Software Development (cs2500)

Lectures 2 and 3: Java: First Steps

M.R.C. van Dongen

September 29, 2010

Contents

1 Overview

These notes correspond to the first part of Chapter 1 from "the" book but there's no need to read the book to understand the notes. In addition these notes fill in some gaps which are not explained in the book. Some of the presentation about Chapter 1 is different from the book. This is a deliberate choice, which should allow you to study the theory from a different point of view. By the end of these notes you should be able to

- explain the purpose of a Java class,
- understand the difference between Java source and bytecode files,
- turn a Java class into Java bytecode with the `javac` compiler,
- execute the Java bytecode with the `java` application launcher,
- appreciate why types are useful,
- know Java's primitive types,
- write basic variable declarations and method definitions,
- use primitive type and string literals, and
- add comments to your programs.

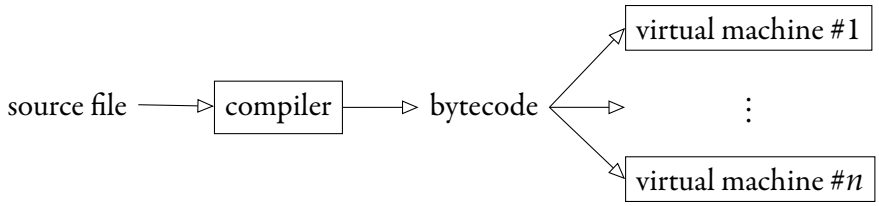

Figure 1: How Java works.

2 How Java Works

Figure ?? is a graphical representation of how Java works. The following explains Figure ?? in further detail.

Source: You start by writing one or more Java source files. You may create a Java source code file using any text editor of your choice. However, each source file should be plain text. If you're using `drjava` then `drjava` will make sure that your files are saved as plain text.

Compiler: You compile each source file using a Java compiler. Each source file consists of one or more *Java classes*. Compiling any given source file turns the classes in the source file into a *class file*. The extension of class files is '`.class`'. Each class file contains a *Java bytecode* representation of the corresponding class. If ⟨`source`⟩`.java` is your Java source file then you compile it by executing the command '`javac` ⟨`source`⟩`.java`'. Alternatively: use `drjava`.

When compiling your source file, the `javac` and `drjava` compilers main complain about certain errors. If they do complain then you must fix these errors and recompile until all errors have gone. For example, `javac` and `drjava` may complain about syntax errors and certain kinds of logic errors. In addition they may output warning messages. Usually warning messages are related to logic errors which may result in run-time errors. It is your task to resolve *all* errors. Lazy programmers prefer to ignore warning messages — they find it *easier* that way. Usually, this ignorance requires a dear price in the form of run-time errors. Good programmers resolve errors and warnings. (**As part of this course you are also expected to resolve all your errors. If you don't then you will lose marks.**) Finally, note that the ability to compile your source programs does not provide any guarantee that your program is correct:

> 'Compiling can only show the presence of errors, not their absence.'
>
> Adapted from Edsger W. Dijkstra

Bytecode: The output of the `javac` `drjava` compilers is low-level *Java bytecode*, which can be executed on any device capable of running `java`. The Java bytecode may be viewed as low level instructions which can easily be translated to machine instructions for most modern computer processors.

Virtual machines: You can now execute the Java bytecode on any device that has a *Java virtual machine* (JVM). Here a *Java virtual machine* is a program that can interpret/run Java bytecode. To start the JVM on your computer, you run `java` ⟨`Main Class`⟩, where ⟨`Main Class`⟩ is the class that contains "the" method `main`. Alternatively, you may use `drjava` to launch your application.

JVMs come in three main flavours:

Interpreters: These JVMs emulate the JVM instruction set by interpreting the JVM instructions.

JIT compilers: This class is the class of just-in-time compilers. A JIT compiler compiles the JIT instructions to native code *at runtime* prior to executing the resulting native code.

Ahead-of-time compilers: This class precompiles the entire Java application into native code *before* executing the resulting native code.

The main advantage of using Java bytecode in combination with JVMs is that this makes Java programs very portable. For example, you develop your Java application once and can run it on any machine that has a JVM. This portability aspect is best characterised by the following:

> Write once, run anywhere.

