Numbers & Strings

Numbers

This section begins with a discussion of the Number class (in the java.lang package) and its subclasses. In particular, this section talks about the situations where you would use instantiations of these classes rather than the primitive data types. Additionally, this section talks about other classes you might need to work with numbers, such as formatting or using mathematical functions to complement the operators built into the language. Finally, there is a discussion on autoboxing and unboxing, a compiler feature that simplifies your code.

Strings

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects. This section describes using the String class to create and manipulate strings. It also compares the String and StringBuilder classes.

@Komina Todolo

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.

The Numbers Classes

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

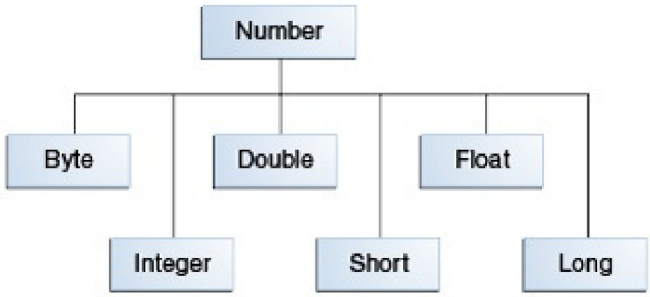
Example:

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

Using objects in place of primitives:

1. The Java platform provides wrapper classes for each of the primitive data types.
2. These classes "wrap" the primitive in an object.
3. Wrapping is done by the compiler
4. 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.
5. Autoboxing and Unboxing

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



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 3 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).

@Komina Todolo

Numbers

The Numbers Classes, cont.

Table lists the instance methods that all the *subclasses* of the Number class implement.

Methods Implemented by all Subclasses of Number

Method	Description

`byte byteValue()`
`short shortValue()`
`int intValue()`
`long longValue()`
`float floatValue()`
`double doubleValue()`
	Converts the value of this `Number` object to the primitive data type returned.

`int compareTo(Byte anotherByte)`
`int compareTo(Double anotherDouble)`
`int compareTo(Float anotherFloat)`
`int compareTo(Integer anotherInteger)`
`int compareTo(Long anotherLong)`
`int compareTo(Short anotherShort)`
	Compares this `Number` object to the argument.

`boolean 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*.

This table lists these methods in the Integer class.

Conversion Methods, `Integer` Class

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.
`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.

Methods for the other Number subclasses are similar.

@Komira Todoo

Numbers

Formatting Numeric Print Output

Earlier Example: Use of the print and println methods for printing strings to standard output (System.out).
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.

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

The syntax for these two **java.io.PrintStream** methods are 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.

Example:

```
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 output is:

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

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

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:

```
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);
```

@Kamina Todolo

Numbers

Example

Formatting Numeric Print Output:

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

Converters and Flags Used in `TestFormat.java`

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.

@Komisa Lodolo

Numbers

Formatting Numeric Print Output, cont:

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 the Essential trail, in the "Formatting" page. Using String.format to create strings is covered in Strings.

@Komina Lodolo

Numbers

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:

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
```

Table explains each line of output:

DecimalFormat.java 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.
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`.

@Komina Jodolo

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: `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. 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.

Basic Math Methods

Method	Description

`double abs(double d)`
`float abs(float f)`
`int abs(int i)`
`long abs(long lng)`
	Returns the absolute value of the argument.

`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)` `int round(float f)`	Returns the closest long or int, as indicated by the method's return type, to the argument.

`double min(double arg1, double arg2)`
`float min(float arg1, float arg2)`
`int min(int arg1, int arg2)`
`long min(long arg1, long arg2)`
	Returns the smaller of the two arguments.

`double max(double arg1, double arg2)`
`float max(float arg1, float arg2)`
`int max(int arg1, int arg2)`
`long max(long arg1, long arg2)`
	Returns the larger of the two arguments.

@Kanisa Lodolo

Beyond Basic Arithmetic

Constants and Basic Methods cont.

```
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));
    }
}
```

The output is:

```
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
```

@Kamina Todolo

Beyond Basic Arithmetic

Exponential and Logarithmic Methods

This table lists exponential and logarithmic methods of the Math class.

Exponential and Logarithmic Methods

Method	Description
`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.

```
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));
    }
}
```

The output is:

```
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
```

@Kamina Todolo

Beyond Basic Arithmetic

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 method to convert from degrees to radians.

