CSCI 182: Introductory Programming for Media Applications

Introduction

(Chapters 1 and 2, p. 1-21)

“Moodle” Course Webpage

- http://learnonline.unca.edu/
- Use your **UNCA login & password.**
- Our class page is
  - “Computer Science 182.001: Introductory Programming for Media Applications: Whitley”
- Syllabus link at the top.
  - Course policies
  - Contact information
  - Textbook, etc

Computer Programming

- A (high level) *programming language* is a language used by people to tell a computer what to do.
- These languages are used to write *source code*, which consists of a series of *statements*: small tasks for the computer to perform, such as drawing a rectangle or adding two numbers.
- A *compiler* or *interpreter* is used to convert the human-written source code into low level *machine code* the computer can use.

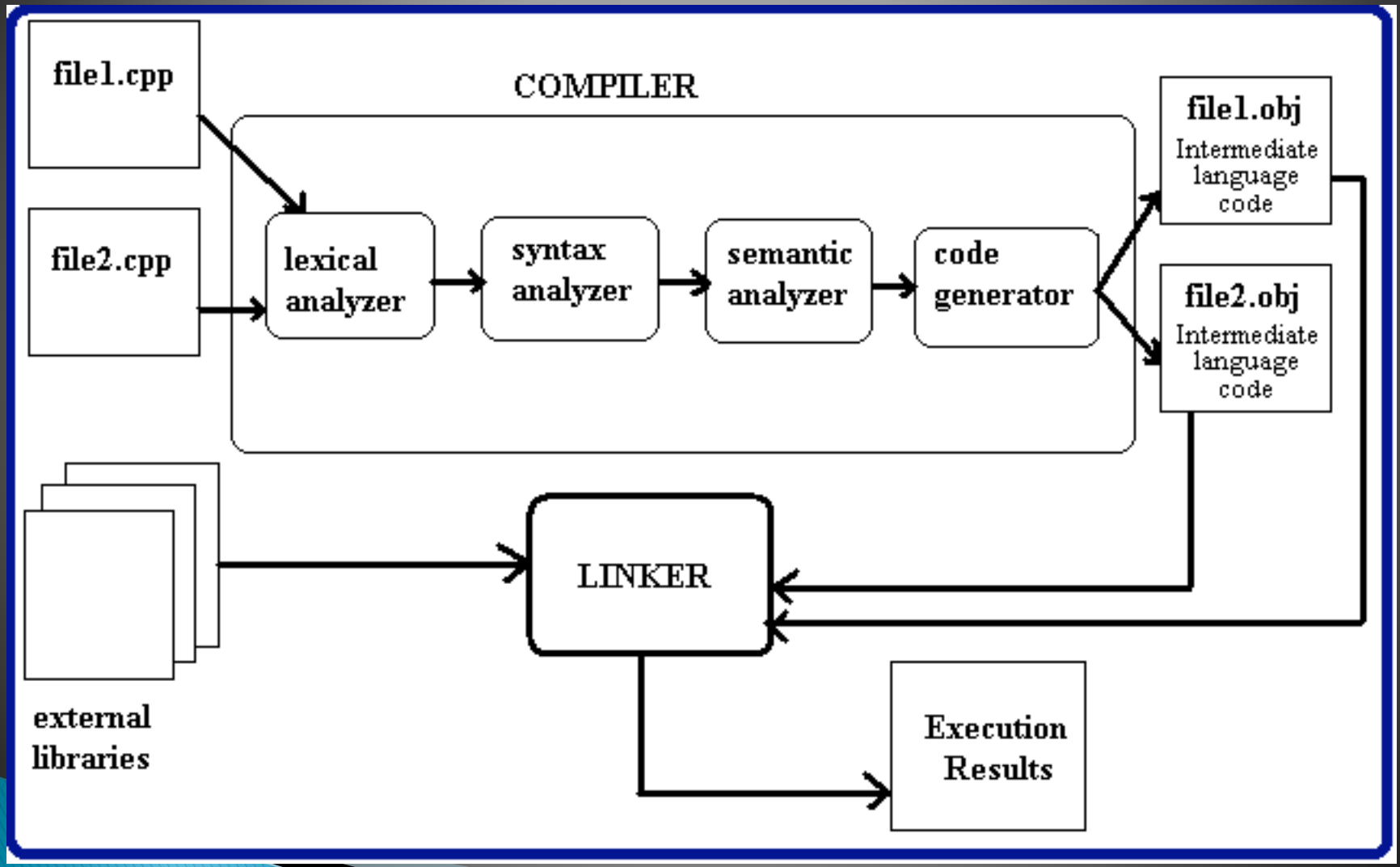
Computer Programming

- Machine code is written in binary, a series of 0's and 1's called *bits*.

0000 0000 0011 0001 0010 0011 …

- These bits are grouped into *instructions*: very simple actions such as moving a number from one place to another.
- Good news: we don't have to deal with this soup, because we have high level languages with compilers & interpreters!

A Generic Compiler



Java Source Code

- *Java* is a high level programming language. Here's how programs are made with Java:
- 1) A person writes Java source code and saves it into a .java file. Example:

```
/* this is a simple Java program */
class Example {
  public static void main(String args[]) {
     System.out.println("this is a simple Java program");
   }
}
```

Java Byte Code

- 2) The Java compiler (named *javac*) compiles the .java file into a .class file containing *Java Byte Code*, an intermediate level of code somewhere **between** Java source code and machine code. Example:

```
public static void
main(java.lang.String[]);
Code:
0: iconst_0
1: istore_1
2: goto 30
5: getstatic
8: new
11: dup
12: ldc
14: invokespecial #23
17: iload_1
18: invokevirtual #27
21: invokevirtual #31
…
```

The Java Virtual Machine

- 3) The *Java Virtual Machine* (JVM), also called the *Java Runtime Environment* (JRE), runs the Java Byte Code.
- The JVM is an *interpreter**. It takes each line of Java Byte Code one at a time, turns it into machine code, and runs it.
  - Picture recap:

  - Contrast with a compiler.

* a half-truth

Why bother with all that?

- Compiling Java is complicated, so why is it done this way?
- Every type of CPU ever made has different machine code instructions. You can't learn them all!
- Java Byte Code is **universal**. You can run it on Microsoft Windows, Solaris, Linux, Mac OS, …
  - All you need is the Java Virtual Machine on your computer, and Oracle gives that away for free!

Processing

- In this course, we'll learn a high level language called *Processing*.
  - Download free for your Windows, Mac, or Linux computer at http://www.processing.org/
- It's very similar to Java.
  - Think of it as an extra "layer" of code on top of Java.
  - Processing source code gets turned into Java source code, and given to the Java compiler.
- Processing has a lot of built-in features that make pictures, drawings, and animations easy.
- Easy to transition to Java later this semester, and next semester in CSCI 202.

