

# CS1Ah Lecture Note 4

## Types and Java

Some languages used in computing are *typed* and some are not. Several benefits come from having types in a programming language. The foremost is that a typed language may prevent programs from being executed if they contain errors which would cause them to go wrong during execution. We call these kind of errors *type errors*. A simple example of a type error is the attempt to add an integer value to a boolean value, writing `3 + true`. The `javac` compiler will refuse to compile a program which contains this expression and will return a diagnostic error message similar to the one below.

```
operator + cannot be applied to int, boolean
```

Because this kind of error is caught by the compiler, the erroneous expression will never be executed in a running program. This is useful because we do not have any expectation of the “correct” result of this expression anyway.

As an example of an untyped language which is not typed, consider the language of UNIX operating system commands. The command `lpr` enters a file into the queue for a printer. However, some UNIX files cannot be converted to a printable representation, for example, any `.class` file created by the Java compiler. If the `lpr` command considered types then when we issued the UNIX print command `lpr Prog.class` we would expect to be informed that the command was nonsensical and that the file `Prog.class` would not be sent to the printer. Instead the file is entered into the queue for the printer, which we can see with the `lpq` command:

```
Queue: 1 printable job
Filter_status: printer connection established for stg
Rank   Owner/ID           Class Job  Files           Size Time
active stg                 X  770  Prog.class      228  14:24:10
```

When the file reaches the head of the queue it is found to be unprintable and we will receive an electronic mail message which states “Couldn’t convert binary file to PS”<sup>1</sup>. Here a program (the `lpr` command) was allowed to run even though it was certain to fail. This kind of problem arises with *untyped* programming languages such as the language of Unix commands.

---

<sup>1</sup>In the deplorable tradition of terse error messages from computer programs this message abbreviates the name of the PostScript page description language used by laser printers to “PS”. This is confusing until one knows that PS stand for PostScript.

## 4.1 Reference types and primitive types

Java is a typed programming language. Java's types are split into the **reference types** and the **primitive types**. Roughly, reference types are the types of objects, in particular, the names of classes (e.g. `String`, `Student`). By convention these names begin with an upper case letter. Reference types are so-called because their values (objects) are manipulated *by reference*. This gives rise to the notion of object identity explained in CS1Ah Lecture Note 1, *Introduction to Objects*: we may have two *different* objects that have the same contents, as well as two names (aliases) for the same object.

Speaking roughly again, the *primitive types* are the types of numeric values. By convention, these names begin with a lower case letter, and are fixed as part of the language; it isn't possible for the programmer to define new primitive types. In contrast with reference types, values of primitive type are treated directly: the number 27 treated as an integer is equal to every other integer which has the value 27, whatever its name. The location in the computers memory that stores the number is hidden from us.

### Primitive types

Java's primitive types are given in the following table.

Type	Size	Minimum value	Maximum value
<code>boolean</code>	1 bit	This type is not ordered. Its two values are <code>true</code> and <code>false</code>	
<code>char</code>	16 bits	<code>\u0000</code>	<code>\uFFFF</code>
<code>byte</code>	8 bits	<code>Byte.MIN_VALUE</code> ( $-2^7$ )	<code>Byte.MAX_VALUE</code> ( $2^7 - 1$ )
<code>short</code>	16 bits	<code>Short.MIN_VALUE</code> ( $-2^{15}$ )	<code>Short.MAX_VALUE</code> ( $2^{15} - 1$ )
<code>int</code>	32 bits	<code>Integer.MIN_VALUE</code> ( $-2^{31}$ )	<code>Integer.MAX_VALUE</code> ( $2^{31} - 1$ )
<code>long</code>	64 bits	<code>Long.MIN_VALUE</code> ( $-2^{63}$ )	<code>Long.MAX_VALUE</code> ( $2^{63} - 1$ )
<code>float</code>	32 bits	<code>Float.MIN_VALUE</code> ( $1.4 \times 10^{-45}$ )	<code>Float.MAX_VALUE</code> ( $3.4 \times 10^{38}$ )
<code>double</code>	64 bits	<code>Double.MIN_VALUE</code> ( $4.9 \times 10^{-324}$ )	<code>Double.MAX_VALUE</code> ( $1.8 \times 10^{308}$ )

Booleans are the simplest type: a boolean variable can be one of two values. Instead of using numbers to represent these values, Java uses the words `true` and `false`.

Character sets in programming languages map symbols onto particular numeric values. Java's character set (Unicode) is distinctive in providing access to many characters from many alphabets. There are 65536 (two to the power sixteen) character positions in the Unicode set. Many programming languages provide access to only 128 (two to the power seven) characters. It is possible to convert between integer and character values by **casting**. For example, `(int)'£'` returns 163 and `(char)163` gives the '£' character. Characters can also be accessed by giving the hexadecimal value of their Unicode number. For example, the literal `'\u00A3'` also gives the '£' character.

