Primitives and Arithmetic

Objects vs. Primitives
Types in Java are divided into two camps: objects and primitives. Objects are relatively complex structures, allocated with "new", accessed through pointers, manipulated with "." method calls. "Primitives" are much more simple. For example, the "int" primitive type can only store a single integer value, such as 4 or 27. We can declare a variable of type int, and that variable can then store a single int value. With primitives, there are no pointers, or "new", or dereference ".". Primitives are used as building blocks inside of objects, but on their own they are simple. This handout discusses the two very commonly used arithmetic primitive types: int and double.

int
The "int" type is a primitive type that stores integer values like 34, or 23467, or -237. We use the int type to count things that naturally come in round, integral numbers, such as a number of cups of coffee or a number of students or an amount of money (stored as an int number of pennies).

Ints are relatively small, taking up only 4 bytes of memory. Instance variables, locals, and parameters may all be declared as the int type. An int variable is like a box that can hold a single int value. The int type is "exact" – the value 3 is exactly 3, and adding 1 yields exactly 4. (Surprisingly, the floating-point types below do not have this property of exactness.) In code, an int value is just typed out like this: 234 – a value in the code like that is called a "literal". The int type can store values approximately in the range -2 billion...+2 billion. The exact range is -2147483648...2147483647.

Totally optional aside: where do those odd looking min and max int values come from? Inside the computer, an int is represented as a 32 bit binary number – 0's and 1's, with the leftmost bit representing the sign: 0=positive, 1=negative. So the largest possible int is a 0 followed by 31 1's, 01111111111111111111111111111111. That value is 2^31-1, which is 2147483647. These details about binary numbering are not something we normally need to know about to program a computer, and you don't need to know it for CS106A.

If arithmetic with ints should yield a value outside the +2 billion range, it "wraps around" to the other end of the range, resulting essentially in wrong answers from there on. Therefore, to use ints, the must be confident that all the values and intermediate values will be within the +2 billion range. Java also has a primitive type "long" that is like int but uses more space (8 bytes) and therefore can store bigger values, in the range +9^18 (9 times 10 to the 18th).

4 Arithmetic Operators
The 4 standard arithmetic operations work with pairs of ints: + (addition), - (subtraction), * (multiplication), and / (division). The multiplication and division operations have a
higher precedence than addition and subtraction. This means that multiplication and division evaluate in the expression before addition and subtraction. The expression does not just evaluate left-to-right. Instead, the higher precedence operators evaluate first, followed by the lower precedence operators. So in the "1 + 2 * 3" example above, the "2 * 3" evaluates first to 6, then "1 + 6" evaluates to give 7. We can add parenthesis to force the expression to evaluate in a particular order. So writing "(1 + 2) * 3" forces the addition to go before the multiplication, yielding 9. Dividing by 0 causes a runtime exception.

**Precedence Table – Version 1**
Here we see a partial table of the precedence of operators in Java – operators higher in the table evaluate first. A "binary" operator, such as "*", is one which takes two arguments. Binary operators at the same precedence level evaluate left-right. A "unary" operator is an operator that works on just one value, such as by putting a "-" in front of a number. Handy fact to remember: all the unary operators have higher precedence than all the binary operators. The dereference "." has the highest precedence.

- Unary 1 (operator on the right): `ptr.` (i.e. the "." dereference on a pointer)
- Unary 2 (operator on the left): `~expr` (i.e. negative), `(cast)expr`
- Arithmetic 1: `* / %` (multiplication, division, mod)
- Arithmetic 2: `+ -` (addition, subtraction)

**int Division Pitfall**
There is a classic problem with int arithmetic where the code will not yield the answer you expect. The problem occurs when dividing one int by another int to yield an int answer – for example, what is the value of the expression "3 / 2"? The result needs to be an int value, but the conceptually correct value, 1.5, is not representable as an int. In this situation, int division automatically "truncates" the result, dropping the "fractional" part of the answer that would go to the right of the decimal point. So "3 / 2" evaluates to the int value 1, and "22 / 7" evaluates to the int value 3. Int truncation happens whenever the values on both sides of the / are of type int or long.