The Processing Window

Menu & Toolbar

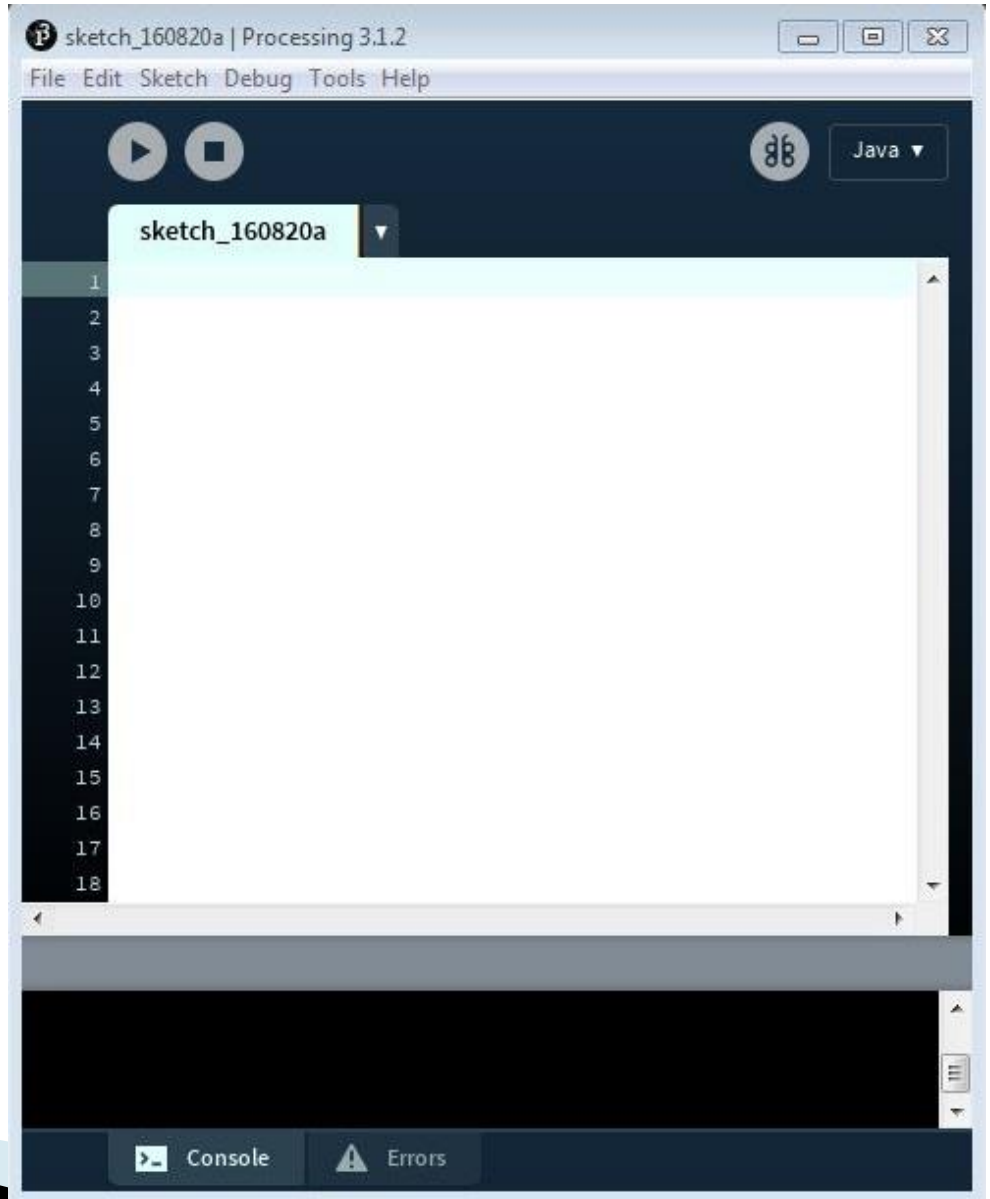
Run Button ⇨

Tabs

Text editor

Messages & errors

Console



Hello World

- Let's do a simple program.
- 1) In the text area, type

```
println("Hello, world!");
```

- 2) Press the Run button on the toolbar.
  - (It's a triangle pointing to the right.)
- Two things happen:
  - A message will appear in the console

    ```
    Hello, world!
    ```

  - A small square window will open
    - We'll call it the *display window.* It's where images and animations will appear later.

Debugging

- The *syntax* of a language describes the rules of how to write code.
  - It's like English grammar rules, but for computer programs.
- If you mistype something, or get the syntax wrong, the program will not run. These are called *syntax errors* or simply *bugs*.
- Example:

```
println("Hello, world!";
```

- *Debugging* is the art of fixing these bugs.

Comments

- You can add *comments* anywhere in your code. They have **no effect** on the program.
- Useful for making notes or describing the code to people. There are two kinds.
- One-line comments: after `//`

```
// This is a comment.  Oogabooga!
```

- Multi-line comments: between `/*` and `*/`

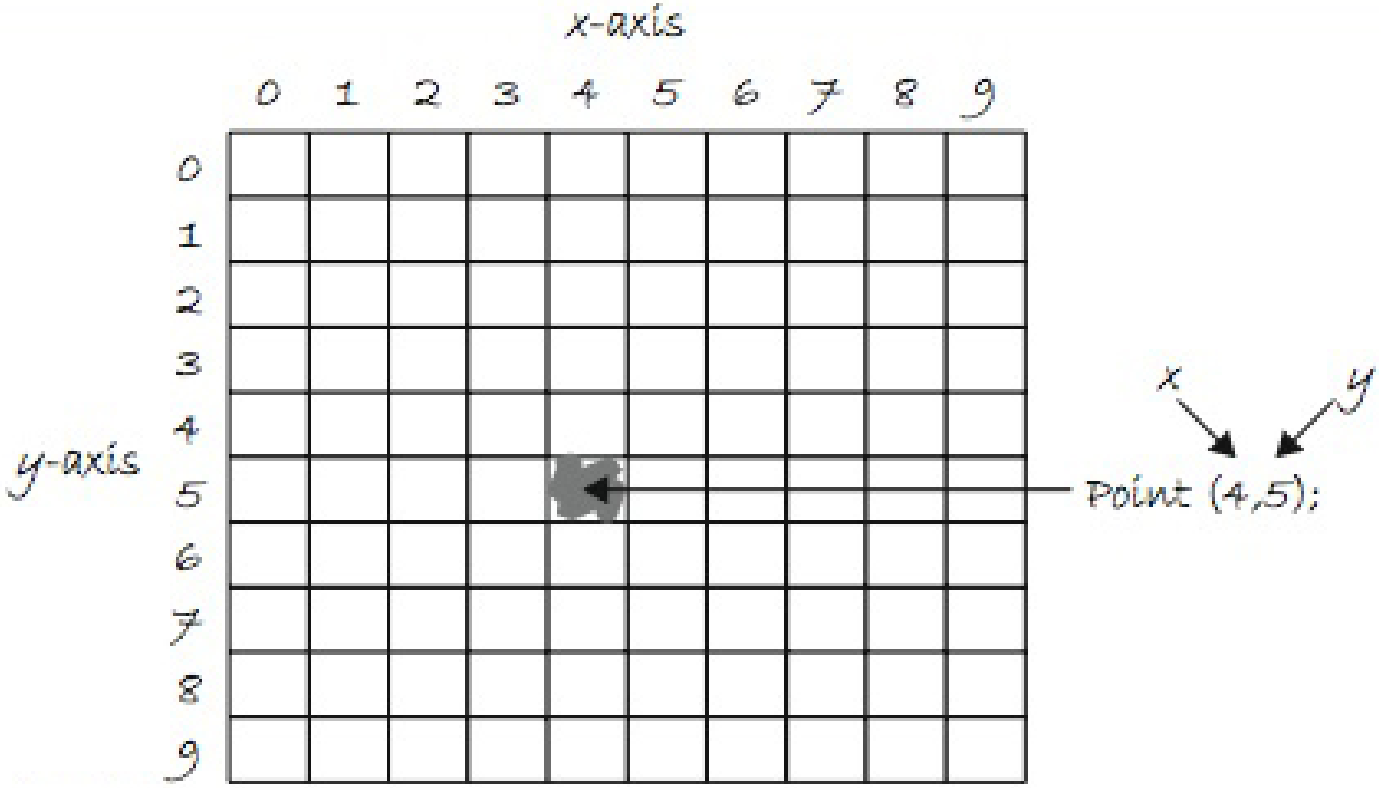
```
/* This is a larger comment
stretching over two lines. */
```

