CSC142 Supplemental Reading: Week01

Contents

1 Primitive Data Types 2
1.1 Numbers 2
1.1.1 Numeric Literals 2
1.1.2 Exponential Notation 3
1.2 Characters 3
1.3 Escape Sequences 3
1.4 Type *boolean* 4
2 Numeric Processing 4
2.1 Numeric Operators 4
2.2 Operators and Operator Precedence 7
2.3 Relationships between Numeric Data Types: Widening 7
2.4 Explicit Type Casts: Narrowing 8
2.5 Operator Overloading: + ... 9
3 Operator Precedence Table 10

1 Primitive Data Types

Java is a strongly-typed programming language. Every variable, literal value (constant), and expression has a data type, and a variable may store only information that conforms to its data type. Most data types are reference types, like type String; we will learn more about reference types later in the quarter. Besides these, Java provides eight **primitive types** to store information of a simpler nature: integer and floating-point numbers, characters, and Boolean (true/false) values. Here is a summary of the eight primitive types:

Data Type	Description	Bits
byte	an integer value in the range -128 to +127, or -2^7 to $+(2^7-1)$	8
short	an integer value in the range -32,768 to +32,767, or -2^{15} to $+(2^{15}-1)$	16
int	an integer value in the range ~-2 billion to ~+2 billion, or -2^{31} to $+(2^{31}-1)$	32
long	an integer value in the range ~$-9x10^{18}$ to ~$+9x10^{18}$, or -2^{63} to $+(2^{63}-1)$	64
float	a floating-point number in the range ~±1.4E-45 to ~±3.4E38, with 6 to 7 significant digits of accuracy	32
double	a floating-point number in the range ~±4.9E-324 to ~±1.8E308, with 14 to 15 significant digits of accuracy	64
char	a character in the Unicode character set	16
boolean	one of two possible logical values: ***true*** or ***false***	8

The Reges/Stepp textbook covers a fundamental group of four of these primitive types (*int*, *double*, *char* and *boolean*) in Section 2.1; the others are covered in Appendix C.

Every one of the primitive types is a Java keyword, as are the values ***true*** and ***false***. Let's examine each of these data types in more detail.

1.1 Numbers

Numbers in computer programs are generally grouped into two categories: integers and floating-point numbers. Java provides four different **integer** data types, which differ by the amount of memory required to store a number and, as a consequence, the range of different values that can be stored. The most commonly used integer type is ***int***.

A **floating-point** number is a number that may have a decimal part. A floating-point number may take on a much wider range of values, but with a limited number of significant digits. Java provides two different floating-point types, and ***double*** is the most commonly used.

1.1.1 Numeric Literals

Every literal number used in a Java expression has a data type. If the number has no decimal point, it is type ***int***. If the number has a decimal point, it is type ***double***. It is also possible to specify literal values of type ***float*** or type ***long*** by appending a letter (upper- or lower-case) at the end of the number.

Here are some literal values and their data types:

Data Type	Sample Literal Values
`int`	0, -4, 125, 12345678
`long`	0L, -4L, 1234567891234567L
`double`	0.0, -12.5, 7.65, .5, 5E6, 1.23456e-30, 5E-3
`float`	0.0F, -12.5f, .5f, 5E6F, 1.234e-30F, 5e-3f

Note that commas and spaces are not allowed in numeric literals.

1.1.2 Exponential Notation

Some of the entries in the tables above make use of a notation referred to as **exponential notation** (you may know it as scientific notation). The number 5E6 stands for "5 times 10 to the 6th power," or $5x10^6$ (5 million). This notation can be used to enter numerical literals (always interpreted as floating-point numbers) in a program, and is also sometimes used when very large or small numbers are printed. The 'E' may be upper- or lower-case. Here are two more examples:

```
double distanceToSun = 93e6; // Miles
double electronCharge = 1.602E-19; // Coulombs
```

1.2 Characters

Type ***char*** stores a single character from the Unicode character set. Like all other data in a computer program, characters are represented by numbers. The number that represents a given character is referred to as a character code, and these codes range from 0 to 65,535. The number of different characters that can be displayed may be system-dependent and may also depend on the context in which the character is used.

Literal characters may be used in a program by placing the character inside single-quote marks. (Literal string values are always surrounded by double-quote marks.) A literal ***char*** may be used anywhere a ***char*** is needed. Here's an example that illustrates the difference between a char literal and a string literal:

```
char initial = 'V';
String s = "V";
```

So, 'V' is not the same thing as "V" at all -- the latter is an object! Unlike in HTML and JavaScript, single-quotes and double-quotes are not interchangeable in Java.