**int Modulus (%)**
The "modulo" or "mod" operator (%) is closely related to int division (/). The mod operator returns the remainder leftover after doing division. So "3 % 4" is 3, "4 % 4" is 0, and "5 % 4" is 1. The result of modding by N is always in the range 0..N-1. As an example, the following table shows the result of modding values 0..10 by 4:

<table>
<thead>
<tr>
<th>N % 4</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 % 4</td>
<td>0</td>
</tr>
<tr>
<td>1 % 4</td>
<td>1</td>
</tr>
<tr>
<td>2 % 4</td>
<td>2</td>
</tr>
<tr>
<td>3 % 4</td>
<td>3</td>
</tr>
<tr>
<td>4 % 4</td>
<td>0</td>
</tr>
<tr>
<td>5 % 4</td>
<td>1</td>
</tr>
<tr>
<td>6 % 4</td>
<td>2</td>
</tr>
<tr>
<td>7 % 4</td>
<td>3</td>
</tr>
<tr>
<td>8 % 4</td>
<td>0</td>
</tr>
<tr>
<td>9 % 4</td>
<td>1</td>
</tr>
<tr>
<td>10 % 4</td>
<td>2</td>
</tr>
</tbody>
</table>
Suppose we have a variable "count" that is always supposed be in the range 0..3 When adding one to the count, we can mod by 4 to wrap the value back into the 0..3 range: count = (count + 1) % 4.

It is a run-time error to mod by 0 (just as it is a run-time error to divide by 0). There is not wide agreement in Computer Science what the % operator should do when one or the other of the values is negative. Although Java does handle that case, as a matter of style we will never write code that feeds negative numbers to %. Similarly, it is traditional to only use int values with %. Java supports floating point %, but that is an odd feature and we will not use it.

double Type

The "double" and "float" primitive type stores what are called "floating point" numbers, such as 3.14 or -6.0 or 123.01. The double type is the most commonly used floating point type. Doubles use 8 bytes of memory, and have approximately 15 digits of accuracy. The rarely used "float" type is like double, but uses less space (4 bytes) and is less accurate (about 7 digits). Double literals are distinguished from int literals in the source code by the presence of the decimal point, such as 2.0 or 3.0E27 (i.e. 3.0 times 10 to the 27th power). It makes sense to use floating point types to store values that vary continuously, such as a distance or a velocity.

Floating point types have one great peculiarity: they are fundamentally imprecise. The double value 6.0 is not stored as exactly 6, but includes a small error term, so the actual value in the computer is something like 6.0000000000000021. Because of this small imprecision, it does not make sense to use == to compare double values. Relative comparisons, such as < work correctly with doubles (we'll talk about this in detail in the discussion of boolean expressions). The lack of precision is a cost due to the way floating point values are stored. The resulting benefit is that the range of representable values for the double type is inconceivably large: +-1.0E308 (i.e. 10 to the 308th power), compared to just +-2 billion for int.

Double Operations, Promotion

The 4 arithmetic operations work with doubles and without the confusion of int division. With doubles (1.0 / 2.0) is just 0.5 as we expect. If an operator, such as + or /, combines an int with a double, then the int value is automatically "promoted" to double before the operation proceeds. So with "2 * 3.0", the 2 is promoted (converted) to 2.0 before the multiplication, which then yields 6.0. So "2 / 4" yields 0, but "2 / 4.0" yields 0.5, since the 2 is promoted to 2.0 before the division. So the int-division pitfall only happens when we have a 2 ints with a /.
Types in Expressions