CSCI 182: Introductory Programming for Media Applications

Graphics Primitives / Shapes
(Chapter 6, p. 103-120)

The Display Window

- The *display window* is a coordinate plane for displaying drawings and animations.



The Display Window

- Every location in the display window is a *pixel* (a dot of light) with a unique x,y coordinate.
- The *origin* with coordinates (0,0) is at the top left corner.
- **Remember**: the positive y axis goes **down!**
- 100 by 100 is the default size. Change the size of the window with the `size` function.

```
size(400, 300); // 400 by 300 pixels
```

Points and Lines

- We can draw a point at any location in the display window with the `point` function.

```
point(40, 60); // x,y coordinates 40,60
```

- Lines can be drawn connecting any two locations with the `line` function.

```
line(40, 60, 80, 20);
```

- That line connects the first location (40, 60) to the other location (80, 20).

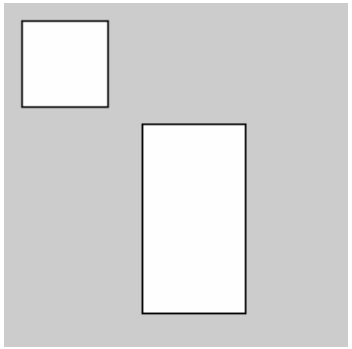
In-Class Lab 1

- Make a 300 x 300 display window.
- Draw at least 6 points and 6 lines.
- Arrange them into a composition by changing the coordinates.
- Raise your hand when you're done, or if you need help.
- I'll help everyone finish and get checked off.

Rectangles

- Draw rectangles and squares with the `rect` function.
- Provide 4 numbers:
  - x,y coordinates of top left corner
  - Width of the rectangle
  - Height of the rectangle

```
rect(10, 10, 50, 50);
rect(80, 70, 60, 110);
```



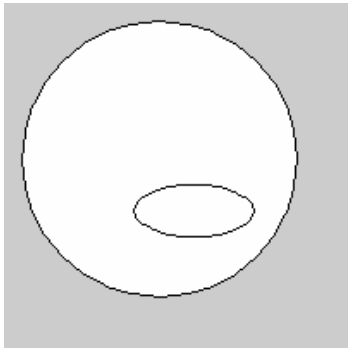
Rectangles

- You can change the way `rect` draws rectangles by using the `rectMode` function.
- `rectMode(CORNER);`
  - Default. Draw rectangles with the top-left corner x,y then width and height.
- `rectMode(CORNERS);`
  - Specify x,y coordinates of any two opposite corners.
- `rectMode(CENTER);`
  - Specify the x,y coordinates of the center, then width and height.
- `rectMode(RADIUS);`
  - Specify the x,y coordinates of the center, followed by half the width and half the height.
- You only need to call rectMode once. It's effect is **permanent** until the end of the program, or until you call it again to *override* the old value.

Ellipses

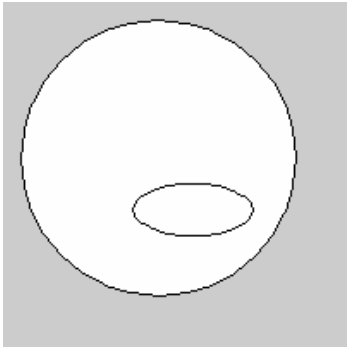
- Draw ellipses and circles with the `ellipse` function.
- Provide 4 numbers:
  - x,y coordinates of the center
  - Width of the ellipse
  - Height of the ellipse
- `ellipseMode` works just like `rectMode`, but the default is `CENTER`.

```
ellipse(90, 90, 160, 160);
ellipse(110, 120, 70, 30);
```



Order Matters!

```
ellipse(90, 90, 160, 160);
ellipse(110, 120, 70, 30);
```



- Why is the small ellipse in front?
- The code you write is *sequential*. Code is executed one line after the other, from top to bottom.
- The code for the large ellipse appeared above the other, so it was drawn **first**.
- Then, the small ellipse was drawn **on top** of whatever was already there.

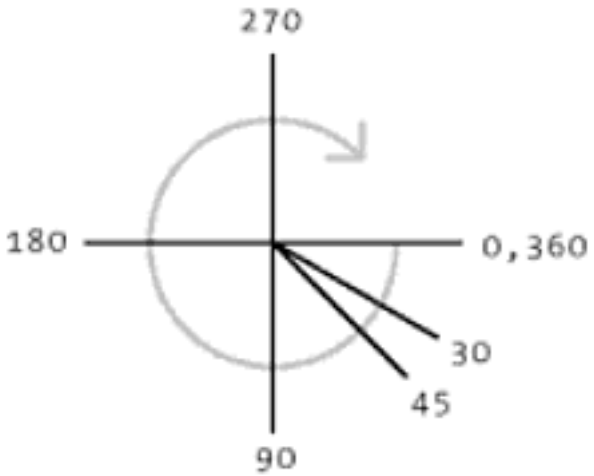
Arcs

- You can draw a part of an ellipse (like a slice of pie) with the `arc` function.
- Provide 6 numbers:
  - Same 4 numbers as an ellipse, then
  - The start angle and stop angle (in *radians*)

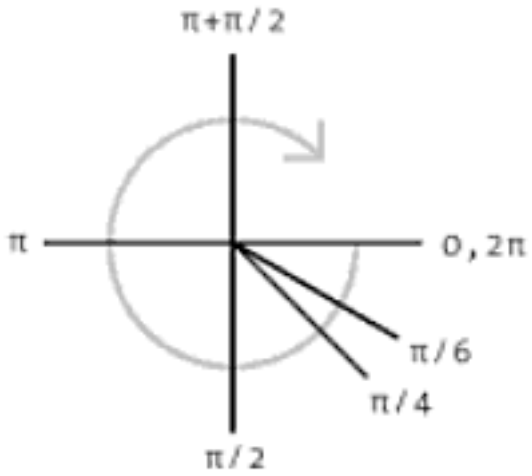
```
// Top half of a circle
arc(90, 90, 160, 160, PI, 2*PI);
// PI is a built-in number, approx 3.14159
```

Angles

- A circle has 360 degrees, or 2*PI radians.