3 Code Structure

Figure ?? depicts the code structure of a Java source file (`.java` extension). The source file contains a class, and the class consists of two methods. Each method consists of statements.

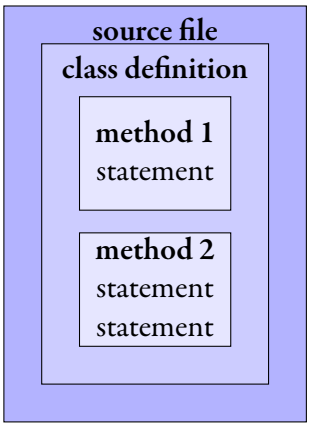

Figure 2: Code structure in Java.

Having studied how Java works at an abstract level, we shall now study an example that shows how Java works at the source code level. In our example we shall create a class called Dog that defines a

method called `bark` that lets `Dog` objects "bark." Further on, we shall implement another class that uses the `Dog` class.

The following is an example of a source file called `Dog.java`. The source file defines a single class, which is called `Dog`. Note that the name of the class starts with an uppercase letter. This is a convention: all class names start with an uppercase letter.

```Java
public class Dog {
␣␣␣␣public void bark( ) {
␣␣␣␣␣␣␣␣System.out.println( "Bark" );
␣␣␣␣␣␣␣␣System.out.println( "Arf" );
␣␣␣␣}
}
```

- The text '`public class Dog`' on the first line starts the definition of the class. The text '`Dog`' determines the name of the class. The opening brace following '`Dog`', the closing brace on the last line, and the text inside the braces defines what's in the class.

  Each `Java` source file should define at least one class — usually there is one class per source file. Since our example has only one class — it's called `Dog` — our `Dog` *class* goes in the *source file* `Dog.java`.

  The main class in the source file should have the same name as the *base name* of the source file. Here the *base name* of a file is the name of the file without the extension. For example, the base name of `Dog.java` is `Dog`.

- A class consists of *methods* and *attributes*. Here the *attributes* of a class/object are its variables. The discussion of attributes is postponed until some other lecture. In our example class there is one method called `bark`. The `bark` method goes in the class `Dog`.

  The text '`public void bark( )`' in our `Dog` class defines the *visibility*, the *name*, the *return type*, and the *formal parameter list* of an *instance method*. Methods may always be used in the class that defines them. A method's *visibility* determines whether the method may also be used outside the defining class. If a method's *visibility* is `public`, which it is in our case, then the method may also be used outside the defining class. The *name* of the method is `bark`. The *return type* of the method is `void`, which indicates that the method does not return any value. The *formal parameter list* of the method is empty, which indicates the method does not take any *parameter* — parameters are also referred to as *arguments*. Finally, the method `bark` is a *`Dog` instance method*, which means you can only use the method in combination with a `Dog` object *reference*. We'll get to that in a few moments. Each `Dog` object is an *instance* of the `Dog` class, hence the phrase "instance method."

  The symbols '␣' at the start of the lines are called *visible space* symbols. They are commonly used in textbooks to make space characters explicit but they do not occur in real source files. Each visible space in the example corresponds to a real space character in the `Java` source file.

  The reason for adding the extra spaces at the start of the lines in the body of the class, is to emphasise that the `bark` method is part of the `Dog` class. By adding the extra space characters, you can

recognise the structure of the source file by looking at the layout of the source file — as opposed to parsing the text from left to right, which is *much, much* more difficult. Usually, programmers add 2–4 spaces for each extra nesting level. For this course you are expected to do the same. **If you don't, you will lose marks.**

- The `bark` method contains statements. In the example there are two statements, which output the texts `Bark` and `Arf`, each on a line of its own. The spell '`System.out.println(` ⟨`string`⟩ `)`' outputs the text ⟨`string`⟩ to standard output.

It is common practice to start each class name with an uppercase letter and each method name with a lower case letter. This convention makes it easier to distinguish class identifiers from other identifiers.

4 More about Classes

Let's assume we also have the following class definition, which is stored in a source file called `Noise.java`. This time we're omitting the visible spaces.

```Java
public class Noise {
    public static void main( String[] args ) {
        // Create a new Dog object.
        Dog barney = new Dog( );
        // Make dog object bark.
        barney.bark( );
    }
}
```

The class `Noise` defines a method called `main`. The text inside the braces of `main` is called the *body* of `main`. Before looking at the body of the method `main`, it is important to notice that the definition of the method uses the keyword '`static`'. This makes the method `main` a *class method*. It is recalled that instance methods may only be used in combination with object references and the dot notation. Class methods are different. They are used without object references.

