CSCI 182: Introductory Programming for Media Applications

Introduction

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

Course Webpage

- Moodle: http://learnonline.unca.edu/
- Use your UNC Asheville username & password.
- Our class page is
  - "Computer Science 182.002/003: Introductory Programming for Media Applications: Reagan"
- Syllabus link at the top.
  - Course policies
  - Contact information
  - Textbook, etc

When the PowerPoint go from "Syllabus" to "Lecture 1"

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.

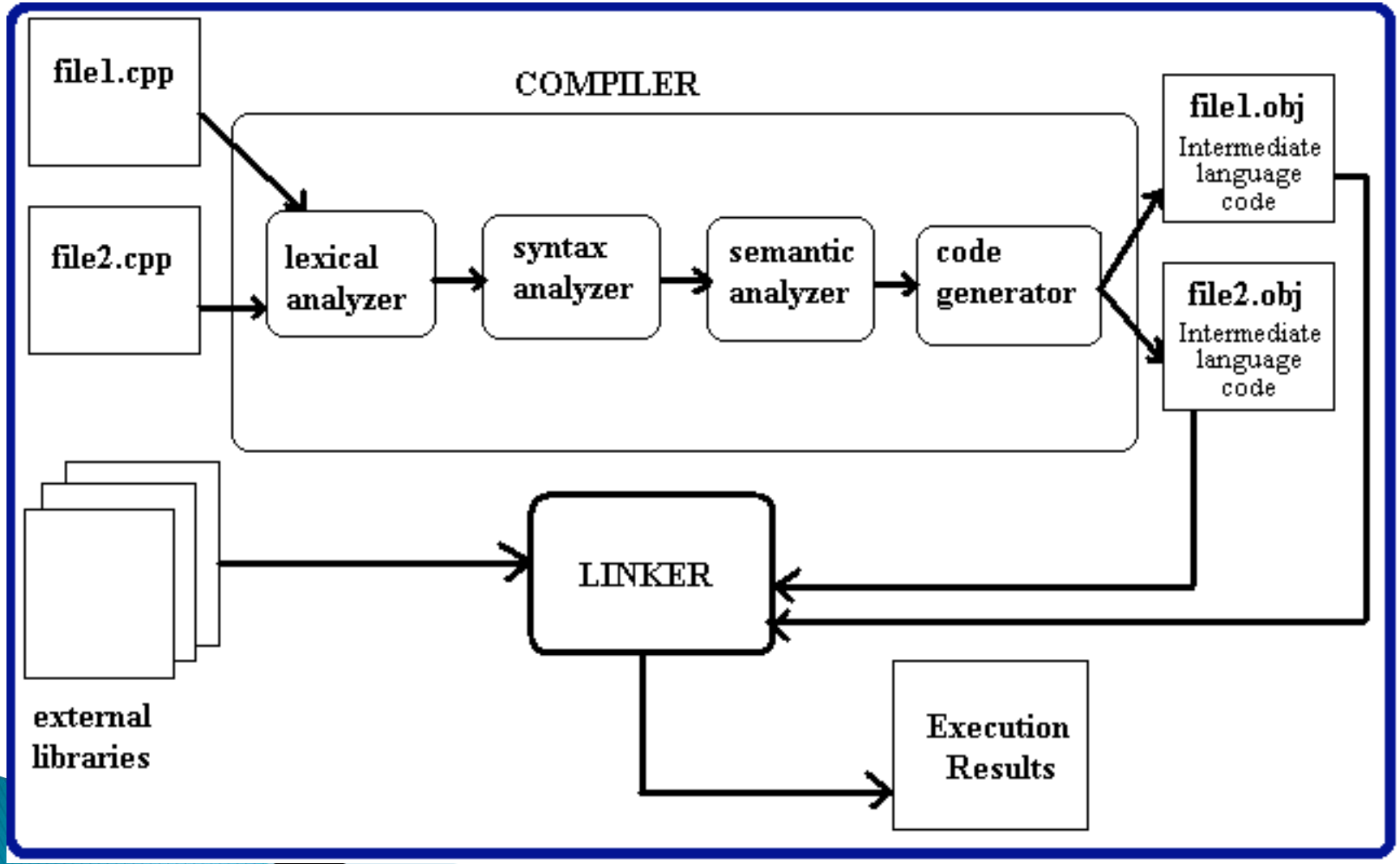
Computer Programming

- 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…stuff?

