Yes, this stuff is also on the exam.
Know it well.
Bring your questions to class.

Types of Data:

Programming in 15-102 has and will continue to require us to store data for use by our programs. Processing comes from the "factory" already "trained" to store and work with 8 types of data: 4 types for integer values, 2 types for fractional values, 1 type for characters, and 1 type to store the values **true** and **false**. These 8 "built-in" types are often referred to as "primitive types".

For our class we will use on a regular basis only four of these: **int**, **float**, **char**, and **boolean**. On a very rare occasions we will use another type named **long**. We will explore the differences and similarities to these types later after you have some experience using them .

In case you have forgotten or have not read the earlier notes here is a brief review of the types **int** and *float*.

The type, **int** is used for storing whole or integer numbers. There is a limited range of values that we can store using variables of type **int**. This range is roughly negative 2.7 billion to positive 2.7 billion. You may be thinking that this is more than enough but it may not always be sufficient.

The type *float* is used for fractional numbers such as 3.14159. There is also a limited range of values that we can store with variables of type **float** but, like **int**, the range is usually sufficient.

The thing to remember about **float** values is that they are approximations. Round-off errors occur for some fractional values. The common examples are the fractional values 1/3 and 2/3. These have no exact decimal equivalent. In 15-102 we usually do not worry about this type of error.

If we declare a variable that is not inside a function's braces, this variable can be described as a global variable because it can be used by the entire program.

Global variables of type **int** *are initialized to* **zero** *by Processing. Global variables of type* **float** *are initialized to* **zero point zero** *by Processing.*

Rules of the Road for Using int and float Variables:

You can use an **int** *value as a substitute for a* **float** *value. This will compile and work properly in Processing:*

```
float x = 42;
int number = 12;
float y = number;
```

Processing just adds a .0 to the end of the **int** *values so the result of the three lines of code above is that x has a value of 42.0 and y has a value of 12.0.*

You cannot use a **float** *value as a substitute for an* **int** *value without some extra syntax.*
The following lines of red code are illegal and will not compile

```
int x = 42.1;
float number = 12.5
int y = number;
```

If this were legal in Processing, what would happen to the fractional parts in the first and third line of code? They would be lost. Such a loss of data is not acceptable in most programming languages without extra syntax.

In case you are wondering what the extra syntax is, it would look like this:

```
float number = 12.5
int y = int(number);
```

Processing has a function named **int( )** *that will take a* **float** *as a parameter and return an* **int** *value that is truncated (NOT ROUNDED).*

Most of the graphics functions in Processing take **int** *and* **float** *values. However, the system variables height and width are* **int** *variables.*

Arithmetic and Types

The rule is fairly straightforward

int + int → int	float + float → float
int - int → int	float - float → float
int * int → int	float * float → float
int / int → int	float / float → float
int % int → int	float % float → float

But what happens if we used mixed types with these operators. Again, the rule is straightforward:

int + float → float	float + int → float
int - float → float	float - int → float
int * float → float	float * int → float
int / float → float	float / int → float
int % float → float	float % int → float

In this mixed type mode of arithmetic, the result of the evaluation becomes a float at the point the evaluation involves the float. Thus this expression:

```
7 + 3 - 2.0  + 4
```

evaluates like this:

```
7 + 3 - 2.0  + 4
  10 - 2.0  + 4
       8.0  + 4
              12.0
```

We need to look at the division operator / for **int** values because we can get into trouble and not understand why.
These are fairly easy:

```
8 / 4
12 / 3
```

The results are 2 and 4 respectively. But what about this one?

```
2 / 5
```

Remember that if both operands are **int**, the result must be in **int**
So what is the value?

The answer is **0**. That's right, **zero**. The / operator for two **int** values results in the quotient. The remainder is lost.

You need to know this because can cause you grief (as it has done to several of you…). Let's assume you want to use two thirds of the width of the window as the width of a rectangle. This works just fine:

```
rect( x, y, width*.66, height*.2);
```

What about this?

```
rect( x, y, width*2/3, height*.2);
```

It works just fine.
What about this?

```
rect( x, y, 2/3*width, height*.2);
```

It compiles and runs but you will not see much. Look closely at the third parameter:

```
2 / 3 * width
```

Let's trace the evaluation:

```
2 / 3 * width
    0 * width
        0
```

The order of operations of division and multiplication is left to right. The first evaluation is to divide 2 by 3. Since both operands are **int** values, the result is the quotient, which, for 2/3 is **zero**. Multiplying the width by zero results in a rect with a width of zero pixels. As stated above, this runs but you do not see very much on the screen. This particular nastiness is why

we used multiplication to get different fractions of the width and height on the days that were focused on homework #2.