The class method `main` is very important because every `Java` program starts by executing this method. The statements in the body of `main` (1) create a new `Dog` object, (2) assign the resulting `Dog` object reference to the `Dog` object reference variable `barney`, and (3) call the `Dog` object's instance method `bark( )`. The following explains this in more detail.

1. **Create a new `Dog` object**: This is done by calling the `Dog` constructor method: using `new Dog( )`. The actual `Dog` object is stored in a part of memory which is known as the *heap*. The expression '`new Dog( )`' creates the dog object on the heap and returns a *reference* to the object. The assignment results in copying the `Dog` object reference (the right hand side of the assignment operator) into the memory cells which store the value of the variable `dog` (the left hand side of the assignment operator). You may think of the *`Dog` object reference* as a 'remote control' that may be used to tell the newly created `Dog` object what to do.

2. **Assign the resulting `Dog` object reference to the `Dog` object reference variable `barney`**: By doing this we can use `barney`'s `Dog` object reference — the remote control — to control the newly create `Dog` object.

3. **Call the object's `bark` method**: This is done using the construct '`barney.bark( )`'. To understand how this works, we have to understand the difference between *class* and *instance* methods.

   Class methods: Methods that have '`static`' as part of their return type are called *class* methods. For example, the method '`main`' in the `Noise` class is a class method. Calls to class methods always look like

   '⟨`class name`⟩`.`⟨`method`⟩`(` ⟨`actual parameter list`⟩ `)`'.

   Instance methods: Instance methods of a given class can only be called in combination with objects that are defined in the same class. Instance method *definitions* do not involve the `Java` keyword '`static`'. For example, the method `bark` in the `Dog` class is an instance method. Calls to instance methods always look like

   '⟨`object reference`⟩`.`⟨`method`⟩`(` ⟨`actual parameter list`⟩ `)`'.

   Intermezzo: The calls of the form '`System.out.println(` ⟨`string`⟩ `)`' in the `Dog` class are instance method calls. The instance method is called `println`. The object reference variable is `System.out`, which is `Java` speak for a class attribute (variable) called `out` of a class called `System`. For the moment you may forget about attributes.

   Having studied the difference between class and instance methods, we can now see that `bark` is defined as an *instance* method in the `Dog` class. For example, the method is defined without referring to the `Java` keyword '`static`'. Furthermore, the method is called in combination with a `Dog` reference (`barney`).

   Since `barney` is a `Dog` object reference variable it is allowed to use this instance method. Using our remote control analogy, each instance method call is a channel on a TV and `barney` owns (`barney`'s value is) a TV remote control that allows you to view these channels. You may carry out a given instance method call by selecting the corresponding instance method channel on the remote control. This works as follows. For each possible `Dog` instance method call (for each possible channel) there is a button on `barney`'s remote control that lets us call the method remotely (select the channel by remote control). Since '`bark( )`' is a possible instance method call, `barney`'s remote control has a button for selecting this instance method call.

   - The text on this button (the number of the channel) is '`bark( )`'.
   - The button on the remote control has a similar shape as a dot ('`.`').
   - The button and the text on the button are called '`.bark( )`'.
   - Pressing the button '`.bark( )`' on `barney`'s remote control results in executing '`barney.bark( )`'.

Summarising, 'barney.bark( )' lets barney's Dog object carry out the instance method call 'bark( )' remotely.

Remember that the program javac turns a source file into Java bytecode. Also remember that the Java virtual machine (JVM) runs the bytecode. Let's assume we use the two as follows.

Unix Session

```
$ ls
Dog.java    Noise.java
$ javac Noise.java
$ ls
Dog.java    Dog.class    Noise.java     Noise.class
$ java Noise
Bark
Arf
$
```

By running the JVM java as 'java Noise' you tell java that Noise is the main class. The first thing the JVM (java) will do is load the file Noise.class and look for the definition of the class Noise. Since Noise is the main class, the JVM starts looking for the method main in that class and execute your application by executing the statements in the body of the method main.

All Java programs start by executing a class method public static void main(String[] args). The class which this main is in is called the main class. In our example, our main class is Noise.