Trigonometric Methods

Method	Description
`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)` `double toRadians(double d)`	Converts the argument to degrees or radians.

```
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 is:

```
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
```

@Kamina Todolo

Numbers

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.

Example, generating an integer between 0 and 9: `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.

Summary of Numbers

You use one of the wrapper classes `Byte, Double, Float, Integer, Long, or Short` to wrap a number of primitive type in an object.

The Java compiler automatically wraps (boxes) primitives for you when necessary and unboxes them, again when necessary.

The Number classes include constants and useful class methods. The MIN_VALUE and MAX_VALUE constants contain the smallest and largest values that can be contained by an object of that type. The byteValue, shortValue, and similar methods convert one numeric type to another. The valueOf method converts a string to a number, and the toString method converts a number to a string.

To format a string containing numbers for output, you can use the printf() or format() methods in the PrintStream class. Alternatively, you can use the NumberFormat class to customize numerical formats using patterns.

The Math class contains a variety of class methods for performing mathematical functions, including exponential, logarithmic, and trigonometric methods. Math also includes basic arithmetic functions, such as absolute value and rounding, and a method, *random()*, for generating random numbers.

@Komina Lodolo

Characters

If you are using a single character value, you will use the primitive char type.

Example:

```
char ch = 'a';
// Unicode for uppercase Greek omega character
char uniChar = '\u03A9';
// an array of chars
char[] charArray = { 'a', 'b', 'c', 'd', 'e' };
```

There are times, however, when you need to use a char as an object—for example, as a method argument where an object is expected. The Java programming language provides a wrapper class that "wraps" the char in a Character object for this purpose. An object of type Character contains a single field, whose type is char. This Character class also offers a number of useful class (that is, static) methods for manipulating characters.

You can create a Character object with the Character constructor:

```
Character ch = new Character('a');
```

The Java compiler will also create a Character object for you under some circumstances. For example, if you pass a primitive char into a method that expects an object, the compiler automatically converts the char to a Character for you. This feature is called autoboxing—or unboxing, if the conversion goes the other way.

The Character class is immutable, so that once it is created, a Character object cannot be changed.

This table lists some of the most useful methods in the Character class, but is not exhaustive.
There's +50 of all methods in this class - refer to the *java.lang.Character* API specification

Useful Methods in the `Character` Class

Method	Description
`boolean isLetter(char ch)` `boolean isDigit(char ch)`	Determines whether the specified char value is a letter or a digit, respectively.
`boolean isWhitespace(char ch)`	Determines whether the specified char value is white space.
`boolean isUpperCase(char ch)` `boolean isLowerCase(char ch)`	Determines whether the specified char value is uppercase or lowercase, respectively.
`char toUpperCase(char ch)` `char toLowerCase(char ch)`	Returns the uppercase or lowercase form of the specified char value.
`toString(char ch)`	Returns a `String` object representing the specified character value — that is, a one-character string.

Escape Sequences

A character preceded by a backslash (\) is an escape sequence and has special meaning to the compiler. The following table shows the Java escape sequences:

When an escape sequence is encountered in a print statement, the compiler interprets it accordingly.

Example:

Putting quotes within quotes - use the escape sequence, \", on the interior quotes.

Printing this sentence

```
She said "Hello!" to me.
```

would be written like

```
System.out.println("She said \"Hello!\" to me.");
```

Escape Sequences

Escape Sequence	Description
`\t`	Insert a tab in the text at this point.
`\b`	Insert a backspace in the text at this point.
`\n`	Insert a newline in the text at this point.
`\r`	Insert a carriage return in the text at this point.
`\f`	Insert a form feed in the text at this point.
`\'`	Insert a single quote character in the text at this point.
`\"`	Insert a double quote character in the text at this point.
`\\`	Insert a backslash character in the text at this point.

Strings

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects.
The Java platform provides the String class to create and manipulate strings.

Creating Strings

The most direct way to create a string is to write: `String greeting = "Hello world!";`

In this case, "Hello world!" is a string literal—a series of characters in your code that is enclosed in double quotes. Whenever it encounters a string literal in your code, the compiler creates a String object with its value—in this case, Hello world!.

As with any other object, you can create String objects by using the new keyword and a constructor. The String class has thirteen constructors that allow you to provide the initial value of the string using different sources, such as an array of characters:

```
char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };
String helloString = new String(helloArray);
System.out.println(helloString);
```

The last line of this code snippet displays ***hello***.

Note: The String class is immutable, so that once it is created a String object cannot be changed. The String class has a number of methods, some of which will be discussed below, that appear to modify strings. Since strings are immutable, what these methods really do is create and return a new string that contains the result of the operation.

String Length

Methods used to obtain information about an object are known as accessor methods. One accessor method that you can use with strings is the length() method, which returns the number of characters contained in the string object. After the following two lines of code have been executed, len equals 17:

```
String palindrome = "Dot saw I was Tod";
int len = palindrome.length();
```

A palindrome is a word or sentence that is symmetric—it is spelled the same forward and backward, ignoring case and punctuation. Here is a short and inefficient program to reverse a palindrome string. It invokes the String method charAt(i), which returns the ith character in the string, counting from 0.