1.3 Escape Sequences

Some characters are not available on the keyboard, or have other meaning when typed in code. These characters may be entered into a program by using an escape sequence -- a backslash character followed by a pre-defined code. Notice that the two symbols together represent a single character. Here are a few examples:

```
char newline = '\n';
String s = "She said \"Yes!\"";
```

Here is a short list of the most important escape sequences:

Escape Sequence	Character or Description
\n	**newline character**
\\	****
\'	'
\"	"

1.4 Type *boolean*

Variables of type ***boolean*** may be used to store a logical value -- either ***true*** or ***false***. We'll say more about this when we cover Boolean operators and the ***if*** statement. Instance variables of type *boolean* are initialized to *false* by default.

2 Numeric Processing

To perform numeric calculations, we will use expressions.

An expression is any combination of variables, literals (values), operators and/or method calls that, when evaluated, can be reduced to a single value.

We've seen variables and literals already; now let's take a look at operators.

2.1 Numeric Operators

Here is a list of the numerical operators in Java that we'll be using. Each of these may be applied to numbers to construct an expression.

Java Operator	Description
++ -- -	Increment by one Decrement by one Unary minus (change the sign)
* / %	Multiplication Division Modulus (the remainder of division)
+ -	Addition Subtraction
= *=, /=, %=, +=, -=	Assignment Combined assignment

(continues…)

Let's go through what each operator does:

- The **increment (++)** and **decrement (--)** operators change the value of a variable by one. For example:

```
x = 3;
x++;
```

After this code executes, the value stored in x is 4. The statement **x++; is just short-hand for x = x + 1;** The effect is the same. Decrement works the same way, only it subtracts 1 from the variable. So, the statement **x--; has the same effect as x = x - 1;**

Now, there is a school of thought that says that this is as far as a good programmer should ever go with the ++ and -- operators. The uses in the next paragraph are considered bad programming style in some circles because they make code unnecessarily difficult to read. However, it's common to see these other uses in programming placement tests used by some 4-year schools, so we include them for completeness. read the next paragraph and the accompanying table to see other ways in which the ++ and -- operators can be used, but then consider ***not*** doing so in your own programs.

Both of these operators may be used either in the prefix location (e.g. ++x) or in the postfix location (e.g. x++), and the behavior is slightly different when used in a larger expression. ++x means "increment x by 1, then use the new value in the larger expression." x++ means "use the current value of x in the larger expression, then increment x by 1." Here's a trace of a short sequence of statements to make this clearer:

Statement	value of x	value of s
`int x = 3;`	3	--
`int s = 2;`		2
`x++;`	4	
`x = 7;`	7	
`s = x++; // assign first, then increment`	8	7
`x = 7;`	7	
`s = ++x; // increment first, then assign`	8	8
`s = 2 + ++x;`	9	11
`x = 7;`	7	
`s = x--;`	6	7
`x = 7;`	7	
`s = --x;`	6	6

- The **unary minus operator (-)** will change the sign of the expression (but not the variable it is next to). For example:

```
int x = 3;
int y = - x;
```

This stores a -3 in **y**. The variable **x** isn't changed.

- The **multiplication (*)** operator works as you expect.

- The **division (/)** operator's behavior depends on the data type of its operands. If the operands are type float or double, the operation works as you expect, giving you a floating-point result. However, if the operands are integers, then integer division is performed. This means that the result is truncated. This happens because the data type of the result of a numeric operation is always the same as the data type of the operands (the numbers used in the operation). Here's an example to illustrate the point:

  ```
  int x = 3 / 4; // the value 0 is stored in x
  double d = 3.0 / 4.0; // the value 0.75 is stored in d
  ```

  This subject is discussed in more detail in the section on type casts.

- The **modulus (%)** operator (sometimes called the modulo operator) gives the remainder when integer division is performed. For example:

  ```
  x = 7 % 5;
  ```

  This would store a 2 in x (when you perform 7 / 5 as integer division, the remainder is 2).

  Look at these results for modulus operations -- can you see a pattern?

  ```
  0 % 3 is 0
  1 % 3 is 1
  2 % 3 is 2
  3 % 3 is 0
  4 % 3 is 1
  5 % 3 is 2
  6 % 3 is 0
  7 % 3 is 1
  ```

  What information could you determine about a number **n** by checking **n % 2**?