The types `byte`, `short`, `int` and `long` represent integral values of increasing magnitudes. Floating-point numbers are the computer's approximation to real numbers. The

two types `float` and `double` can store values of differing magnitudes, but also differing *precision*. Casting also converts floating-point numbers to integers. For example, `(int)3.8` returns 3.

The important thing to remember about floating-point numbers is that unlike real numbers, they have a limited precision, so some values can only be represented approximately. This means that in programming it is safer to compare a floating point value against a range of values rather than against a single value. Floating-point arithmetic has other peculiarities in its behaviour, different from integer arithmetic. For example, floating-point division by zero returns NaN (“Not a Number”, a representation of infinity) whereas integer division by zero raises a Java *exception* (an error flag).

## The purpose of primitive types

One use of Java’s primitive types is to detect errors when programs are compiled instead of when they are executed. For example, it is never meaningful to subtract two boolean values so any Java program which contains such an expression is rejected when it is submitted for compilation and cannot be executed at all.

Another important use of types is in improving the efficiency of programs. Integer arithmetic can be performed more efficiently than floating-point arithmetic so by using types to separate out floating-point values from integer values we can reduce the execution time of programs. Modern computers are impressively fast but the advantage which we gain by using integer arithmetic over floating-point can be noticeable in cases where many numerical calculations need to be performed at high speed.

## 4.2 Wrapper classes

Primitive types are needed to represent simple data values and operations on them. In an object-oriented language such as Java it is often also necessary to have an object which stores a simple data value, often because some methods only accept objects as their parameters. A class whose purpose is to convert values to objects in this way is known as a *wrapper class* because it does little more than wrap up a value.

Java provides operations to convert between values of primitive types and objects:

Converting a literal to an object	Converting an object to a value
<code>Byte B = new Byte((byte)27);</code>	<code>byte b = B.byteValue();</code>
<code>Short S = new Short((short)500);</code>	<code>short s = S.shortValue();</code>
<code>Integer I = new Integer(1234);</code>	<code>int i = I.intValue();</code>
<code>Long L = new Long(10000000L);</code>	<code>long l = L.longValue();</code>

Since objects which store simple values seem useful why does the Java language not simply define all simple values to be objects in the first place? That approach is appealing, because it treats data uniformly. Both computer science and software engineering value elegance and consistency in design. The disadvantage of treating all primitive values as objects, however, is that it incurs a cost. For each data value stored as an object we must have an additional *pointer* to the address in computer memory where it resides (a pointer is either 32 or 64 bits). For a byte stored as an object we then have 8 bits of data and 32 or 64 bits of overhead. Although computer memory is often

available in generous quantities it is distasteful to waste it in this fashion. Computer science and software engineering do value simplicity and elegant design, but they are also practical subjects: the decision to have primitive types in Java and convert these to objects when needed strikes a balance between simplicity and efficiency.

Another question which the alert reader may be wondering at this point, is what happens with equality when we treat primitive types as objects? For example, we now have the case that two different *Byte* objects B1 and B2 which both wrap up the value 27 may appear different. (In fact, every execution of `new` in Java creates a fresh object different from any other, using some new piece of memory). For numbers, this is not desirable: unlike physical objects in the real world, numbers are abstract entities; one 27 value should be the same as any other 27 value.

To avoid this problem, we might compare *Byte* objects by always comparing their containing primitive values with the `byteValue()` method. A better solution is to use the *equality method* which compares two objects, written `B1.equals(B2)`. The `equals` method can be defined in such a way that it determines a notion of equality appropriate for the class. For all of the wrapper classes, `equals` is defined exactly to compare the containing values.

### 4.3 Extending the primitive types

Although we cannot define new primitive types, we can add new classes to the language for representing numeric values as objects. An example of such a class is the `BigInteger` class, which provides *arbitrary precision* integer arithmetic. There are no preset limits on the size of big integers so there are no equivalents of the constants `Integer.MIN_VALUE` and `Integer.MAX_VALUE`. Here is an example program which prints powers of two continuing far beyond two to the power 64:

```
1  import java.math.BigInteger;
2  class Powers {
3      public static void main(String[] args) {
4          final BigInteger two = new BigInteger("2");
5          BigInteger I = new BigInteger("1");
6          while (true) {
7              I = I.multiply(two);
8              System.out.println(I);
9          }
10     }
11 }
```

Because big integers are objects operated upon by methods, instead of values operated upon by operators, the notation used is different. For example, on line 7 we must write `I.multiply(two)` instead of `I * 2` as we would with `int` values. Equivalently we could have written `two.multiply(I)`. The value stored in the `I` object is altered by the assignment, not the method invocation.

*Note by Stephen Gilmore. Revised by David Aspinall.  
David Aspinall, 2002/10/21 15:59:33.*