Degree values



Radian values

Triangles & Quadrilaterals

`triangle(x1, y1, x2, y2, x3, y3);`

- Specify the x,y coordinates of all 3 endpoints.

`quad(x1, y1, x2, y2, x3, y3, x4, y4);`

- Specify the x,y coordinates of all 4 endpoints.

Bezier Curves

- We draw curved lines called *bezier curves* using the `bezier` function. (Extremely useful!)
- Provide 8 numbers:
  - x,y coordinates of the first end point, the first *control point*, the second control point, and finally the second end point

```
bezier(10, 50, 30, 90, 60, 70, 90, 10);
```

- Control points "bend" the bezier curve towards them.
- Imagine the curve begins at the first end point, then bends partway towards the first control point, then bends partway towards the second control point, then goes to the second end point.

RGB Color

- By default, we use *RGB* colors in Processing.
- Define a color as *red*, *green*, and *blue* components. Each component is an intensity between 0 and 255.
  - Bright red is (255, 0 , 0)
  - Dark green is (0, 120, 0)
  - Yellow is (255, 255, 0)
- *Grayscale* colors can be specified with only one component – the brightness.
  - White is (255, 255, 255) or simply 255.
  - Black is (0, 0, 0) or simply 0.

Transparency

- Put an extra component (called the *alpha* or *opacity*) onto the end of any color to change its transparency.
- 255 means opaque, and 0 means fully transparent. (Easy to get this backwards!)
- Examples:
  - (255, 0, 0, 150) is partially transparent red.
  - (0, 0, 255, 0) is completely transparent, with **no visible color.**
  - (150, 255) is a fully opaque gray, equivalent to plain simple 150.

Color & Effects

- Change the background color of the display window with the `background` function.

```
background(255, 0, 255); // purple!
```

- Change the color **inside** shapes with the `fill` function.
  - Remember: use 3 components for a color, 1 component for grayscale.
  - Transparency **doesn't work** on `background`. (Why?!? Who knows!)
- Change the color of lines, points, and the **border** of shapes with the `stroke` function.

```
// rectangle with black insides and a red border
fill(0);
stroke(255, 0, 0);
rect(10, 10, 50, 50);
```

Color & Effects

- Turn off fill color with `noFill();`
  - Shapes will be hollow on the inside, with no filling at all.
- Turn off stroke color with `noStroke();`
  - Shapes will have no separate outline.
  - Points and lines disappear completely!
- Using both `noFill();` and `noStroke();` makes all shapes invisible!
  - After all, a rectangle is just some filling with a border!

Color & Effects

- You can change the thickness of borders, lines, and points with the `strokeWeight` function.
- Specify a width in pixels – the default is 1.

```
strokeWeight(10);
line(10, 10, 90, 90); // thick line
```

- `smooth();` turns on *anti-aliasing* (smoothing).
  - Turn it back off with `noSmooth();`
- These functions all have **permanent** effects until you override it with a new value.
  - Ex: One use of `stroke(255, 255, 0);` and **all** future shapes will be filled with yellow, until the end of the program or you specify otherwise.

In-Class Lab 2

- Create a Processing program to display a composition using **at least:**
  - One colored ellipse
  - One colored rectangle
  - One colored bezier curve
  - More stuff, if you want!
- Don't worry about artistry too much – this is just code-writing practice!

Color Models

- You can change the *color model* in processing if you don't like the "RGB 255" default.

```
colorMode(RGB, 1000000);
```

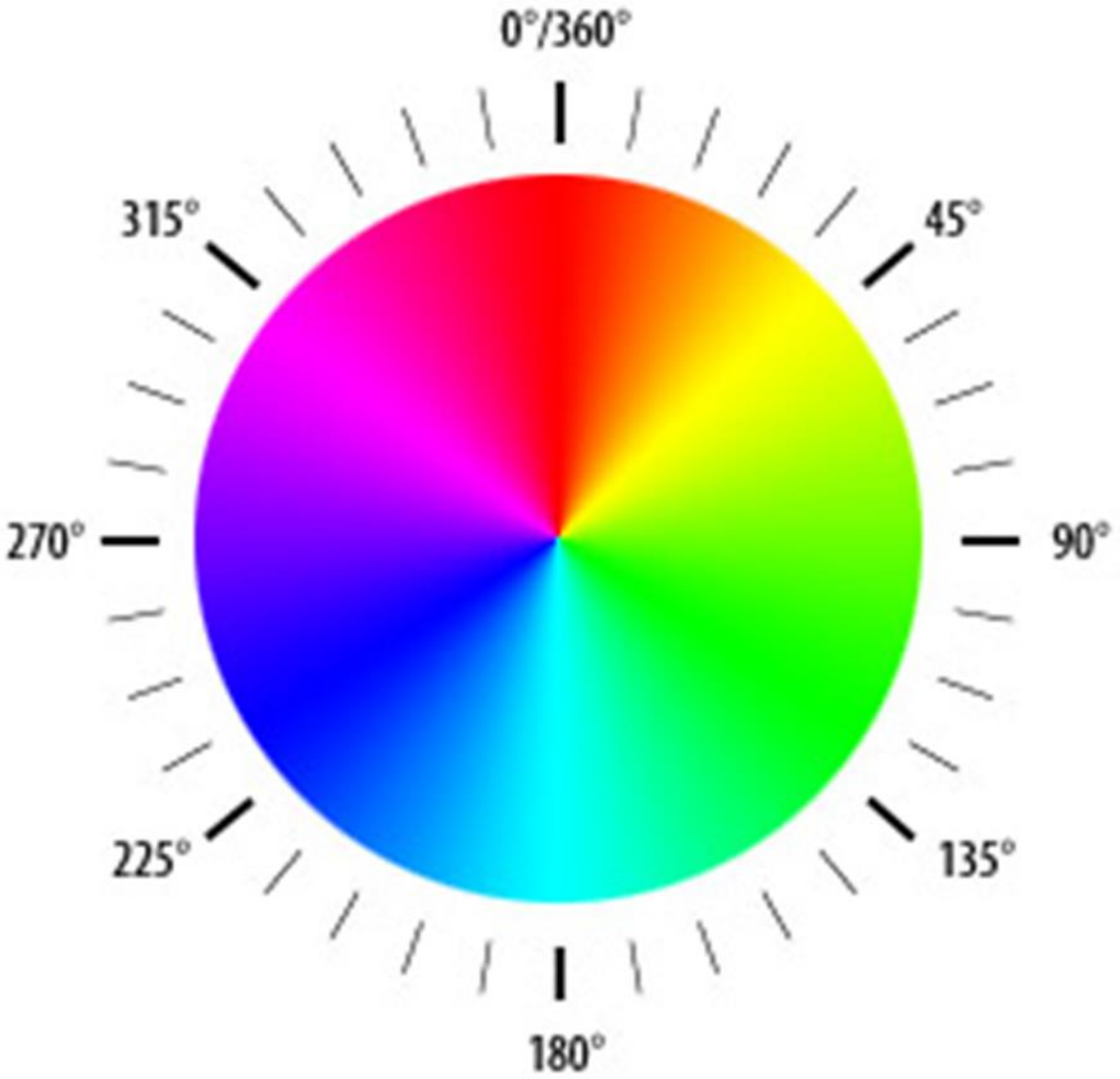
- Now, each red, green, and blue component is a number from 0 to a million!
  - This allows for much more fine-grained colors! Your video card and monitor have limits, though.

