.
123.78	000000.000	000123.780	The `pattern` specifies leading and trailing zeros, because the 0 character is used instead of the pound sign (#).
12345.67	$###,###.###	$12,345.67	The first character in the `pattern` is the dollar sign (`$`). Note that it immediately precedes the leftmost digit in the formatted `output`.

Beyond Basic Arithmetic

The Java programming language supports basic arithmetic with its arithmetic operators: `+`, `-`, `*`, `/`, and `%`. The `Math` class in the `java.lang` package provides methods and constants for doing more advanced mathematical computation.

The methods in the `Math` class are all static, so you call them directly from the class, like this:

```
1 | Math.cos(angle);
```

> *Note: Using the static import language feature, you don't have to write `Math` in front of every math function: `import static java.lang.Math.*;` This allows you to invoke the `Math` class methods by their simple names. For example: `cos(angle);`*

Constants and Basic Methods

The `Math` class includes two constants:

- `Math.E`, which is the base of natural logarithms, and
- `Math.PI`, which is the ratio of the circumference of a circle to its diameter.

The `Math` class also includes more than 40 static methods. The following table lists a number of the basic methods.

Computing an Absolute Value

- `double abs(double d)`
- `float abs(float f)`
- `int abs(int i)`
- `long abs(long lng)`

Rouding a Value

- `double ceil(double d)`: Returns the smallest integer that is greater than or equal to the argument. Returned as a `double`.
- `double floor(double d)`: Returns the largest integer that is less than or equal to the argument. Returned as a `double`.
- `double rint(double d)`: Returns the integer that is closest in value to the argument. Returned as a `double`.
- `long round(double d)` and `int round(float f)`: Returns the closest `long` or `int`, as indicated by the method's return type, to the argument.

Computing a Min

- `double min(double arg1, double arg2)`
- `float min(float arg1, float arg2)`
- `int min(int arg1, int arg2)`
- `long min(long arg1, long arg2)`

Computing a Max

- `double max(double arg1, double arg2)`
- `float max(float arg1, float arg2)`
- `int max(int arg1, int arg2)`
- `long max(long arg1, long arg2)`

The following program, `BasicMathDemo`, illustrates how to use some of these methods:

```
public class BasicMathDemo {
    public static void main(String[] args) {
        double a = -191.635;
        double b = 43.74;
        int c = 16, d = 45;

        System.out.printf("The absolute value " + "of %.3f is %.3f%n",
                          a, Math.abs(a));

        System.out.printf("The ceiling of " + "%.2f is %.0f%n",
                          b, Math.ceil(b));

        System.out.printf("The floor of " + "%.2f is %.0f%n",
                          b, Math.floor(b));

        System.out.printf("The rint of %.2f " + "is %.0f%n",
                          b, Math.rint(b));

        System.out.printf("The max of %d and " + "%d is %d%n",
                          c, d, Math.max(c, d));

        System.out.printf("The min of of %d " + "and %d is %d%n",
                          c, d, Math.min(c, d));
    }
}
```

Here's the output from this program:

```
The absolute value of -191.635 is 191.635
The ceiling of 43.74 is 44
The floor of 43.74 is 43
The rint of 43.74 is 44
The max of 16 and 45 is 45
The min of 16 and 45 is 16
```

Exponential and Logarithmic Methods

The next table lists exponential and logarithmic methods of the `Math` class.

- `double exp(double d)`: Returns the base of the natural logarithms, e, to the power of the argument.
- `double log(double d)`: Returns the natural logarithm of the argument.

- `double pow(double base, double exponent)`: Returns the value of the first argument raised to the power of the second argument.
- `double sqrt(double d)`: Returns the square root of the argument.