```
public class StringDemo {
    public static void main(String[] args) {
        String palindrome = "Dot saw I was Tod";
        int len = palindrome.length();
        char[] tempCharArray = new char[len];
        char[] charArray = new char[len];

        // put original string in an
        // array of chars
        for (int i = 0; i < len; i++) {
            tempCharArray[i] =
                palindrome.charAt(i);
        }

        // reverse array of chars
        for (int j = 0; j < len; j++) {
            charArray[j] =
                tempCharArray[len - 1 - j];
        }

        String reversePalindrome =
            new String(charArray);
        System.out.println(reversePalindrome);
    }
}
```

Running the program produces this output:

```
doT saw I was toD
```

String reversal:
the program had to convert the string to an array of characters (first for loop), reverse the array into a second array (second for loop), and then convert back to a string. The String class includes a method, getChars(), to convert a string, or a portion of a string, into an array of characters so we could replace the first for loop in the program above with

```
palindrome.getChars(0, len, tempCharArray, 0);
```

@Kavisha Todolo

Strings

Concatenating Strings

The String class includes a method for *concatenating* two strings:

```
string1.concat(string2);
```

This returns a new string that is string1 with string2 added to it at the end.

You can also use the concat() method with string literals, as in:

```
"My name is ".concat("Rumplestiltskin");
```

Strings are more commonly concatenated with the + operator: `"Hello," + " world" + "!"`

Result: `"Hello, world!"`

The **+** operator is widely used in print statements.

For example:

```
String string1 = "saw I was ";
System.out.println("Dot " + string1 + "Tod");
```

Such a concatenation can be a mixture of any objects. For each object that is not a String, its toString() method is called to convert it to a String.

Output: `Dot saw I was Tod`

NOTE

The Java programming language does not permit literal strings to span lines in source files, so you must use the + concatenation operator at the end of each line in a multi-line string.

Breaking strings between lines using the + concatenation operator is, once again, very common in print statements.

Example:

```
String quote =
    "Now is the time for all good " +
    "men to come to the aid of their country.";
```

Creating Format Strings

You have seen the use of the printf() and format() methods to print output with formatted numbers. The String class has an equivalent class method, format(), that returns a String object rather than a PrintStream object.

Using String's static format() method allows you to create a formatted string that you can reuse, as opposed to a one-time print statement. For example, instead of

```
System.out.printf("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);
```

you can write

```
String fs;
fs = String.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);
System.out.println(fs);
```

@Kamina Todolo

Converting Between Numbers & Strings

Strings to Numbers

Frequently, a program ends up with numeric data in a string object—a value entered by the user..

The Number subclasses that wrap primitive numeric types (Byte, Integer, Double, Float, Long, and Short) each provide a class method named valueOf that converts a string to an object of that type. Here is an example, ValueOfDemo , that gets two strings from the command line, converts them to numbers, and performs arithmetic operations on the values:

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

        // this program requires two
        // arguments on the command line
        if (args.length == 2) {
            // convert strings to numbers
            float a = (Float.valueOf(args[0])).floatValue();
            float b = (Float.valueOf(args[1])).floatValue();

            // do some arithmetic
            System.out.println("a + b = " +
                               (a + b));
            System.out.println("a - b = " +
                               (a - b));
            System.out.println("a * b = " +
                               (a * b));
            System.out.println("a / b = " +
                               (a / b));
            System.out.println("a % b = " +
                               (a % b));
        } else {
            System.out.println("This program " +
                "requires two command-line arguments.");
        }
    }
}
```

The following is the output from the program when you use 4.5 and 87.2 for the command-line arguments:

```
a + b = 91.7
a - b = -82.7
a * b = 392.4
a / b = 0.0516055
a % b = 4.5
```

NOTE

Each of the Number subclasses that wrap primitive numeric types also provides a parseXXXX() method (for example, parseFloat()) that can be used to convert strings to primitive numbers. Since a primitive type is returned instead of an object, the parseFloat() method is more direct than the valueOf() method.

Example: ValueOfDemo program

```
float a = Float.parseFloat(args[0]);
float b = Float.parseFloat(args[1]);
```

Numbers to Strings

Sometimes you need to convert a number to a string because you need to operate on the value in its string form.
7 easy ways to convert a number to a string:

```
int i;
// Concatenate "i" with an empty
string; conversion is handled for you.
String s1 = "" + i;
```

OR

```
// The valueOf class method.
String s2 = String.valueOf(i);
```

Each of the Number subclasses includes a class method, toString(), that will convert its primitive type to a string. For example:

```
int i;
double d;
String s3 = Integer.toString(i);
String s4 = Double.toString(d);
```