5 Types and Declarations

5.1 Motivation

Java is a *strongly typed* language. In short this poses restrictions on the kinds of operands that are allowed as part of expressions and the kinds of arguments that are allowed as arguments of methods. You are only allowed to use expressions/operands/arguments that have types which make sense. One of the advantages of strongly typed languages is that they help the programmer avoid certain kinds of errors: you cannot compare apples and oranges. Should the programmer make such an error, then the compiler will detect the error and inform the programmer about it.

Note that PHP is not a strongly typed language. The following example should demonstrate this. Notice that the error on the second line only manifests itself at *runtime* when the third line is being executed. Needless to say, a program like this won't impress potential customers whose income depends on applications like this.

Don't Try this at Home

```
$s = "SELECT author FROM BOOKS";
$s = 1; // ???
$result = mysql_query( $s ); // D'oh!
```

As already mentioned, Java *is* strongly typed. The following snippet, which is the Java equivalent of the previous PHP example, should make the advantages of types clear. When you try to compile this

example, the javac compiler will detect the error in Line 2 at *compile time* and report the error. In addition it will refuse to produce a class file. By informing the programmer about the error, the programmer will now notice the error. This allows him to fix the error before any run-time damage can be done.

```Java
String s = "SELECT author from BOOKS";
s = 1; // D'oh!
Query q = new Query( s );
```

5.2 Variables

To help the compiler determine the type of the variables in your program, Java requires that you *declare* each variable. Each variable declaration requires the name and the type of the variable. In addition, Java requires that you provide the *return type* of your methods as well as the *names* and *types* of formal parameters.

Variable declarations come in two flavours:

- The first kind is the simplest form. All it does is state the type and name of the variable. The following example involves (1) the declaration of an integer called expression and (2) an assignment of an expression to the variable. Here the declaration does not initialise the variable.

  ```Java
  int expression;
  expression = 1 + 2;
  ```

- The second form combines declaration and initialisation. The following example, which is equivalent to the previous example, shows how to write this kind of declaration.

  ```Java
  int expression = 1 + 2;
  ```

5.3 Methods

As already pointed out, there are two kinds of methods: *class* methods and *instance* methods. You define a class method by adding the keyword 'static' before the return type in the method definition. Method definitions of instance methods don't have the keyword 'static' before the return type. The following provides the general syntax for method definitions.

```Java
⟨visibility modifier⟩ ⟨static option⟩
⟨return type⟩ ⟨name⟩ ( ⟨formal parameter list⟩ ) {
    ⟨body of method⟩
}
```

For the moment you may assume that you may only write 'public' for ⟨visibility modifier⟩. Using 'public' makes the method visible (read *usable* or *callable*) *anywhere*. Here anywhere includes the class that defines the method and classes which are outside the defining class. The text ⟨static option⟩

is either 'static' or nothing: ''. By using 'static' you define a class method and by omitting it you define an instance method. The text ⟨return type⟩ determines the return type of the method. It can be any existing Java type, the name of a class, or 'void'. Existing Java types may be 'int' (integer), 'char' (character), and so on. If ⟨return type⟩ is 'void' it means that the method does not return any result. The text ⟨formal parameter list⟩ is a comma-separated list of items of the form '⟨type⟩ ⟨name⟩'. Here ⟨type⟩ determines the type of ⟨name⟩ in ⟨body of method⟩. Each ⟨name⟩ in the list should be different. Finally, ⟨body of method⟩ is the body of the method. It consists of statements and declarations.

The following example shows two instance methods called 'add' and 'compute'. The method 'add' returns an int. The other method does not return any value. Both methods take two int parameters.

Java
```
public
int add( int fst, int snd ) {
    int result;
    result = fst + snd;
    return result;
}

public
void compute( int first, int second ) {
    String announcement = "And the result is: ";
    int sum = add( first, second );
    System.out.print( announcement );
    System.out.println( sum );
}
```

5.4 Primitive Types

The simplest types in Java are its *primitive* types. Primitive types don't involve objects. There are three kinds of primitive types: numbers, characters, and Booleans.

Numeric: The numeric primitive types are subdivided in integers (whole numbers) and floating point numbers (fractional numbers).

Integral: The integer types are byte, short, int, and long.

Floating point: The floating point types are float, and double.