What is the % operator - nothing has been said about it:
The % operator has nothing to do with percentages. It is also a division operator. While it works with **floats**, *we rarely ever seen it used with them. In 15-102 we will use it exclusively with* **int** *values.*

Wait a minute - you may be thinking that we already have a division operator which is the **/**
and you are correct so

```
2 / 5  ➔ 0
```

results in the quotient which is **0**. *What about the remainder? The* **%** *operator produces the remainder. So:*

```
2 % 5  ➔ 2
```

results in the remainder which is **2**

When we use the % operator, we get the remainder of division. This operator is often called the "mod" operator. **2 % 5** ***is frequently read as "2 mod 5"***

If you do not remember this, do the division the way you first learned to divide:

```
      _____
5 ) 2
```

gives us this:

```
    _ 0_r_2
5 ) 2
    0
    2
```

The quotient is **zero** *and the remainder is* **2**

If we use the **%** *operator on an* **int** *value, the result will always be in the range of* **0** *to one less than the value of the divisor*

```
anyNumber % 99  ➔ [0 .. 98 ]
```

anyNumber % width ➔ [0 .. one less than the width]

Oh Great!! More mindless trivia about programming - just when or why would I ever use this???

It turns out that we can use the **%** *operator to keep any number within a specific range.*

There are different ways we can get numeric input into our programs:

1. *arithmetic*
   x = x * 100;
2. **x** *equals user input from the keyboard or mouse*
3. **x** *= a random value returned by the* **random( )** *function*

Suppose we have to keep the x value of the center of a circle between 0 and 100 regardless of the value of the variable **x***?*

We can use the % *operator to do this.*

```
ellipse( x%100, y, height*.01, width*.01 );
```

The first parameter will always be between 0 and 99. It will never be 100 or larger because the **%** *operator returns the remainder. And the remainder of* **x % 100** *is always between zero and one less than the divisor which is 100.*

Using an "offset"

Suppose we have a window that is 300 pixels wide and we want to keep a circle in the middle third of the window - between 100 and 200 pixels.

This problem introduces an idea we call the "offset." Instead of generating a value between zero and 100, we need a value between 100 and 200. The number of values in the range we need to maintain is 100:

(200 - 100).

One way to do this is to generate a value between 0 and 100 and then add 100 to the result. This added value of 100 is the offset. By generating a value between 0 and 100 and then

adding 100 to the result, we are assured of a final result between 100 and 200. Note that we may be off by one pixel on either or both ends. For some computing applications (such as landing an airplane or performing hip replacement surgery) this is not acceptable. For 15-102, it usually does not matter.

Here is how our code might look:

```
    x = user-input-that-may-not-be-in-the-proper-range;
  ellipse( ( x % 100) + 100, y, height*.01, width*.01 );
```

Let's assume that the user entered t value 213 for the variable x from the keyboard, Here is the evaluation:

```
  ellipse( ( x % 100)     + 100, y, height*.01, width*.01 );
  ellipse( ( 213 % 100) + 100, y, height*.01, width*.01 );
  ellipse( ( 13) + 100,          y, height*.01, width*.01 );
  ellipse(          113,          y, height*.01, width*.01 );
```

The **%** *operator used in this manner keeps the circle in the middle third of the window.*

Wait a minute - you used a magic number for the offset..

Ok - we can cure that with this code:

```
 ellipse( ( x % (width/3)  ) + 100, y, height*.01, width*.01 );
```

Note that the parentheses can get out of hand very quickly. We strongly suggest you type both the open and closing parentheses and then go back inside of them to key in the expression.

In Closing:

This is about as complex as 15-102 gets in term of arithmetic. However, this is very important.

The correct evaluation of expressions involving float and int values with all five operators is very important to you in your projects AND your exams.

Work through the class code and bring your questions to class if you are not sure how this stuff works.