- The **addition (+)** and **subtraction (-)** operators work as you expect.

- The **assignment (=)** operator works as we've seen. It operates from right to left.

- The **combined assignment** operator is another short-hand operator. It performs the arithmetic first, and then the assignment. For example:

  ```
  x = 3;
  x += 7;
  ```

  After this executes, the value in x is 10. First the 3 is stored. Then 7 is added to x and the sum 10 is stored in x.

  `x += 7;` **has the same effect as** `x = x + 7;`
  `x *= 7;` **has the same effect as** `x = x * 7;`
  etc...

2.2 Operators and Operator Precedence

When an expression makes use of more than one operator, the **rules of operator precedence** are used to determine the order in which operations are performed. Most likely, **these rules will seem natural to you because they match your previous experience with algebra**. Consider the following **algebraic equalities** (these are NOT Java statements):

Given x = 2
y = 3 + 4x

What is the value of y? I'm sure you concluded that y equals 11. You instinctively understood that the multiplication operation needed to be performed before the addition. The same is true in Java -- we would say that the multiplication operator has a higher **precedence** than the addition operator. The operators in the table at the beginning of this page are listed in order of precedence from highest to lowest; each row represents a precedence level.

However, note that there are only 4 precedence levels in the figure. Operators within each group (e.g. + and -) are said to have the same precedence. Basic arithmetic operators with the same precedence level are evaluated from left to right as they appear in the expression. So, ...

```
int z = 8 / 2 * 4; // z is assigned the value 16
// going from left to right, you divide 8 by 2 first,
// then multiply by 4
```

As in algebra, parentheses may be used to override the order of operations, and nested parentheses are always evaluated from the innermost to the outermost. Only this style of parentheses may be used: ()

You can find activities with numeric expressions on the Practice-It site.

IMPORTANT POINT: An assignment statement is not the same as an algebraic equality.

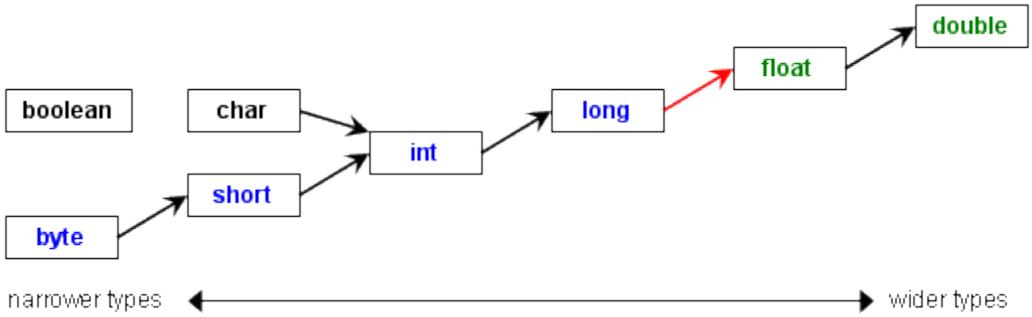
You could never write x = x + 1 in algebra, but this is an extremely common construction in programming. We would read this statement "x **gets** x plus 1". That is, "calculate the value 'x + 1' and store the result in the variable x". The Java '=' operator is so different from mathematical '=' operator that we use a different word when we read code containing this operator! Statements that begin with a construction like 'x = x ...' are very common in programming because we often need to change the value of a variable to a new value that is based on its old value.

2.3 Relationships between Numeric Data Types: Widening

Any operator that takes 2 operands is called a **binary operator**. When the two operands of a binary operator have **different** data types, the expression is called a **mixed-type expression**. Some mixed-type expressions are allowed in Java. However, a binary operator can only operate on 2 values of the same data type. So, for the operation to proceed, the data type of one of the values needs to be temporarily changed. This change is called **widening**.

The concept of the 'width' of a primitive type is a way of describing the range of values that can be represented. The greater the range of values that can be represented, the 'wider' the type. Widening is the process of changing **a value's** data type from a narrower one to a wider one (realize that the value is widened, not the variable). Widening is performed automatically as needed because it is considered to be a "safe" conversion. Consider widening from type ***int*** to type ***long***. Every possible value in type ***int*** can also be represented in type ***long***, so there is no possible loss of

information converting from type ***int*** to type ***long***. The diagram below shows the primitive types and the allowable widening operations, represented by arrows pointing from one data type (the narrower) to another (the wider).