Color Models

- In the *HSB color model*, a color is defined by a *hue*, a *saturation*, and a *brightness*.
- A hue is like an angle on the *color wheel*.
  - Like having a rainbow wrapped around a frisbee!!!
- Saturation determines how "vibrant" or "colorful" a color is. Grayscale colors have 0 saturation, and bright neon colors have high saturation.
- Brightness is… well, you know. Brightness! Low brightness makes a dark color, and any color with a brightness of 0 is black.

```
colorMode(HSB, 360, 100, 100);
// maximum hue is 360. max saturation
// and brightness are 100 each.
```

Hue Color Wheel



CSCI 182: Introductory Programming for Media Applications

Variables

(Chapter 3, p. 23 – 42)

Computer Memory

- All data and information a program uses is stored in the computer's *memory*.
- Every location in memory has a unique number called its *address*.
  - Think of memory like a line of numbered PO boxes at the post office.
- Each memory location can store information.
- A *variable* is a name given to a memory location. We create and use them to store, retrieve, and modify information in memory.

Variables

- In order to use a variable, we must first *declare* it.
- Every variable has a *type* and a *name*. Example:

```
int x; // variable type int, named x
```

- The variable x can hold an integer such as 1337.
- There are many different types of variables, suited for different kinds of information.
- The simplest types are called *primitive types*.
  - Later, we'll learn about more complex types of variables called *arrays* and *objects*.

Primitive Types

- `boolean` holds a truth value: `true` or `false`.
- `char` holds a character – symbols such as letters of the alphabet and punctuation marks
- `byte`, `short`, `int`, and `long` are types that hold integer numbers.
  - `byte` can hold any number from -128 to 127
    - A byte is stored with only 8 bits, btw!
  - `short` can hold any number from -32768 to 32767
  - `int` can hold any number in the range of ~ 2 billion
  - `long` can hold truly enormous numbers!
- `float` and `double` are types that (sorta) hold real numbers (we call them *floating point numbers*)
  - `float` can hold ~ 8 significant digits, and exponents up to ~ 10^{38}
  - `double` can hold ~ 15 significant digits, and exponents up to ~ 10^{308}. More precise than a `float`!

Using Variables

- To declare a variable, say its type, then its name, then a semicolon. Examples:

```
int sum;
char c;
float number;
```

- You can even declare more than one variable of the same type, all at once.

```
int e, f, g;
```

- You can assign a value to a variable with an *assignment statement*.
  - It looks like a math equation with a semicolon, but it's not. The source is on the **right** side, and the destination is on the **left**.

```
sum = 7;
number = 3.1;
```

Using Variables

- You can give a variable a value at the same time you declare it. It's called *initializing* the variable.

```
int y = 7;   // y is declared, value 7
boolean z = true;
```

- You can even assign a value from one variable to another!

```
short a = 3;
short b;
b = a;                          // b gets 3
short e = b;                    // e gets 3
```

Variable Names

- You can name a variable almost anything you want. You may use
  - Alphabet letters, both uppercase and lowercase
  - Numbers
  - Underscores _
- But, the name **must begin with a letter.**
  - `thisIsAGoodName`
  - `fifty7`
  - `5seven` – illegal name, because it starts with a number!

Variable Names

- It's good style to:
  - Name a variable with a descriptive word or phrase.
  - Capitalize the first letter of each word, **except** the first word.
  - Don't use excessively long names.
  - Examples of good names:
    - `imageWidth`
    - `sumTotal`
    - `isComplete`
- Variables that you intend to never change are called *constant*.
  - For example, `PI` is a constant, of value ~3.14159…
- Constants are usually named in all-caps, with underscores for spaces.
  - `PI`, `CORNERS`, `CENTER`, `MAX_VALUE`

Strongly Typed vs. Weakly Typed

- Many programming languages (e.g. Java, Processing, C) are called *strongly typed*. When you declare a variable, you **must** specify its type, too.
- Some programming languages (e.g. Perl, PHP, BASIC) are called *weakly typed*, so variable types don't have to be declared before use. The type of a variable is **inferred** by what sort of thing is put into it.
  - "Oh, you put the letter 'R' into that variable? Ok, that means it's the type of variable that holds a character!"

Arithmetic

- You can do arithmetic with variables. Algebra works just like you'd expect, mostly…
- There are several *arithmetic operators*, most of which you already know:

+ is addition	3 + 5
- is subtraction	4 - 2
* is multiplication	6 * 8
/ is division	9 / 3

  - Division is a little strange…

% is modulus (a.k.a. "mod")	4 % 3

  - Mod gives the **remainder** of long division.

Arithmetic

- = is called the *assignment operator*, used to assign a value into a variable.

```
byte z;
z = 4 + 9; // z gets the value 13
```

- You can make arithmetic expressions out of *constants* like 17, as well as variables like `z`.

```
long k = 2;
long m = 10 * k + 1; // m gets 21
```

Arithmetic

- Whenever you do an arithmetic expression with variables of the same type, the result is **the same type.**
  - Sounds obvious, right? When the computer evaluates 1 + 2, the answer is 3, and its an integer because the operands are integer.
- This does strange things to division, though!
- 1 / 4 evaluates to an integer, because the two operands are integers.
- Since we can't store 0.25 in an integer, the computer *truncates* (rounds down), by **forgetting everything after the dot**!

Arithmetic

- 1 / 4 is **0**, and don't let anyone tell you different! *lol...*
  - It's called *integer division*.
- However, if at least one operand is a floating point number, the **result** will be floating point!
- So, 1.0 / 4 is actually 0.25
  - Called *floating-point division*.