Characters: Characters are used to represent text.

Booleans: Boolean values are used to represent truth values, yes/no, on/off, two-state valued things, and make decisions.

5.4.1 Numeric Types

Table **??** list Java's primitive numeric types.

Integral Types			
Name	**Storage**	**Minimum Value**	**Maximum Value**
`byte`	8 bit	-128	127
`short`	16 bits	-32,768	32,767
`int`	32 bits	-2,147,483,648	2,147,483,647
`long`	64 bits	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
Floating Point Types			
Name	**Storage**	**Minimum Positive Value**	**Maximum Positive Value**
`float`	32 bits	$\pm 1.4 \times 10^{-45}$	$\pm 3.43 \times 10^{38}$
`double`	64 bits	$\pm 4.9 \times 10^{-324}$	$\pm 1.80 \times 10^{308}$

Table 1: Primitive numeric types in Java.

5.4.2 Rounding

It is important to realise the consequences of *finite* representation: an n-bit primitive type cannot represent more than 2^n different values. For most day-to-day applications an `int` should suffice for "counting". If 32 bits isn't enough, then use a `long`. If 64 bits isn't enough, then you may have to use a `BigInteger`, which is an object type. Finally, floating point computations usually result in *rounding errors*. For example, the largest and smallest positive `double` values are `Double.MAX_VALUE` $= 2^{1024} - 2^{971}$ and `Double.MIN_VALUE` $= 2^{-1074}$. Still, subtracting `Double.MIN_VALUE` from `Double.MAX_VALUE` results in `Double.MAX_VALUE`.

A common error is that beginning Java programmers use the wrong methods for certain kinds of integer computations. For example, they use the class method `pow` from the `Math` class for exponentiation. Just by looking at the type signature of the method you can tell that this method should never be used for integer exponentiation: it is `static double pow( double a, double b )`. There is even a comment in the JavaDoc documentation (`http://download.oracle.com/javase/1.4.2/docs/api/java/lang/Math.html`) stating that

> If both arguments are integers, then the result is exactly equal to the mathematical result of raising the first argument to the power of the second argument if that result can in fact be represented exactly as a double value.

In short you cannot use this method to implement general integer exponentiation.

5.4.3 Representation

Java's integral types are represented as two's complement integers. Theoretically, the two's complement representation support *signed* and *unsigned* integers. However, Java only supports signed integers. Before continuing, lets remind ourselves what it means for a bit sequence to be a two's complement integer.

5.4.4 One's Complement

The *one's complement* of a bit sequence is computed by *flipping* each bit. Here flipping a bit, b, means turning it into its complement, $1 - b$. This turns a 1 into a 0 and a 0 into a 1.

5.4.5 Two's Complement

In Java an n-bit integers is represented using the *two's complement* format.

Non-negative: The value of a non-negative n-bit `int` is represented by a 0 followed by $n - 1$ bits. The value of the resulting bit sequence is as per usual, so the bit sequence 01010000 represents the number $2^6 + 2^4 = 64 + 16 = 80$

Negative: Negative values are represented as follows. First you compute the absolute value and represent it in binary using n-bits. Next you take the one's complement of the bit sequence. Finally, you add one (ignoring the left-most overflow bit).

The largest possible number is represented by a zero followed by $n - 1$ ones. This bit sequence represents the number $2^{n-1} - 1$. A bitsequence represents a negative number if (and only if) it starts with a one. The smallest possible number is represented by a one followed by $n - 1$ zeros. This bit sequence represents the number -2^{n-1}. We have 2^{n-1} negative, $2^{n-1} - 1$ positive, and 1 zero value. In total there are 2^n possible values.

The following example shows how to get the 32-bit two's complement representation of -1. First take the representation of absolute value of -1:

Representation of `abs( -1 )`			
B_3	B_2	B_1	B_0
00000000	00000000	00000000	00000001

Next take the one's complement:

One's Complement			
B_3	B_2	B_1	B_0
11111111	11111111	11111111	11111110

Finally, add 1:

Add 1			
B_3	B_2	B_1	B_0
11111111	11111111	11111111	11111111

5.4.6 Characters

The Java type for representing characters is called 'char'. Java characters are based on Unicode, which is a character standard which supports up to 65536 different characters. It is recalled that $65536 = 2^{16} = 2^8 \times 2^8$. An 8-bit byte can represent 2^8 values. This explains why chars are represented using two bytes.