This is VERY important.

Animation in Processing

Processing provides a "cheap[1]" way to use animation in our code. From last week we know that Processing will execute a **setup( )** function if we have on in the code. It will be the first function that Processing executes.

We also know that if we have a **draw( )** function in the code, Processing executes it when the execution of **setup( )** is finished.

This is only partially true. Processing executes the **draw( )** function over and over until we stop it either by closing the window or calling an API function named **noLoop( )** to stop it. Each newly drawn version of the window is called a *frame*.

If we do nothing special, the new frame is drawn over top of the previous frame. This can produce some interesting results. Read on…

Along with the system variables **width** and **height**, Processing tracks the mouse location with system variables named **mouseX**, **mouseY**, **pmouseX** and **pmouseY**. The variables **mouseX** and **mouseY** are the location of the mouse in *this* frame. The variables **pmouseX** and **pmouseY** were the location of the mouse in the *previous* frame. Using the following code:

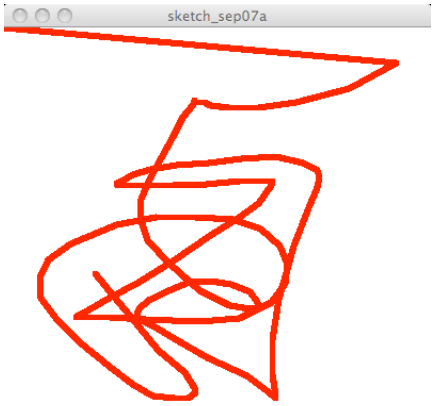
```
void setup( )
{
   size( 400, 400 );
   stroke( 255,  0, 0 );
   strokeWeight( 5 );
   background( 255 );
}

void draw( )
{
   line( pmouseX, pmouseY, mouseX, mouseY);
}
```

here is the window after multiple iterations of the

[1] *by cheap, we mean easy… da' moose*

draw( ) function have produced multiple frames:

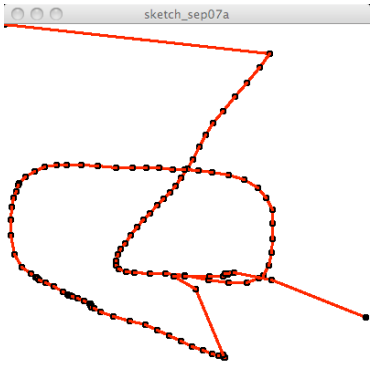


In this code each new frame is drawn on top of the previous frame. One iteration of the draw( ) function draws a line from the position of the mouse in the previous frame to the mouse's current position.

If we modify the code to draw a back point at the mouse's location in each frame like this:

```
void draw( )
{
   stroke( 255, 0, 0 );
   strokeWeight( 2 );
   line( pmouseX, pmouseY, mouseX, mouseY);
   stroke(  0 );
   strokeWeight( 6 );
   point( mouseX, mouseY);
}
```

we get this result:



Our list of system variables includes:

- **width**
- **height**
- **mouseX**
- **mousey**
- **pmouseX**
- **pmouseY**

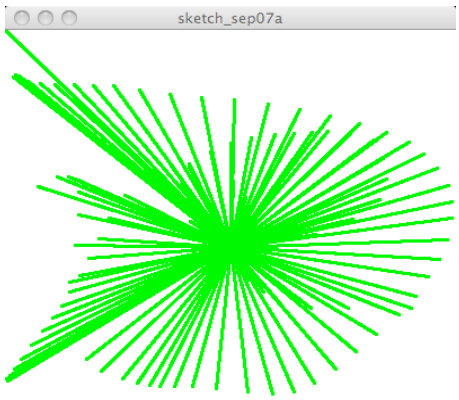
We can add two more:

- **frameRate** *which is the number of frames displayed each second or the number of times the* **draw( )** *function is executed each second.*
- **frameCount** *which is the total count of frames displayed thus far.*

There is a **frameRate( )** *function that allows us to alter the* **frameRate**. *This is confusing but we have to live with it. Look up the* **frameRate( )** *function in the API.*

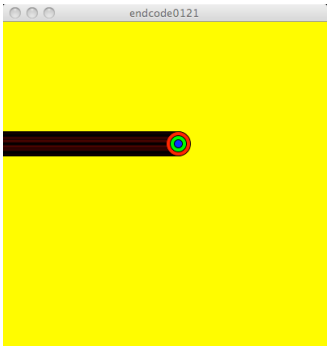
Here is another sample:

```
void draw( )
{
   line( width/2, height/2, mouseX, mouseY);
}
```



For some of our drawings, we will not want to see the old frame when the new frame is drawn. We will want just the new frame all by itself. Let's see why.