The ToStringDemo example uses the toString method to convert a number to a string. The program then uses some string methods to compute the number of digits before and after the decimal point:

```
public class ToStringDemo {

    public static void main(String[] args) {
        double d = 858.48;
        String s = Double.toString(d);

        int dot = s.indexOf('.');

        System.out.println(dot + " digits " +
            "before decimal point.");
        System.out.println( (s.length() - dot - 1) +
            " digits after decimal point.");
    }
}
```

The output of this program is:

```
3 digits before decimal point.
2 digits after decimal point.
```

@Kamina Todolo

Manipulating Characters in a String

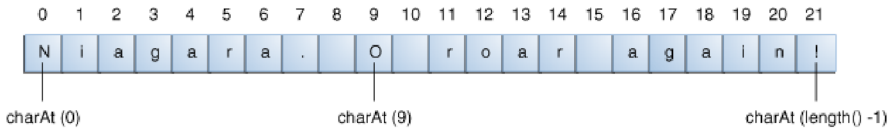
The String class has a number of methods for examining the contents of strings, finding characters or substrings within a string, changing case, and other tasks.

Getting Characters & Substrings by Index

You can get the character at a particular index within a string by invoking the charAt() accessor method. The index of the first character is 0, while the index of the last character is length()-1. For example, the following code gets the character at index 9 in a string:

```
String anotherPalindrome = "Niagara. O roar again!";
char aChar = anotherPalindrome.charAt(9);
```

Indices begin at 0, so the character at index 9 is 'O', as illustrated in the following figure:



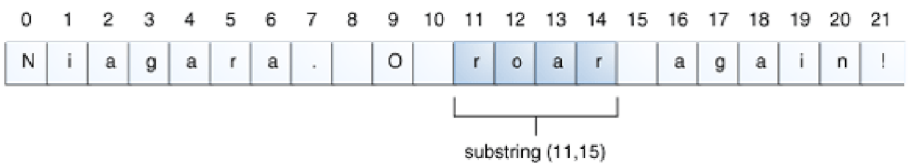
If you want to get more than one consecutive character from a string, you can use the substring method. The substring method has two versions, as shown in the following table:

The `substring` Methods in the `String` Class

Method	Description
`String substring(int beginIndex, int endIndex)`	Returns a new string that is a substring of this string. The substring begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`.
`String substring(int beginIndex)`	Returns a new string that is a substring of this string. The integer argument specifies the index of the first character. Here, the returned substring extends to the end of the original string.

The following code gets from the Niagara palindrome the substring that extends from index 11 up to, but not including, index 15, which is the word "roar":

```
String anotherPalindrome = "Niagara. O roar again!";
String roar = anotherPalindrome.substring(11, 15);
```



Other Methods for Manipulating Strings

7 other String methods for manipulating strings:

Other Methods in the `String` Class for Manipulating Strings

Method	Description
`String[] split(String regex)` `String[] split(String regex, int limit)`	Searches for a match as specified by the string argument (which contains a regular expression) and splits this string into an array of strings accordingly. The optional integer argument specifies the maximum size of the returned array. Regular expressions are covered in the lesson titled "Regular Expressions."
`CharSequence subSequence(int beginIndex, int endIndex)`	Returns a new character sequence constructed from `beginIndex` index up until `endIndex - 1`.
`String trim()`	Returns a copy of this string with leading and trailing white space removed.
`String toLowerCase()` `String toUpperCase()`	Returns a copy of this string converted to lowercase or uppercase. If no conversions are necessary, these methods return the original string.

@Kamina Lodol

Manipulating Characters in a String

Searching for Characters and Substrings in a String

Here are some other String methods for finding characters or substrings within a string. The String class provides accessor methods that return the position within the string of a specific character or substring: indexOf() and lastIndexOf(). The indexOf() methods search forward from the beginning of the string, and the lastIndexOf() methods search backward from the end of the string. If a character or substring is not found, indexOf() and lastIndexOf() return -1.

The String class also provides a search method, contains, that returns true if the string contains a particular character sequence. Use this method when you only need to know that the string contains a character sequence, but the precise location isn't important.

The following table describes the various string search methods:

The Search Methods in the `String` Class

Method	Description
`int indexOf(int ch)` `int lastIndexOf(int ch)`	Returns the index of the first (last) occurrence of the specified character.
`int indexOf(int ch, int fromIndex)` `int lastIndexOf(int ch, int fromIndex)`	Returns the index of the first (last) occurrence of the specified character, searching forward (backward) from the specified index.
`int indexOf(String str)` `int lastIndexOf(String str)`	Returns the index of the first (last) occurrence of the specified substring.
`int indexOf(String str, int fromIndex)` `int lastIndexOf(String str, int fromIndex)`	Returns the index of the first (last) occurrence of the specified substring, searching forward (backward) from the specified index.
`boolean contains(CharSequence s)`	Returns true if the string contains the specified character sequence.

NOTE

CharSequence is an interface that is implemented by the String class. Therefore, you can use a string as an argument for the contains()method.