5.4.7 Booleans

Java represents truth values using the type 'boolean'. The type boolean has only two values, which are written in the Java language as 'true' and 'false'. The Java language does not specify how how to represent boolean values and how many bits should be used per boolean value.

5.5 Primitive Type Literals

Clearly, knowing about Java's primitive types is not enough to work with them. For example, which values may you assign to their corresponding variables, what do you type to express these values, and what kind of expressions are you allowed to form with them? In this section we shall address the second question, namely, what are the primitive type *literals*? Here a *literal* is an explicit data value in a program.

5.5.1 Integer Literals

If there is no ambiguity then integer literals are represented as decimal numbers. The following shows an example.

```Java
short s = 100;
int   i = 0;
long  l = -100;
```

This is the default representation and Java assumes that each such literal is an int. However, the value must be in the right range, so byte s = 128 is not allowed.

Occasionally, you may need to write long literals that cannot be represented using a 32-bit two's complement value. Adding an 'l' or 'L' to the end of a decimal number makes the resulting sequence an explicit long literal. The following provides an example.

```Java
long l1 = 2147483647;  // Largest possible int.
long l2 = 2147483648;  // Too large: not allowed.
long l3 = 21474836481; // Also too large.
long l4 = 21474836481; // Allowed but not clear.
long l5 = 2147483648L; // Perfect!
```

The previous example shows that it is difficult to see the difference between the letter 'l' and the digit '1'. For this reason you should always prefer the 'L' to the letter 'l'.

Java also lets you write integral literals in octal (base 8) and in hexadecimal (base 16). Octal literals start with a zero (sigh): '022' corresponds to '18'. Hexadecimal literals start with the string '0x' (zero

then x): '0x12' corresponds to '18'. Starting hexadecimal literals with '0X' is also allowed. However, using the uppercase X is not as clear as using the lowercase x.

5.5.2 Double Literals

By default, Java assumes that floating point literals are doubles. The following are possible ways to write floating point literals.

- '⟨sign option⟩⟨digit sequence⟩.⟨digit sequence⟩'. These literals have the "expected" value, so '-10.5' corresponds to $-10\frac{1}{2}$.
- '⟨sign option⟩⟨digit sequence⟩.' or '⟨sign option⟩.⟨digit sequence⟩' These literals also have the "expected" value, so '-10.' corresponds to -10 and '1.' corresponds to 1.
- '⟨base⟩⟨exponent⟩', where ⟨base⟩ is given by

  '⟨sign option⟩⟨digit sequence⟩.⟨digit sequence⟩',

  and ⟨exponent⟩ is given by

  '⟨E or e⟩⟨sign option⟩⟨digit sequence⟩'.

  This form corresponds to scientific notation. Let b be the number before the 'E' or 'e' and let e be the number after the 'E' or 'e'. The value of the literal corresponds to $b \times 10^e$. So '1.5E2' corresponds to $1.5 \times 10^2 = 150$.
- Variations of scientific notation are also possible. This allows you to write literals like '-1.E0', '.1E0', and so on.

5.5.3 Float Literals

Adding an 'f' or 'F' at the end turns a floating point literal into a 'float'. If you need a float, then the letter 'f' or 'F' is required. The reason why this is required is that in *general* double literal, which is the default, cannot always be converted to a corresponding float value without loss of precision. Likewise, adding a 'd' or 'D' at the end states that the literal is a 'double'. The following shows some examples of floating point literals.

```Java
double d1 = 1.0E10;    // Grand.
double d2 = -1.0E-10D; // Grand.
double d3 = -.1;       // Grand.
float  f1 = 1.0;       // Not allowed.
float  f2 = 1.00F;     // Grand.
float  f3 = -1.0E-10F; // Grand.
```

5.5.4 Character Literals

Regardless of the kind, character literals are always written inside two single quote symbols (`'`). There are three main classes of character literals:

Normal characters: Here we have any Unicode character inside the quotes. The character that is represented by the literal is given by the character that is inside the quotes. The following are possible example: `'a'`, `'B'`, `'ñ'`, ...

Escape sequences: This class of literals start with a backslash after the first quote. Examples are `'\n'` (newline), `'\t'` (tab), `'\"'` (double quote), `'\''` (single quote), `'\\'` (backslash), ...