Arithmetic

- `++` and `--` are unary (one-operand) operators for incrementing and decrementing.

```
int x = 5;
x++; // x is now 6
x--; // x is now 5
```

- There are also *compound operators* that do arithmetic followed by assignment.

```
x += 5;
x = x + 5; // identical to the above line
```

- The LHS and RHS (the two sides) are used as operands for arithmetic, then assigned into LHS.
- `+=` `-=` `*=` `/=` `%=` each do the specified math operation, then assignment.

Precedence Rules

- Operator precedence works just like in "normal" math.
- Anything in parentheses is evaluated first.
  - The "innermost parens" always have priority.
- Then * / and % are evaluated left to right.
- Lastly + and binary – are evaluated left to right.

```
int g = 7 + 3 * 6;                  // g is 25
int h = (4 + 2) % 3 + 2 / 3;// h is 0
```

Precedence Rules

Operator(s)	Operation(s)	Order of evaluation (precedence)
()	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses "on the same level" (i.e., not nested), they are evaluated left to right.
*, /, %	Multiplication, Division Modulus	Evaluated second. If there are several, they are evaluated left to right.
+, -	Addition Subtraction	Evaluated last. If there are several, they are evaluated left to right.

- Arithmetic operators are *left-associative*: evaluated from the left.
- The assignment operator = is *right-associative*: evaluated from the right.

Built-In Variables

- Processing has many *built-in variables* that are used without declaring them.
  - They have highly useful values, which automatically **always stay up-to-date**! They "already exist."
- `width` is the current width of the display window in pixels.
- `height` is the current height of the display window in pixels.
- We'll learn many more soon (teasers!), including `mouseX`, `mouseY`, and `frameCount`, which are used for animations.

Examples on Moodle!

In-Class Lab

- 1) Make a diameter-10 circle that always stays in the **middle** of the display window, no matter the size.
  - Hint: remember width and height built-in variables, and do arithmetic on the coordinates!
- 2) Make another circle that's **halfway** between the first circle and the origin.
- 3) Make 2 more circles exactly **25 pixels below** the first two.
- Raise your hand when you're done, or if you need help!

Implicit Type Conversion

- Every variable has a type. Only data of that type may be stored into it.
- You can assign between variables of the same type.
- You can also assign from a "smaller" type into a "larger" but similar type.
  - This is called *implicit type conversion*; it converts from one type to another automatically, without telling you.

Implicit Type Conversion

- In order from largest to smallest, the numeric primitive types are:
  - double
  - float
  - long
  - int
  - short
  - byte
- You can assign from any of those types to a higher type on the list.
- Rule of thumb: you can't assign when there is a possible **loss of information**, such as assigning a double to a float.

(Explicit) Type Casting

- *Type casting* allows you to break the rules of implicit type conversion. You can "force it" to assign, even if information is lost.
- So, if you use type casting to force the value 7.5 into an int, it will truncate and 7 will be stored.
- Put the type you want in front of the variable/value you want to change.
  - Works like a unary operator, but you **must** use parens.
  - There are 3 forms of correct syntax:

```
int i = (int) 7.5;
int j = int (7.5);
int k = (int) (7.5);
int m = int 7.5;      // Wrong!  Need parens!
```

(Explicit) Type Casting

- Type casting is **temporary**.
  - It does not change the type of a variable.
  - Type casting only makes it "behave" as if it were another type for a single use.

```
int h;
float f = 7.5;
h = (int) f;
println(h); // 7
println(f); // still 7.5
```

CSCI 182: Introductory Programming for Media Applications

Boolean Arithmetic and Conditionals
(Chapter 4.8 – 4.10, p. 65 – 78)

Boolean Arithmetic

- You can make arithmetic expressions with boolean values / variables, kinda like you do with numbers.
- A *boolean expression* uses special *boolean operators* instead of the normal arithmetic operators like + and −.
- A boolean expression evaluates to either true or false – not a number.
  - 5 > 4 is true

Boolean Operators

- *Relational operators* (more often called *comparisons*) are a kind of boolean operator that takes normal numbers (or numerical variables) as operands.
  - < less than
  - > greater than
  - == equal to (**Don't confuse with assignment** =)
  - <= Less than or equal to
  - >= Greater than or equal to
  - != Not equal to
- Examples:
  - 5 < 3 false
  - 4 >= 4 true
  - -14 == -13 false
  - 1 != 2 true
  - x > y depends on the values of x and y

or smileys =)

Boolean Operators

- *Logical operators* take two boolean values or boolean expressions as operands. They allow us to combine boolean expressions into larger ones.
  - || OR (true when at least 1 operand is true)
  - && AND (true when **both** operands are true)
  - ! NOT (unary, true when operand is false)

- A note for mathematicians:
  - Sorry, the implication operator $\rightarrow$ isn't in Java or Processing, but you can create it manually.
    - A $\rightarrow$ B is the same as !A || B or equivalently !(A && !B)
  - The bijection operator $\leftrightarrow$ is covered by ==
    - Or, if you feel pedantic, (!A || B) && (!B || A). Sorry, I digress!

Precedence Rules, again?!?

- Precedence rules apply. Here's a fuller list, highest precedence to lowest.
  - (Taken from http://en.wikipedia.org/wiki/Java_operators)

(	)			
++	--	!	Unary -	
*	/	%		
+	-			
<	<=	>	>=	
==	!=			
&&				
||				
=				

Boolean Expression Examples (answers on next slide)

- !(9 > 3)
- !(5 < 3 && !false) || 2 == 2
- (-47 <= -46 || 12 > 13) && !!!!!!(true || false)

```
int x = 5, y = 6, z = -7; // use below
```

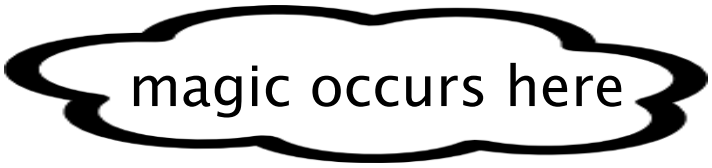
- (!(x+1 <= y) || (2*z*-1%5) > 3*x/y) && !!true

Boolean Expression Examples (with answers)

- !(9 > 3)
  - false
- !(5 < 3 && !false) || 2 == 2
  - !(false && true) || true
  - true
- (-47 <= -46 || 12 > 13) && !!!!!!(true || false)
  - (true || false) && true
  - true

```
int x = 5, y = 6, z = -7; // use below
```

- (!(x+1 <= y) || (2*z*-1%5) > 3*x/y) && !!true



  - true

Control Flow

- So far, we've written programs with statements executed one after the other, *sequentially*. The program flows from top to bottom.
  - It's called *Sequential Control Flow*.
- Sometimes, you may want your program to take different actions in different situations, based on some condition.
  - It's called *Conditional Control Flow*.
- A *conditional* is code that does this.
- The simplest conditional in Java / Processing is the *if statement*.

if Statement

- An *if statement* has the word "if", then a boolean expression in parens, then a block of code called the *body* in curly brackets.

```
if(boolean_expression) {
    // body
}
```

- First, the boolean expression is evaluated.
  - If it's true, the body is executed.
  - If it's false, the body is completely skipped.
- Example:

```
int x = 5;
if (x == 5) {
    line(10, 10, 70, 70); // it will be drawn!
}
```

if Statement

```
int x = 150;
if (x > 100) {
  ellipse(50, 50, 36, 36);
}
if (x < 100) {
  rect(35, 35, 30, 30);
}
line(20, 20, 80, 80);
```

if-else Statement

- An if statement can have an optional *else case* on the end. Together they're called an *if-else statement*.

```
if(boolean_expression) {
    // body executed when boolean true
} else {
    // body executed when boolean false
}
```

- First, the boolean expression is evaluated.
  - If it's true, only the first body is executed.
  - If it's false, the other body is executed **instead**.

if-else Statement

```
int x = 90;
if (x > 100) {
  ellipse(50, 50, 36, 36);
} else {
  rect(33, 33, 34, 34);
}
line(20, 20, 80, 80);
```

"And now for something completely different!"