Replacing Characters and Substrings into a String

The String class has very few methods for inserting characters or substrings into a string. In general, they are not needed: You can create a new string by concatenation of substrings you have removed from a string with the substring that you want to insert.

The String class has 4 methods for replacing found characters or substrings:

Methods in the `String` Class for Manipulating Strings

Method	Description
`String replace(char oldChar, char newChar)`	Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
`String replace(CharSequence target, CharSequence replacement)`	Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence.
`String replaceAll(String regex, String replacement)`	Replaces each substring of this string that matches the given regular expression with the given replacement.
`String replaceFirst(String regex, String replacement)`	Replaces the first substring of this string that matches the given regular expression with the given replacement.

@Kamina Lodolo

Manipulating Characters in a String

Example: The following class, Filename, illustrates the use of lastIndexOf() and substring() to isolate different parts of a file name.

NOTE: The methods in the following Filename class don't do any error checking and assume that their argument contains a full directory path and a filename with an extension. If these methods were production code, they would verify that their arguments were properly constructed.

```
public class Filename {
    private String fullPath;
    private char pathSeparator,
                 extensionSeparator;

    public Filename(String str, char sep, char ext) {
        fullPath = str;
        pathSeparator = sep;
        extensionSeparator = ext;
    }

    public String extension() {
        int dot = fullPath.lastIndexOf(extensionSeparator);
        return fullPath.substring(dot + 1);
    }

    // gets filename without extension
    public String filename() {
        int dot = fullPath.lastIndexOf(extensionSeparator);
        int sep = fullPath.lastIndexOf(pathSeparator);
        return fullPath.substring(sep + 1, dot);
    }

    public String path() {
        int sep = fullPath.lastIndexOf(pathSeparator);
        return fullPath.substring(0, sep);
    }
}
```

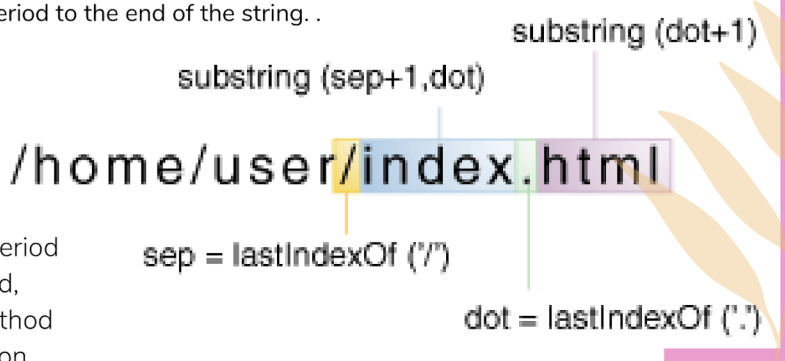
Here is a program, *FilenameDemo*, that constructs a Filename object & calls all of its methods:

```
public class FilenameDemo {
    public static void main(String[] args) {
        final String FPATH = "/home/user/index.html";
        Filename myHomePage = new Filename(FPATH, '/', '.');
        System.out.println("Extension = " + myHomePage.extension());
        System.out.println("Filename = " + myHomePage.filename());
        System.out.println("Path = " + myHomePage.path());
    }
}
```

Output

```
Extension = html
Filename = index
Path = /home/user
```

As shown in the following figure, our extension method uses lastIndexOf to locate the last occurrence of the period (.) in the file name. Then substring uses the return value of lastIndexOf to extract the file name extension — that is, the substring from the period to the end of the string. .



This code assumes that the file name has a period in it; if the file name does not have a period, lastIndexOf returns -1, and the substring method throws a StringIndexOutOfBoundsException

The extension method uses dot + 1 as the argument to substring. If the period character (.) is the last character of the string, dot + 1 is equal to the length of the string, which is one larger than the largest index into the string (because indices start at 0).
This is a *legal* argument to substring because that method accepts an index equal to, but not greater than, the length of the string and interprets it to mean "the end of the string."

@Kenisa Lodolo

Comparing Strings & Portions of Strings

The String class has a number of methods for comparing strings and portions of strings. The following table lists these methods.