- 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

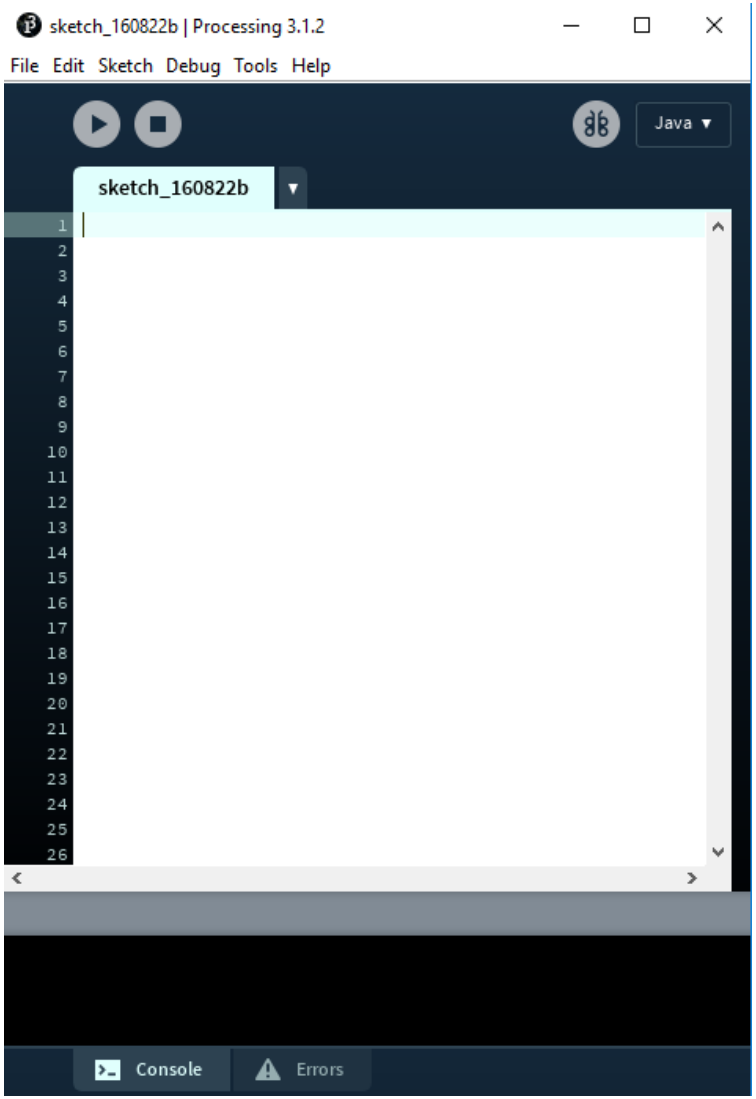
Menu & Toolbar

Tabs

Text editor

Messages & errors

Console for text output



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 the 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.

Hello, World!

- 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.* This is 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:

  Try typing: `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.