- A *print statement* is used to output plain text to the console, like we did with "Hello world!"

```
print("Hello world!");
```

- You can print variable values, numbers, sentences, or just about anything!
- To print a sentence or some other string of letters, put it in double quotes.
- If you print multiple things, they appear right next to each other. Try this:

```
print("1");
print("2");
print("3");
```

- Use `println` instead of `print` and it'll put a newline on the end for you.

In-Class Lab 1

- Suppose I need help deciding whether or not to wear a coat.
- Declare a variable named `temperature` and initialize it to 20. Make an if-else statement:
  - If the temperature is less than 32 degrees, print that I should wear a heavy coat.
  - Otherwise, print that I shouldn't wear any coat, because it's too hot outside.
- Change the value of `temperature` to 70, to make sure the else case works too.

if-else Statement

- If you want to choose between more than 2 things, you can put additional cases in an if-else statement.

```
if(boolean_expression) {
    // body
} else if (boolean_expression) {
    // body
} else if (boolean_expression) {
    // body
} else {
    // body
}
```

- Each boolean expression from top to bottom is evaluated until we find the first one that's true. **Only that body** is executed.
- If none of the expressions were true, we do the else body by default. Note that the **else has no boolean expression.**
  - If there isn't an else case, do nothing! Same as the else body being empty.

if-else Statement

```
int x = 20;
if (x > 200) {
   fill(255, 0, 0);
} else if(x > 100) {
   fill(0, 255, 0);
} else if(x > 0) {
   fill(0, 0, 255);
} else {
   fill(0);
}
ellipse(50, 50, 80, 80);
```

Many if statements, or an if-else?

- So you might be wondering what the difference is between an if-else statement and many separate if statements.
- In an if-else statement, **only one body** is executed, no matter how many cases there are. It's about picking at most one out of many choices.
- It's possible for many if statements to all be true, some of them true, or none true.

Many if statements, or an if-else?

```
int x = 500; // We see one circle!
if(x > 0) {
   ellipse(20, 20, 10, 10);
} else if(x > 100) {
   ellipse(80, 20, 20, 20);
} else if(x > 200) {
   ellipse(80, 80, 30, 30);
} else if(x > 300) {
   ellipse(20, 80, 40, 40);
}
```

Many if statements, or an if-else?

```
int x = 500; // We see 4 circles!
if(x > 0) {
   ellipse(20, 20, 10, 10);
}
if(x > 100) {
   ellipse(80, 20, 20, 20);
}
if(x > 200) {
   ellipse(80, 80, 30, 30);
}
if(x > 300) {
   ellipse(20, 80, 40, 40);
}
```

Range of Values

```
int x = 9001;
if(x < 0) {
   println("x is negative.");
} else if(x >= 0 && x <= 9000) {
   println("x is between 0 and 9000.");
} else {
   println("It's over 9000!");
}
```

In-Class Lab 2

- Suppose I need help deciding what coat to wear.
- Declare a variable named `temperature` and initialize it to 20. Make an if-else statement with 4 cases:
  - 1) If the temperature is less than 32 degrees, print that I should wear a heavy coat.
  - 2) If the temperature is between 32 and 55 degrees, print that I should wear a jacket.
  - 3) If the temperature is exactly 56 degrees, then that must mean it's raining, obviously!? Print that I should wear a rain coat.
  - 4) Otherwise, print that I shouldn't wear any coat, because it's too hot outside.
- Run the program with several `temperature` values, to make sure every case works right!

Switch Statement

- A switch statement is a different type of conditional, similar to if-else.
- You execute one of several cases based on the value of a variable.

```
int x = 3;
switch(x) {
case 1:
   fill(0, 0, 255);
   break;
case 2:
   fill(0, 255, 0);
   break;
case 3:
   fill(255, 0, 0);
   break;
default: // default is optional, kinda like else is optional
   fill(0);
}
ellipse(50, 50, 50, 50);
```

If-else vs. Switch

- Switch statements look very different from if-else, but they're similar.
- We're still picking at most one case out of many.
- Switch statements are very limited, though. They compare the variable against **specific values** such as 1, 2, or 3.
  - It's like being restricted to == comparisons, and the others like < are off limits.
- You can use **any** boolean expression in if-else statements!
- But, what's with all the "break" statements…?

Fall Through and break;

- Switch statements have a weird property called *fall through*, where once we find the case to execute, we "fall through" and execute the ones below it, too!
- The break statement stops that weirdness from happening. `break;` instantly leaves the surrounding statement.
- So, once one body in the switch is executed, we "break out" of the switch statement and we're done.
  - It "breaks your fall"!

Fall Through and break;

```
int x = 1, y = 0;
switch(x) {
case 1:
   y++;
case 2:
   y++;
case 3:
   y++;
default:
   y++;
}
```

CSCI 182: Introductory Programming for Media Applications

Loops

(Chapter 8.1, p. 164–174)

Motivating Example

- Let's say I want to make a row of 10 evenly spaced dots:

```
strokeWeight(4);
point( (width/11)*1, height/2);
point( (width/11)*2, height/2);
point( (width/11)*3, height/2);
point( (width/11)*4, height/2);
point( (width/11)*5, height/2);
point( (width/11)*6, height/2);
point( (width/11)*7, height/2);
point( (width/11)*8, height/2);
point( (width/11)*9, height/2);
point( (width/11)*10, height/2);
```

- Each dot is made in almost the same way – a clear pattern!

Iteration

- Whenever you want the computer to do a task (such as drawing an evenly-spaced dot) many, many times, it's better to use a *loop*.
- A loop is code that repeats itself (*iterates*) many times over and over, while a boolean expression (a.k.a. *conditional*) stays true.
  - It's called *iterative control flow*.
- There are 3 slightly different kinds of loops we'll talk about today.
  - for loop
  - while loop
  - do loop (a.k.a. do-while loop)

while loop

```
while(<conditional>) {
    <body>
}
```

- A *while loop* looks a lot like an if statement, but there's one important difference!
- 1) First, the boolean expression is evaluated.
  - 2) If it's false, skip the body.
  - 3) If it's true, execute the body, **then we go back to step 1 and repeat!**
- Each time through is called an *iteration*.

while loop

- The idea is to control the number of iterations, to accomplish something useful.
  - So, if we drew 1 dot each iteration, then after 10 iterations, we'd have a row of dots like before!
- The key is to **make progress every iteration**, bringing the boolean expression "closer" to becoming false!
- Once the boolean expression becomes false, we stop iterating.