Methods for Comparing Strings

Method	Description
`boolean endsWith(String suffix)` `boolean startsWith(String prefix)`	Returns `true` if this string ends with or begins with the substring specified as an argument to the method.
`boolean startsWith(String prefix, int offset)`	Considers the string beginning at the index `offset`, and returns `true` if it begins with the substring specified as an argument.
`int compareTo(String anotherString)`	Compares two strings lexicographically. Returns an integer indicating whether this string is greater than (result is > 0), equal to (result is = 0), or less than (result is < 0) the argument.
`int compareToIgnoreCase(String str)`	Compares two strings lexicographically, ignoring differences in case. Returns an integer indicating whether this string is greater than (result is > 0), equal to (result is = 0), or less than (result is < 0) the argument.
`boolean equals(Object anObject)`	Returns `true` if and only if the argument is a `String` object that represents the same sequence of characters as this object.
`boolean equalsIgnoreCase(String anotherString)`	Returns `true` if and only if the argument is a `String` object that represents the same sequence of characters as this object, ignoring differences in case.
`boolean regionMatches(int toffset, String other, int ooffset, int len)`	Tests whether the specified region of this string matches the specified region of the String argument. Region is of length `len` and begins at the index `toffset` for this string and `ooffset` for the other string.
`boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)`	Tests whether the specified region of this string matches the specified region of the String argument. Region is of length `len` and begins at the index `toffset` for this string and `ooffset` for the other string. The boolean argument indicates whether case should be ignored; if true, case is ignored when comparing characters.
`boolean matches(String regex)`	Tests whether this string matches the specified regular expression. Regular expressions are discussed in the lesson titled "Regular Expressions."

The following program, *RegionMatchesDemo*, uses the regionMatches method to search for a string within another string:

```
public class RegionMatchesDemo {
    public static void main(String[] args) {
        String searchMe = "Green Eggs and Ham";
        String findMe = "Eggs";
        int searchMeLength = searchMe.length();
        int findMeLength = findMe.length();
        boolean foundIt = false;
        for (int i = 0;
             i <= (searchMeLength - findMeLength);
             i++) {
           if (searchMe.regionMatches(i, findMe, 0, findMeLength)) {
              foundIt = true;
              System.out.println(searchMe.substring(i, i + findMeLength));
              break;
           }
        }
        if (!foundIt)
            System.out.println("No match found.");
    }
}
```

The output from this program is `Eggs`

The program steps through the string referred to by searchMe one character at a time. For each character, the program calls the regionMatches method to determine whether the substring beginning with the current character matches the string the program is looking for.

@Kanika Todolo

The StringBuilder Class

StringBuilder objects are like String objects, except that they can be modified. Internally, these objects are treated like variable-length arrays that contain a sequence of characters. At any point, the length and content of the sequence can be changed through method invocations.

Strings should always be used unless string builders offer an advantage in terms of simpler code (see the sample program at the end of this section) or better performance. For example, if you need to concatenate a large number of strings, appending to a StringBuilder object is more efficient.

Length and Capacity

The StringBuilder class, like the String class, has a length() method that returns the length of the character sequence in the builder.

Unlike strings, every string builder also has a capacity, the number of character spaces that have been allocated. The capacity, which is returned by the capacity() method, is always greater than or equal to the length (usually greater than) and will automatically expand as necessary to accommodate additions to the string builder.

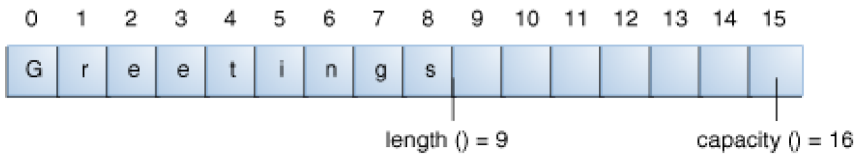
`StringBuilder` Constructors

Constructor	Description
`StringBuilder()`	Creates an empty string builder with a capacity of 16 (16 empty elements).
`StringBuilder(CharSequence cs)`	Constructs a string builder containing the same characters as the specified `CharSequence`, plus an extra 16 empty elements trailing the `CharSequence`.
`StringBuilder(int initCapacity)`	Creates an empty string builder with the specified initial capacity.
`StringBuilder(String s)`	Creates a string builder whose value is initialized by the specified string, plus an extra 16 empty elements trailing the string.

Example:

```
// creates empty builder, capacity 16
StringBuilder sb = new StringBuilder();
// adds 9 character string at beginning
sb.append("Greetings");
```

will produce a string builder with a length of 9 and a capacity of 16



The StringBuilder class has some methods related to length and capacity that the String class does not have:

Length and Capacity Methods

Method	Description
`void setLength(int newLength)`	Sets the length of the character sequence. If `newLength` is less than `length()`, the last characters in the character sequence are truncated. If `newLength` is greater than `length()`, null characters are added at the end of the character sequence.
`void ensureCapacity(int minCapacity)`	Ensures that the capacity is at least equal to the specified minimum.

A number of operations (for example, append(), insert(), or setLength()) can increase the length of the character sequence in the string builder so that the resultant length() would be greater than the current capacity().
When this happens, the capacity is automatically increased.

@Komisa Todolo

The StringBuilder Class

StringBuilder Operations

The principal operations on a StringBuilder that are not available in String are the append() and insert() methods, which are overloaded so as to accept data of any type. Each converts its argument to a string and then appends or inserts the characters of that string to the character sequence in the string builder.