1. One-line comments: after `//`

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

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

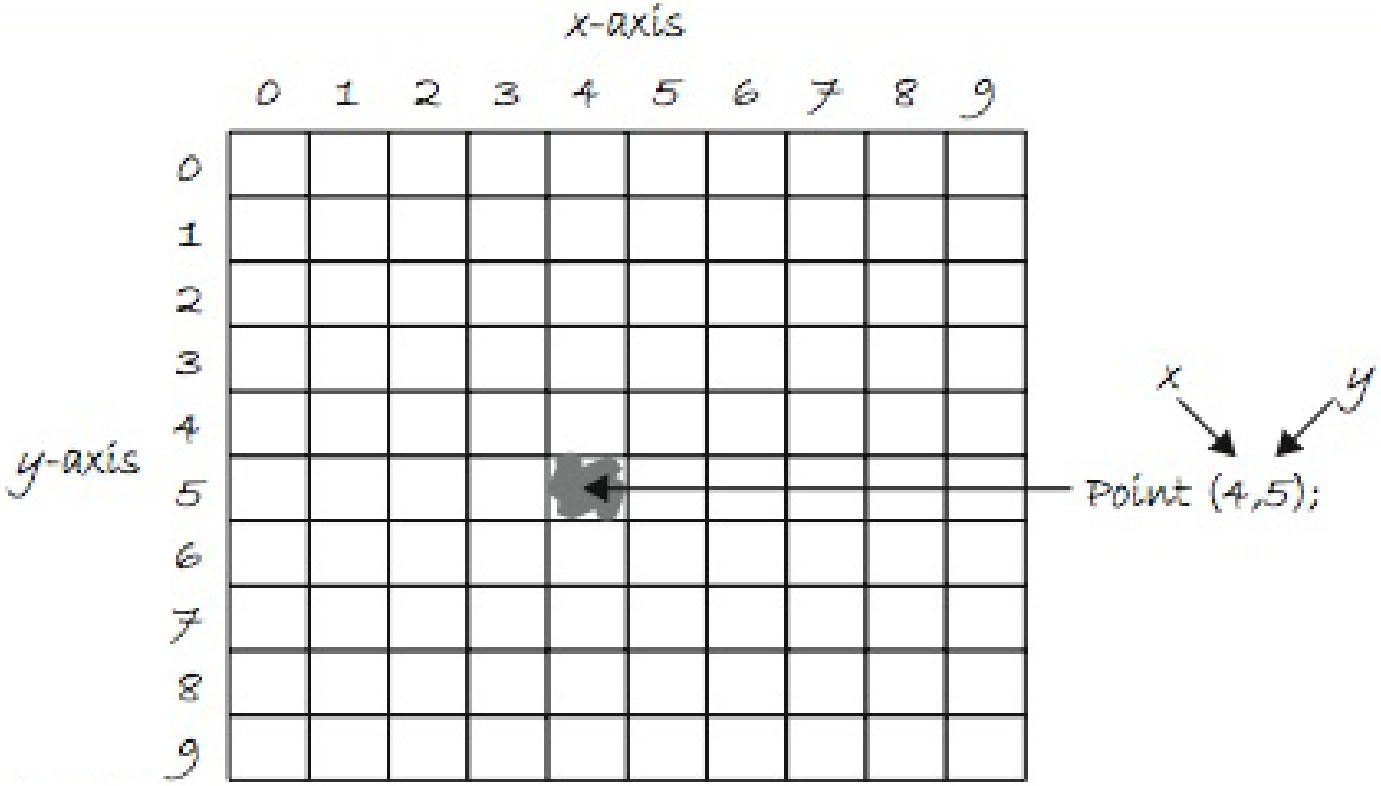
```
/* This is a larger comment
stretching over two lines. */
```

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 X 100 is the default size. Change the size of the window with the `size` function.

```
size(400, 300); // 400 X 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) coordinate (40,60)
```

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

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

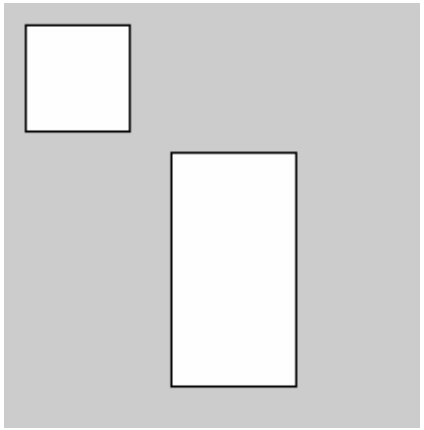
- That line connects the first location (40, 60) to the other location (80, 20).

In-Class Lab 1

Rectangles

- You can draw rectangles and squares with the `rect` function.
- Provide 4 numbers:
  - (x,y) coordinate 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.

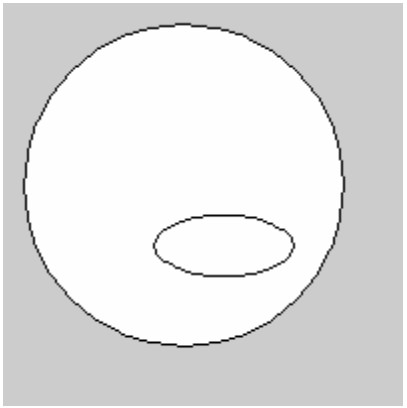
Rectangles

- 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

- You can 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`.

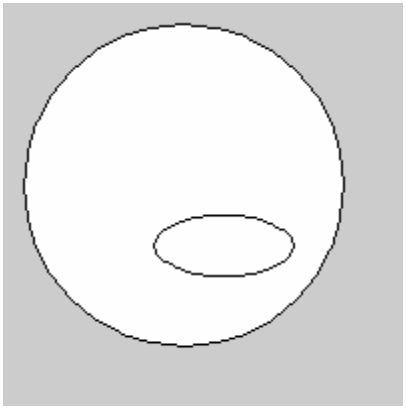
```
ellipse(90, 90, 160, 160);
ellipse(110, 120, 70, 30);
```



Order Matters!