Motivated Example

```
strokeWeight(4);
int i = 1; // initialize
while(i < 11) { // conditional
   point( (width/11)*i, height/2 );
   i++; // update
}
```

- This example shows the three super important steps to making a good loop:
  - **1)** Declare & **initialize** a *loop variable* for controlling the loop. Picking the initial value is crucial. It's like a "starting point."
  - **2) Update** your loop variable each iteration, so you're "making progress."
  - **3)** Write a **conditional** designed to be true several times, then eventually become false. This is your end "goal."

Today's most important slide!

In-Class Lab 1: while loop

- Make a diagonal line one point at a time, like this:
  - Declare two int variables named x and y.
  - Initialize them both to 0.
  - Use a while loop, and in each iteration:
    - Draw a point at (x,y).
    - Increment both x and y.
  - Your while loop should continue until either x reaches the right edge of the display window, or y reaches the bottom.
  - Make sure your program works regardless of display window size.

*(another half-truth we'll bust later)

for loop

```
for(<init>; <conditional>; <update>) {
    <body>
}
```

- Perhaps the most popular kind of loop is the *for loop*.
- They're essentially identical* to while loops, but the syntax puts the three important parts all on the top line.
  - initialization
  - conditional
  - update

Okay, maybe this slide is the most important.

Examples

- Print a column of numbers:

```
int j;
for(j = 0; j < 5; j++) { // 0 thru 4
   println(j);
}
```

- Draw a "bullseye" of concentric circles:

```
size(500,500);
for(int d=width; d>0; d-=30) {
   ellipse(width/2, height/2, d, d);
}
```

Examples

- Make a pattern of shapes:

```
int i, spacing;
spacing = width/5;
for (i = 1; i < 5; i++) {
  ellipse(spacing*i, spacing*i,
          spacing, spacing);
}
```

Conversion: while loop ↔ for loop

- The while loop version:

```
<init>
while(<conditional>) {
    <body>
    <update>
}
```

- The for loop version:

```
for(<init>; <conditional>; <update>) {
    <body>
}
```

Conversion Example

- The while loop version:

```
int i = 1;
while(i < 11) {
    point( (width/11)*i, height/2 );
    i++;
}
```

- The for loop version:

```
int i; //Pro tip: declare variables first
for(i = 1; i < 11; i++) {
    point( (width/11)*i, height/2 );
}
```

do–while loop

```
do {
    <body>
} while (<conditional>);
```

- The elusive *do–while loop*, also called a *do loop*, is similar to a while loop, but the conditional is checked **at the end** of each iteration, instead of at the beginning.
- This means it **will** iterate once, even if the conditional is never true!
  - If the conditional of a while loop or a for loop starts out false, it won't even iterate once.
- Useful when you want to do something at least once, and possibly more.

do-while loop

```
int j = 1;
do {
    print("buffalo ");
    j++;
} while(j <= 8);
```

http://en.wikipedia.org/wiki/Buffalo_buffalo_Buffalo_buffalo_buffalo_buffalo_Buffalo_buffalo

Analyzing Loops

- By looking at a loop, you can calculate how many iterations it will go through, and what will happen in each iteration.

```
int j;
for(j = 0; j < 5; j++) {
   println(j);
}
```

- Ask yourself:
  - What is the loop variable? What is its initial value?
  - Is the conditional satisfied initially?
  - What happens during that first iteration?
  - What **changes** affect the next conditional evaluation?
  - What's the **last** value of the loop variable that satisfies the conditional? What happens during that final iteration?
  - What loop variable value made the conditional turn false?

In-Class Lab 2: for loop

- Set the size to 600 x 600.
- Write a for loop with variable y that makes a column of 10 diamonds, 60 pixels apart. The first is centered at (100, 30).
- Each diamond is 10 pixels wide by 10 pixels tall. Hints:
  - Your loop should have 10 iterations.
  - The next diamond's center is 60 pixels below the previous one.
  - Instead of diamonds, try having each iteration draw just 1 point.
  - Modify it to draw a quad instead, centered where the point was.
    - Each corner goes in a cardinal direction: up, right, down, left.
    - Each coordinate pair of your quad is one of these four corners.
    - Imagine you are standing at location (100, y) in the middle of a diamond. How would you modify y to get from the middle to the top of the 10-pixel-tall diamond? Write it as an expression using y. Your x coordinate will still be 100. This is your first coordinate pair!
    - Then, write similar expressions for the other 3 corners.

Nested Loops

```
int i, j;
for(i = 0; i < 100; i++) {
    for(j = 0; j < 100; j++) {
        point(i, j);
    }
}
```

- You can *nest* a loop inside another loop.
  - Also works with if statements, switch statements, and more. You can even nest multiple levels deep.
- We call the top loop the *outer loop*. The loop inside it is the *inner loop.*
- When nesting loops, use different loop variables.

Nested Loops

```
int i, j;
for(i = 0; i < 100; i++) {
    for(j = 0; j < 100; j++) {
        point(i, j);
    }
}
```

- Each iteration of the outer loop of i, we execute the **entire** inner loop of j, doing **all** iterations until the inner loop conditional turns false.
- Then, we do all of that again in the outer loop's next iteration!
- The number of inner loop iterations gets **multiplied**.
  - So above, 100 * 100 means 10,000 points are drawn! Each gets a different (i, j) location in the display window.

Nested Loops

```
int i, j;
strokeWeight(5);
for(i = 0; i < 100; i+=10) {
   for(j = 0; j < 100; j+=10) {
      point(i, j);
   }
}
```

- You don't have to use ++ for the update statements. Try instead using +=10. You'll see an evenly spaced grid!
- What if I drew something cooler than just a point, each iteration?

In-Class Lab 3: nested loops

- Starting with your lab 2 code, make a 10-by-10 grid of 100 diamonds. Each one must be 10 pixels wide and 10 pixels tall. The first is centered at (30, 30). Each column is 60 pixels from the next column.
- Your lab 2 loop makes 1 column of diamonds. Nest that loop inside **another** loop with a different variable x, with 10 iterations.
- You'll need to rewrite the x coordinates of the quad a little bit, so they are expressions involving x instead of being near coordinate 100.
- Hints:
  - Imagine you are standing at location (x, y) in the middle of a diamond. How would you modify x and y to get from the middle to the top of the 10-pixel-tall diamond? Write an expression using x for the x coordinate, and write your expression from lab 2 using y for the y coordinate. This is your first coordinate pair!
  - Write 3 more pairs of expressions to get you from the middle to each of the other corners. That's your quad!