The append method always adds these characters at the end of the existing character sequence, while the insert method adds the characters at a specified point.

Methods of the StringBuilder class:

Various `StringBuilder` Methods

Method	Description

`StringBuilder append(boolean b)`
`StringBuilder append(char c)`
`StringBuilder append(char[] str)`
`StringBuilder append(char[] str, int offset, int len)`
`StringBuilder append(double d)`
`StringBuilder append(float f)`
`StringBuilder append(int i)`
`StringBuilder append(long lng)`
`StringBuilder append(Object obj)`
`StringBuilder append(String s)`
	Appends the argument to this string builder. The data is converted to a string before the append operation takes place.

`StringBuilder delete(int start, int end)` `StringBuilder deleteCharAt(int index)`	The first method deletes the subsequence from start to end-1 (inclusive) in the `StringBuilder`'s char sequence. The second method deletes the character located at `index`.

`StringBuilder insert(int offset, boolean b)`
`StringBuilder insert(int offset, char c)`
`StringBuilder insert(int offset, char[] str)`
`StringBuilder insert(int index, char[] str, int offset, int len)`
`StringBuilder insert(int offset, double d)`
`StringBuilder insert(int offset, float f)`
`StringBuilder insert(int offset, int i)`
`StringBuilder insert(int offset, long lng)`
`StringBuilder insert(int offset, Object obj)`
`StringBuilder insert(int offset, String s)`
	Inserts the second argument into the string builder. The first integer argument indicates the index before which the data is to be inserted. The data is converted to a string before the insert operation takes place.

`StringBuilder replace(int start, int end, String s)` `void setCharAt(int index, char c)`	Replaces the specified character(s) in this string builder.
`StringBuilder reverse()`	Reverses the sequence of characters in this string builder.
`String toString()`	Returns a string that contains the character sequence in the builder.

Note: You can use any `String` method on a `StringBuilder` object by first converting the string builder to a string with the `toString()` method of the `StringBuilder` class. Then convert the string back into a string builder using the `StringBuilder(String str)` constructor.

@komina_lodolo

The StringBuilder Class

StringBuilder Operations cont.

The StringDemo program that was listed in the section titled "Strings" is an example of a program that would be more efficient if a StringBuilder were used instead of a String. StringDemo reversed a palindrome.

Here, once again, is its listing:

Example:

```
public class StringDemo {
    public static void main(String[] args) {
        String palindrome = "Dot saw I was Tod";
        int len = palindrome.length();
        char[] tempCharArray = new char[len];
        char[] charArray = new char[len];

        // put original string in an
        // array of chars
        for (int i = 0; i < len; i++) {
            tempCharArray[i] =
                palindrome.charAt(i);
        }

        // reverse array of chars
        for (int j = 0; j < len; j++) {
            charArray[j] =
                tempCharArray[len - 1 - j];
        }

        String reversePalindrome =
            new String(charArray);
        System.out.println(reversePalindrome);
    }
}
```

Output: `doT saw I was toD`

To accomplish the string reversal, the program converts the string to an array of characters (first for loop), reverses the array into a second array (second for loop), and then converts back to a string.

If you convert the palindrome string to a string builder, you can use the reverse() method in the StringBuilder class. It makes the code simpler and easier to read:

```
public class StringBuilderDemo {
    public static void main(String[] args) {
        String palindrome = "Dot saw I was Tod";

        StringBuilder sb = new StringBuilder(palindrome);

        sb.reverse();  // reverse it

        System.out.println(sb);
    }
}
```

Running this program produces the same output: `doT saw I was toD`

Note that println() prints a string builder, as in: `System.out.println(sb);`

because `sb.toString()` is called implicitly, as it is with any other object in a println() invocation.

Note: There is also a `StringBuffer` class that is *exactly* the same as the `StringBuilder` class, except that it is thread-safe by virtue of having its methods synchronized. Threads will be discussed in the lesson on concurrency.

@Komina Jodolo

Summary

Summary of Characters and Strings

Most of the time, if you are using a single character value, you will use the primitive char type. There are times, however, when you need to use a char as an object—for example, as a method argument where an object is expected. The Java programming language provides a wrapper class that "wraps" the char in a Characterobject for this purpose. An object of type Character contains a single field whose type is char. This Character class also offers a number of useful class (that is, static) methods for manipulating characters.

Strings are a sequence of characters and are widely used in Java programming. In the Java programming language, strings are objects. The String class has over 60 methods and 13 constructors.

Most commonly, you create a string with a statement like `String s = "Hello world!";` rather than using one of the String constructors.

The String class has many methods to find and retrieve substrings; these can then be easily reassembled into new strings using the **+** concatenation operator.

The String class also includes a number of utility methods, among them split(), toLowerCase(), toUpperCase(), and valueOf(). The latter method is indispensable in converting user input strings to numbers. The Number subclasses also have methods for converting strings to numbers and vice versa.