Every value, variable, and method return type in Java has a well defined type in the source code – int, or double, or String, or whatever. Therefore, you can look at any expression in the source code and look up the types of the parts of the expression to see what sort of promotion and truncation is going to happen. This is exactly what the compiler does when translating your expression – looking up the type of each part of the expression to guide its translation. Suppose have the following expression in a method...

\[ \ldots \text{getValue()} / (\text{myFoo} + 6) \ldots \]

Suppose getValue() returns int, and myFoo is of type int. Then there is no promotion, and the whole expression is done with ints, the division truncates, and the result is an int. Suppose instead that myFoo is a double. Then the 6 will be promoted to 6.0 before the addition, and likewise the getValue() will be promoted to double before the division and the result is a double.

int Truncation Divide/Multiply Trick

Suppose we have an int percent variable that is in the range 0...100. We have an image that is 810 pixels high, and we want to scale it to be percent of its natural height. Here are two ways to compute the desired height...

```java
int percent = <some int in the range 0...100, e.g. 50>

// figure height as percentage of 810
// version 1 - bad
int height1 = 810 * (percent / 100);

// version 2 - good
int height2 = (810 * percent) / 100;
```

Q: what is the problem with version 1? A: int truncation. The (percent/100) is evaluated first, and for any percent less than 100 it will yield the value 0. This is a common problem of any scaling problem that tries to produce a value between 0 and 1 using int division – you just get 0 most of the time. A frequently effective workaround is to do multiplication first, saving the division for last. In this way, the numerator is nice and big, and so the int truncation is not a problem. For example, suppose percent is 50. Then 810*50 yields 40500, and then dividing by 100 yields 405 which is correct. For this trick to work, the multiplication must yield an intermediate value that is within the int +2 billion range. Alternately, we can use doubles (next section) to do the scaling. The divide-last trick is immortalized in the famous old seaman's rhyme...

Multiply then divide, let your smile grow wide

Divide then multiply, bitter tears you will cry

(note: not actually ancient, or famous, or from the sea, but it does rhyme!)
Expression Example Code

```java
public class Expressions {

    public static final int PIXELS = 10;

    public static void main(String[] args) {
        int i = 1;
        i = i + 2;
        System.out.println(i); // print value of i (3)
        System.out.println("i:" + i); // + to combine string and int

        int left = 4;
        int right = 6;
        int width = (left + right) * PIXELS;
        System.out.println("width:" + width); // 100
        int width2 = left + right * PIXELS;
        System.out.println("width2:" + width2); // 64

        int a = 1;
        int b = 2;
        int complex = a + b * 3 - a * b + 3;
        System.out.println("complex:" + complex); // 8

        // division pitfall
        int num = 22;
        int denom = 7;
        int div = num / denom;
        System.out.println("div:" + div); // 3 (not 3.14)

        int val = 7;
        // suppose val is in the range 0..10, and we want
        // to compute a proportionate height in the range 0...100
        // here get int div problems
        int height = (val / 10) * 100;
        System.out.println("height:" + height); // 0

        // here do the multiply first, so it works
        int height2 = (val * 100) / 10;
        System.out.println("height2:" + height2); // 70

        // mod %
        System.out.println("3 % 4:" + (3 % 4)); // 3
        System.out.println("4 % 4:" + (4 % 4)); // 0
        System.out.println("5 % 4:" + (5 % 4)); // 1
        System.out.println("6 % 4:" + (6 % 4)); // 2
        System.out.println("7 % 4:" + (7 % 4)); // 3
        System.out.println("8 % 4:" + (8 % 4)); // 0

        // double type
        double d1 = 7.0;
        double d2 = (d1 / 10.0) * 100.0;
        System.out.println("d2:" + d2); // 70.0
    }
}
```
// int 10 "promotes" when mixed with double
double d3 = (d1 / 10) * 100;
System.out.println("d3:" + d3); // 70.0

int i1 = 7;
double d4 = (i1 / 10) * 100; // problem: still get int / division
System.out.println("d4:" + d4); // 0

int i2 = 7;
double d5 = (i2 / 10.0) * 100; // one fix: use 10.0
System.out.println("d5:" + d5); // 70.0

int i3 = 7;
double d6 = ((double)i3 / 10) * 100; // alt fix: (double) cast
System.out.println("d6:" + d6); // 70.0