Unicode escapes: These literals are of the form `'\u`⟨`hexadecimal number`⟩`'`, where ⟨`hexadecimal number`⟩ represents the number of the Unicode character in hexadecimal (base 16). Examples are `'\u00F1'` (ñ), `'\u0108'` (Ĉ), and so on. For this course you may forget about this class of character literals.

5.5.5 Boolean Literals

We've already seen the `boolean` literals: they're given by '`false`' and '`true`'.

5.6 Object Types

The last class of types is the class of *object types*. Any type which is not primitive is an object type. If ⟨`class`⟩ is the name of the class that defines the object, then ⟨`class`⟩ is the type of the object.

Remember that `Java` does not allow you to deal directly with objects. Instead you deal with them indirectly through the use of object reference variables. The type of a ⟨`class`⟩ object reference variable is written ⟨`class`⟩. It is an commonly accepted `Java` programmer's convention to start class names with uppercase letter. Object reference variable declarations are written as per usual:

```Java
Dog cerberus = new Dog( );
Cat felix = new Cat( );
```

6 Strings

Strings are first-class citizens in `Java`. The type of strings is '`String`'. The uppercase '`S`' suggests that strings are objects. Indeed, this is true. String literals start and end in double quotes (`"`). Inside the quotes you have a (possibly empty) sequence of characters. Here the characters are what's "in" the single quotes of character literals. Notice that each string literal involves the creation of a `String` object. The following is an example with `Strings`.

```Java
String str1 = "Hello world!";
System.out.print( str1 );
String str2 = "String str1 = \"Hello world!\";";
System.out.println( str2 );
System.out.print( "System.out.print( str1 );" )
```

Strings are objects, so it is reasonable to expect that the `String` class defines some instance methods. Indeed, the following are two useful instance methods:

`int length( ):` Returns the number of characters in the string.

`char charAt( int pos ):` Returns the character at position `pos` in the string. As is usual, the first position is 0 and the last position is its length minus 1.

The following provides an example.

```Java
String str = "text";
char second = str.charAt( 2 );
int length = "hi".length( );
```

Here the second assignment assigns the character `'x'` to `second`. The last assignment assigns the value 2 to `length`.

The last statement is interesting because it demonstrates that object reference variables are not always needed for instance method calls. More generally, an object reference *value* is enough. Since a string literal is a `String` (object) reference value, you can write '"hi".length( )'. When we wrote '`barney.bark( )`', the expression '`barney`' also "counted" as an object reference value. However, the value which was used for the object reference value was the (current) value of the object reference variable `barney`. This is ***exactly*** the same mechanism as for `int` expressions:

> When you write a variable where a value is expected then you get the (current) value of that variable. *Anonymous Programmer*

The following should demonstrate this once more. Of course, the values in the first two statements are `int` values, and the values in the last three statements are *object reference* values.

```Java
int first  = 1;     // Use the literal value.
int second = first; // Use the (current) value of the variable first.

String quote = "To be or not to be";          // Use the literal value.
System.out.print( "Here's a famous quote: " ); // Use the literal value.
System.out.println( quote ); // Use the (current) value of the variable quote.
```

7 Comments

Java has two kinds of comments.

- The first class of comments are one liners. They start with '//' and last till the end of the line.
- The next class of comments is usually used for multi-line comments — but they can be used on a single line. These comments start with '/∗' and last until the next occurrence of the sequence '∗/'. Here it is assumed that '/∗' and '∗/' are outside the double quote symbols of string literals.

Figure ?? provides an example with several comments.

```
String spuds = "potatoes"; // Everybody loves them.
String meat  = "steak"; // Vegetarians look other away.
String drink = "beer"; // Drink this sensibly.
String more  = "wine"; // Drink this sensibly too.
/*
 * Invite guest. Wine and dine guest.
 * Make sure guest gets home safe.
 */
Host host = new Host( );
Guest guest = new Guest( );
host.invite( guest );
guest.eat( meat );
guest.drink( drink );
guest.eat( spuds );
guest.drink( more );
host.ring( "021 4272255" ); // Ring Yellow Cabs
```

Figure 3: Comments in Java.

8 For Monday

Study the notes, implement the program that makes dogs bark, compile the required source files, and "run" the resulting byte code.