In addition to the String class, there is also a StringBuilder class. Working with StringBuilder objects can sometimes be more efficient than working with strings. The StringBuilder class offers a few methods that can be useful for strings, among them reverse(). In general, however, the String class has a wider variety of methods.

A string can be converted to a string builder using a StringBuilder constructor. A string builder can be converted to a string with the toString() method.

@Komina Jodolo

Summary

Summary of Characters and Strings

Most of the time, if you are using a single character value, you will use the primitive char type. There are times, however, when you need to use a char as an object—for example, as a method argument where an object is expected. The Java programming language provides a wrapper class that "wraps" the char in a Characterobject for this purpose. An object of type Character contains a single field whose type is char. This Character class also offers a number of useful class (that is, static) methods for manipulating characters.

Strings are a sequence of characters and are widely used in Java programming. In the Java programming language, strings are objects. The String class has over 60 methods and 13 constructors.

Most commonly, you create a string with a statement like `String s = "Hello world!";` rather than using one of the String constructors.

The String class has many methods to find and retrieve substrings; these can then be easily reassembled into new strings using the **+** concatenation operator.

The String class also includes a number of utility methods, among them split(), toLowerCase(), toUpperCase(), and valueOf(). The latter method is indispensable in converting user input strings to numbers. The Number subclasses also have methods for converting strings to numbers and vice versa.

In addition to the String class, there is also a StringBuilder class. Working with StringBuilder objects can sometimes be more efficient than working with strings. The StringBuilder class offers a few methods that can be useful for strings, among them reverse(). In general, however, the String class has a wider variety of methods.

A string can be converted to a string builder using a StringBuilder constructor. A string builder can be converted to a string with the toString() method.

@Komuna Todolo

Autoboxing and Unboxing

Autoboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an int to an Integer, a double to a Double, and so on. If the conversion goes the other way, this is called unboxing.

Here is the simplest example of autoboxing:

```
Character ch = 'a';
```

The rest of the examples in this section use generics.

Consider the following code:

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 50; i += 2)
    li.add(i);
```

Although you add the int values as primitive types, rather than Integer objects, to li, the code compiles. Because li is a list of Integer objects, not a list of int values, you may wonder why the Java compiler does not issue a compile-time error. The compiler does not generate an error because it creates an Integer object from i and adds the object to li. The compiler converts the previous code to the following at runtime:

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 50; i += 2)
    li.add(Integer.valueOf(i));
for (int i = 1; i < 50; i += 2)
    li.add(i);
```

Converting a primitive value (an int, for example) into an object of the corresponding wrapper class (Integer) is called autoboxing. The Java compiler applies autoboxing when a primitive value is:

- Passed as a parameter to a method that expects an object of the corresponding wrapper class.
- Assigned to a variable of the corresponding wrapper class..

Consider the following method:

```
public static int sumEven(List<Integer> li) {
    int sum = 0;
    for (Integer i: li)
        if (i % 2 == 0)
            sum += i;
        return sum;
}
```

Because the remainder (%) and unary plus (+=) operators do not apply to Integer objects, you may wonder why the Java compiler compiles the method without issuing any errors. The compiler does not generate an error because it invokes the intValue method to convert an Integer to an int at runtime:

```
public static int sumEven(List<Integer> li) {
    int sum = 0;
    for (Integer i : li)
        if (i.intValue() % 2 == 0)
            sum += i.intValue();
        return sum;
}
```

@Romina Lodolo

Autoboxing and Unboxing

Converting an object of a wrapper type (Integer) to its corresponding primitive (int) value is called unboxing. The Java compiler applies unboxing when an object of a wrapper class is:

- Passed as a parameter to a method that expects a value of the corresponding primitive type.
- Assigned to a variable of the corresponding primitive type.

The Unboxing example shows how this works:

```
import java.util.ArrayList;
import java.util.List;

public class Unboxing {

    public static void main(String[] args) {
        Integer i = new Integer(-8);

        // 1. Unboxing through method invocation
        int absVal = absoluteValue(i);
        System.out.println("absolute value of " + i + " = " + absVal);

        List<Double> ld = new ArrayList<>();
        ld.add(3.1416);    // Π is autoboxed through method invocation.

        // 2. Unboxing through assignment
        double pi = ld.get(0);
        System.out.println("pi = " + pi);
    }

    public static int absoluteValue(int i) {
        return (i < 0) ? -i : i;
    }
}
```

The program prints the following:

```
absolute value of -8 = 8
pi = 3.1416
```

Autoboxing and unboxing lets developers write cleaner code, making it easier to read. The following table lists the primitive types and their corresponding wrapper classes, which are used by the Java compiler for autoboxing and unboxing:

Primitive type	Wrapper class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

@Romina Lodolo