- 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.

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



Break time!

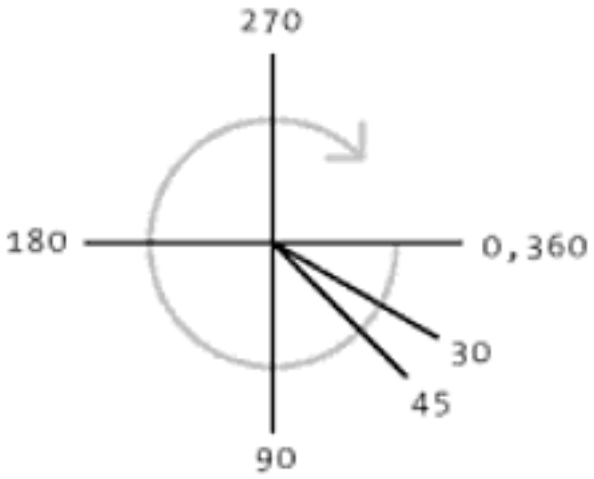
Arcs

- You can draw a part of an ellipse or circle (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*)

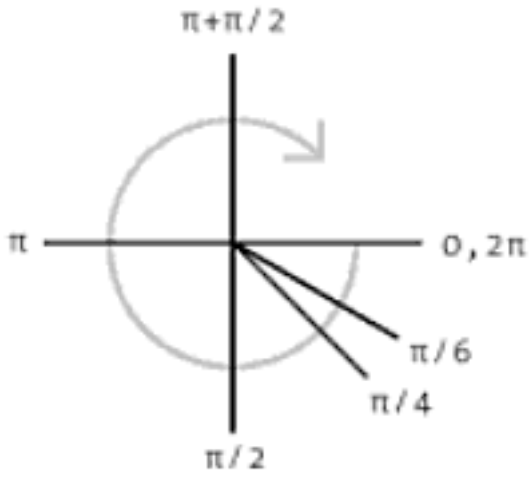
```
// Top half of a circle
arc(90, 90, 160, 160, PI, 2*PI);
// PI is a built-in number, ~3.14159
```

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 (Bez-E-ay)

- We can draw curved lines called *bezier curves* using the `bezier` function.
- 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.
- Visualize: 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.
- Defines 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 color, 1 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 until you specify otherwise.

In-Class Lab 2

Color Models

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

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

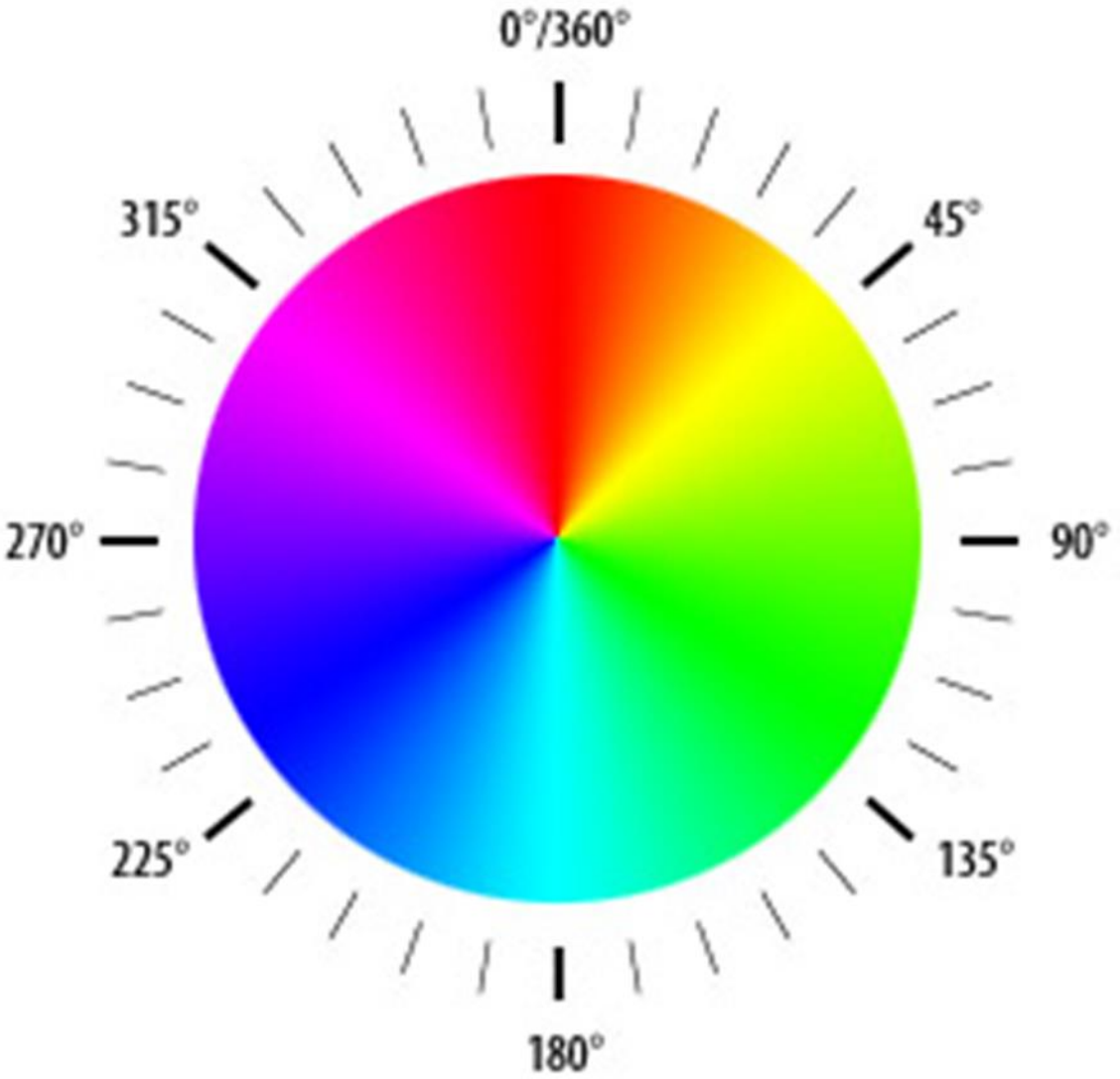
- Now, each red, green, and blue component is a number from 0 to a million!
  - This allows for much more fine-grained colors! (Your video card and monitor have limits, though.)