One arrow in the figure above is colored red. Widening from type ***long*** to type ***float*** is legal because the range of possible ***float*** values includes all possible ***long*** values. However, some loss of precision is possible in such a conversion because a ***long*** value can have as many as 19 significant figures, whereas a ***float*** is limited to about 7.

Let's look at some examples:

```
double x = 3.4;
long z = 5;
```

Widening occurs in the second statement. The literal 5 (type ***int***) must be widened to ***long*** before the assignment can proceed. Continuing on:

```
x = x + z;
```

Widening: the value of **z** must be widened to double before the addition can be performed. Note that this has no effect whatsoever on the ***variable*** **z**. Rather, the value is extracted from **z** and then converted before the addition is performed.

```
x = 2.25 + 3 / 4;
```

The result here may not be what you expect. First, division is performed. Since both operands are type ***int***, integer division is performed and the result is 0. Then, before the addition can be performed, the 0 is widened to type ***double*** (0.0). The value stored in **x** is 2.25, type ***double***. So, all values in an expression with multiple operands are not automatically widened. Rather, you must evaluate the expression one operation at a time, using the rules of operator precedence. Only when a binary operator sees a type mismatch is widening performed.

2.4 Explicit Type Casts: Narrowing

Widening is performed automatically as needed, but narrowing is not automatic because narrowing involves a possible loss of information. However, a programmer may force narrowing to occur by using an **explicit type cast** (often just called a **cast**) in an expression. A type cast is produced by placing a data type inside parentheses to the left of a variable or expression. Here are two points to keep in mind when evaluating code with type casts:

- When a floating-point number is cast to an integer value, **truncation** occurs.
- Type casts have a higher precedence than all arithmetic operations.

Here are some examples of expressions with explicit type casts, along with their values and data types:

```
double x = 12.8;
double y = 2.9;
```

`(int)x`	evaluates to 12, type *int*
`(int)x * 2`	evaluates to 24, type *int*
`(int)(x * 2)`	evaluates to **25**, type *int*
`(int)x * (long)y`	evaluates to 24, type *long*
`(long)x * (int)y`	evaluates to 24, type *long*
`x * (int)y`	evaluates to 25.6, type *double*

One common pattern used when it is necessary to round a double value to type int is this:

```
int z = (int)Math.round( x );
```

In this case, the type cast is needed because the version of the round method that takes a parameter of type double returns type long.

2.5 Operator Overloading: +

The plus sign (+) is said to be an **overloaded operator** because it performs different operations on different types of data. If the operands are type String, then the + operator performs string concatenation (joining together 2 strings to create a new, longer one). However, if the operands are numeric, this operator performs addition. (The division operator is also overloaded, as we saw with integers vs doubles.)

So, what happens with **+** when there is a mixed-type expression?

If either operand of the + operator is a String, then string concatenation will be performed. This may require that the other operand's value be converted to a String equivalent, which is always possible. This can be very useful, but can also cause confusion. Consider the following example:

```
int x = 3;
int y = 2;
System.out.println( "The value of x is " + x ); // concatenation
System.out.println( "The sum is " + x + y ); // concatenation
System.out.println( "The sum is " + ( x + y ) ); //inside the parentheses is a sum
```

This code prints the following in the terminal window:

The value of x is 3
The sum is 32
The sum is 5

Notice that the use of parentheses to force the addition to occur first solved a confusing problem.

3 Operator Precedence Table

The following table is not complete, but covers all the operators we are concerned with in this course. Note that we have not yet encountered some of these operators when this list is introduced during Week 1. They are included here for completeness; don't worry about the things you don't recognize right now.

Highest Precedence ...

parentheses; array subscript; member selection	`() [ ] .`
unary postfix operators	*`expr`*`++ `*`expr`*`--`
unary prefix operators	`++`*`expr`* `--`*`expr`* `+`*`expr`* `-`*`expr`* `!`
instantiation; type cast	`new (type)`
multiplicative	`* / %`
additive	`+ -`
relational and *instanceof*	`< > <= >=` *`instanceof`*
equality (sometimes referred to as relational operators)	`== !=`
logical AND	`&&`
logical OR	`||`
conditional	`? :`
assignment	`= += -= *= /= %=`

... Lowest Precedence

For more information about operators and operator precedence, consult:
http://download.oracle.com/javase/tutorial/java/nutsandbolts/expressions.html