}   
}

**int vs. double Type**

It is fine to store an int value into a double variable. No intervention is required; the int value is automatically promoted to double. However, the reverse direction does not work. It is not valid to store a double value into an int variable. This makes some intuitive sense – how could the double value 3.5 store into an int variable which is only capable of representing round values such as 3 or 4?

double d;
d = 5;       // works fine – the 5 is promoted to 5.0 automatically

int i;
i = 3.5;    // ERROR, does not compile. Cannot store double value into int

**int Cast**

In Java, we can "cast" an expression of one type to be another type using the desired type in parenthesis like this...

int i;

// the "(int)" cast converts 5.5 to 5
i = (int)(2 + 3.5);

// NO, does not compile
// the double expression on the right is not compatible with the int var
i = 2 + 3.5;

A cast is a unary operator – working on the value to its right and changing its type. Like other unary operators, cast has a very high precedence. Casting a double value to int works by truncating off the fractional part, so 5.5 becomes 5, -7.5 becomes -7. The precedence of cast is higher than the precedence of +/-*, so the following form does not work..
double Cast
We can cast an int value to double to avoid the int-division problem...

double d1 = (double)1/2;  // 0.5

double d2 = ((double)1)/2;  // 0.5

double d3 = (double)(1/2);  // 0.0

Cast Example Code
public void casts() {
    int seven = 7;

    // the old int division problem
    int i = seven / 4;
    System.out.println("i:" + i);

    // this works fine -- can store an int into a double
    // with no extra syntax. The int value is automatically
    // "promoted" to double. The reverse direction, double->int
    // is not automatic.
    double d1 = seven;
    System.out.println("d1:" + d1);  // 7.0

    // storing into a double does not fix the int-div problem.
    // The promotion to double happens AFTER the int division.
    // (we want the value 1.75, but we get the value 1.0)
    double d2 = seven / 4;
    System.out.println("d2:" + d2);  // 1.0

    // does not work -- identical to above, but puts in the cast explicitly
    double d3 = (double) (seven / 4);
    System.out.println("d3:" + d3);  // 1.0

    // works right, since the precedence of cast
    // is higher than precedence of /
    double d4 = (double) seven / 4;
    System.out.println("d4:" + d4);  // 1.75

    // works right -- identical to above, but parens added for readability
    // This is the way I would really write it. Note also that I removed
    // the space after the cast to emphasize that it happens first.
    double d5 = ((double)seven) / 4;
    System.out.println("d5:" + d5);  // 1.75
// does not work -- just like 'd2' above
// The cast on the 'seven' in the expression above does not change its type
// permanently. The cast is just a transient part of the above expression, so
// in the expression below, the seven is still of type 'int' so we have
// the same old problem.
double d6 = seven / 4;
System.out.println("d6:" + d6); // 1.0

i = seven * 2; // sure, can store int into int

// NO, does not compile, cannot store double value 10.5 into int
// i = seven * 1.5;

// NO, does not compile, but it's close.
// Does not work because precedence of cast is higher
// than *, so cast happens to the seven THEN it does the * double multiply
// i = (int) seven * 1.5;

// correct -- multiply first, then cast result to int (will truncate 10.5 to 10)
i = (int)(seven * 1.5);
System.out.println("i:" + i); // 10

// More sophisticated than (int) truncation -- use standard Math.round()
// Math.round() returns a 'long' which then must be cast to (int)
// Note that the precedence of '.' is even higher than (cast), so the method
// call happens first.
i2 = (int) Math.round(seven * 1.5);
System.out.println("i2:" + i2); // 11