The following program, `ExponentialDemo`, displays the value of `e`, then calls each of the methods listed in the previous table on arbitrarily chosen numbers:

```
public class ExponentialDemo {
    public static void main(String[] args) {
        double x = 11.635;
        double y = 2.76;

        System.out.printf("The value of " + "e is %.4f%n",
                          Math.E);

        System.out.printf("exp(%.3f) " + "is %.3f%n",
                          x, Math.exp(x));

        System.out.printf("log(%.3f) is " + "%.3f%n",
                          x, Math.log(x));

        System.out.printf("pow(%.3f, %.3f) " + "is %.3f%n",
                          x, y, Math.pow(x, y));

        System.out.printf("sqrt(%.3f) is " + "%.3f%n",
                          x, Math.sqrt(x));
    }
}
```

Here is the output you will see when you run `ExponentialDemo`:

```
The value of e is 2.7183
exp(11.635) is 112983.831
log(11.635) is 2.454
pow(11.635, 2.760) is 874.008
sqrt(11.635) is 3.411
```

Trigonometric Methods

The `Math` class also provides a collection of trigonometric functions, which are summarized in the following table. The value passed into each of these methods is an angle expressed in radians. You can use the `toRadians(double d)` method to convert from degrees to radians.

- `double sin(double d)`: Returns the sine of the specified double value.
- `double cos(double d)`: Returns the cosine of the specified double value.
- `double tan(double d)`: Returns the tangent of the specified double value.
- `double asin(double d)`: Returns the arcsine of the specified double value.
- `double acos(double d)`: Returns the arccosine of the specified double value.
- `double atan(double d)`: Returns the arctangent of the specified double value.
- `double atan2(double y, double x)`: Converts rectangular coordinates (x, y) to polar coordinate (r, theta) and returns theta.
- `double toDegrees(double d)` and `double toRadians(double d)`: Converts the argument to degrees or radians.

Here is a program, `TrigonometricDemo`, that uses each of these methods to compute various trigonometric values for a 45-degree angle:

```
public class TrigonometricDemo {
    public static void main(String[] args) {
        double degrees = 45.0;
        double radians = Math.toRadians(degrees);

        System.out.format("The value of pi " + "is %.4f%n",
                          Math.PI);

        System.out.format("The sine of %.1f " + "degrees is %.4f%n",
                          degrees, Math.sin(radians));

        System.out.format("The cosine of %.1f " + "degrees is %.4f%n",
                          degrees, Math.cos(radians));

        System.out.format("The tangent of %.1f " + "degrees is %.4f%n",
                          degrees, Math.tan(radians));

        System.out.format("The arcsine of %.4f " + "is %.4f degrees %n",
                          Math.sin(radians),
                          Math.toDegrees(Math.asin(Math.sin(radians))));

        System.out.format("The arccosine of %.4f " + "is %.4f degrees %n",
                          Math.cos(radians),
                          Math.toDegrees(Math.acos(Math.cos(radians))));

        System.out.format("The arctangent of %.4f " + "is %.4f degrees %n",
                          Math.tan(radians),
```

```
                        Math.toDegrees(Math.atan(Math.tan(radians))));
    }
}
```

The output of this program is as follows:

```
The value of pi is 3.1416
The sine of 45.0 degrees is 0.7071
The cosine of 45.0 degrees is 0.7071
The tangent of 45.0 degrees is 1.0000
The arcsine of 0.7071 is 45.0000 degrees
The arccosine of 0.7071 is 45.0000 degrees
The arctangent of 1.0000 is 45.0000 degrees
```

Random Numbers

The `random()` method returns a pseudo-randomly selected number between 0.0 and 1.0. The range includes 0.0 but not 1.0. In other words: `0.0 <= Math.random() < 1.0`. To get a number in a different range, you can perform arithmetic on the value returned by the random method. For example, to generate an integer between 0 and 9, you would write:

```
int number = (int)(Math.random() * 10);
```

By multiplying the value by 10, the range of possible values becomes `0.0 <= number < 10.0`.

Using `Math.random` works well when you need to generate a single random number. If you need to generate a series of random numbers, you should create an instance of `java.util.Random` and invoke methods on that object to generate numbers.

Last update: *September 14, 2021*