Here is Jim's target drawing code that uses the **frameCount** as the x coordinate value of the target to move the target from left to right on the screen. This version draws the new frame on top of the old one with the old one visible:



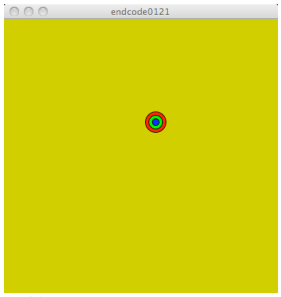
Each new frame draws the target one pixel further to the right. The new frame is drawn on top of the previous frame and this results in the black "smear" extending to the left edge. Here is the code:

```
void draw( )
{
   drawTarget( frameCount%width, 20, 20, 10);
}
```

The smear comes from the black stroke of the lartest ellipse in the target. If Jim wants to avoid this smearing, he needs to opaquely cover the previous frame. Unfortunately there is no **opaqueCover( )** function in Processing so he does what we do when we paint our room. We do not remove old paint when we paint, we just paint over it. So, Jim can paint over it by using the **background( )** function:

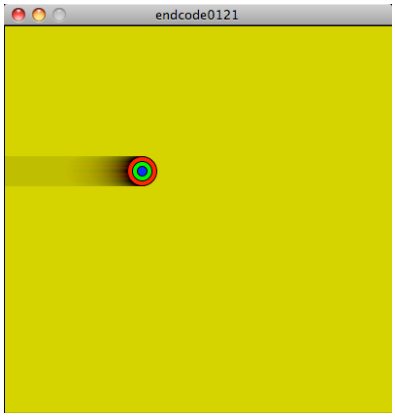
```
void draw( )
{
   background( 200, 200, 0 );
   drawTarget( frameCount%width, 20, 20, 10);
}
```

The new line of code covers the old frame with an opaque yellow background and allows the new target to be drawn on a clean yellow background:



It is difficult to show the animation in print but the target began on the left side of the window and moved across to the right. The black smear is gone.

There is a middle ground to explore. We have seen a long smear and no smear. What if we want a little bit of smearing to give the impression of movement. The effect is shown below:



This effect can be created by using what is called the alpha value or the alpha channel translucence. Here is the code that creates this effect:

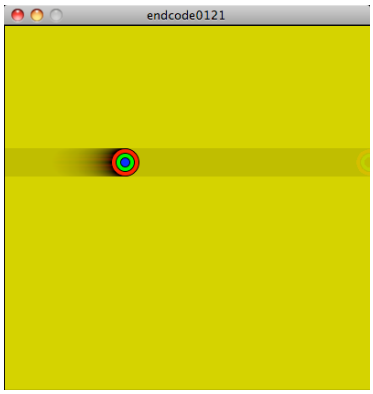
```
void draw( )
{
   fill( 200, 200, 0, 10 );
   rect( 0, 0, width, height);
   drawTarget( frameCount%width, 20, 20, 10);
}
```

Notice that the call of the **fill( )** *function has a fourth argument. This is the alpha value. This value can be between 0 and 255. The value of 10 creates the degree*

of transparency that allows us to see a part of the previous frame.

What is happening here is that each new frame begins with the drawing of a yellow rectangle[2] that covers the entire screen. This rectangle is not totally opaque. It has a small amount of transparency or translucence. The drawing of the multiple frames with this small amount of transparence eventually covers the older frames and leaves only a small percentage of the most recent frames visible. This creates the smearing which gives the impression of movement.

Use of the alpha value takes a lot of experimenting. Here is the same program after several transits of the target across the window:



The use of yellow as the background and black as the stroke around the target results in a smear that is never completely erased.

[2] *As far as we know,* ***background( )*** *is always opaque even if you use an alpha argument. We have tried but it does not work. If you get it to work, please let us know.*