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



In-Class Lab 3

CSCI 182: Introductory Programming for Media Applications

Documentation & Style

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.
  1. One-line comments: after `//`
  2. Multi-line comments: between `/*` and `*/`

Comment Examples

1. One-line comments:

```
// This is a one-line comment.
/* This is a one-line comment. */
```

2. In-line comments:

```
noStroke(); // Turn off the outline for this shape.
noStroke(); /* Turn off the outline for this shape. */
```

Comment Examples

3. Multi-line comments:

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

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

// This is also an acceptable
// way to comment your code
// over multiple lines.
```

Program Header

- Every program you write should have a header that documents that author, purpose, etc. of the source code that follows:

```
//*************************************
// Program Name:
// Author:
// Date:
// Version:
// Purpose:
// Comments:
//*************************************
```

Source Code

- Code should be left-aligned.
- One functional “piece” of code per line.
- Use additional whitespace to make things more readable.
- Variables and functions should start with a lowercase letter.
- Multi-word variables and functions should begin lowercase and switch to Proper Case for each consecutive word.

`variableName = 3;` OR `noFill();`

Christmas_Tree | Processing 3.1.2

File Edit Sketch Debug Tools Help

Christmas_Tree

```
//*****************************************************************************
// Program Name: Christmas_Tree
// Author: Adam Reagan
// Date: 08/23/2016
// Version: 0.1
// Purpose: To draw a Christmas tree using the Processing programming language
// Comments:
//*****************************************************************************

//Set size of canvas
size(400, 600);

//Set canvas color to white
background(255);

//Set all drawings to smooth edges and no outline
smooth();
noStroke();

//Draw nighttime sky and fill it to be dark blue
fill(25, 25, 92);
rect(0, 0, 500, 450);

//Draw Christmas Tree
fill(0, 205, 50);                       //Green tree top
triangle(50, 500, 350, 500, 200, 100);  //Green tree top
fill(137, 100, 90);      //Brown stump
rect(175, 500, 50, 60);  //Brown stump

//Draw Star
fill(255, 236, 23);
triangle(200, 150, 175, 100, 225, 100); //Upside down triangle
triangle(175, 125, 225, 125, 200, 75);  //Upright triangle

//Draw Ornaments
fill(225, 102, 60);
ellipse(160, 300, 10, 10);
```

Done saving.

Console Errors

CSCI 182: Introductory Programming for Media Applications

Variables

(Chapter 3, p. 23 – 42)

Computer Memory

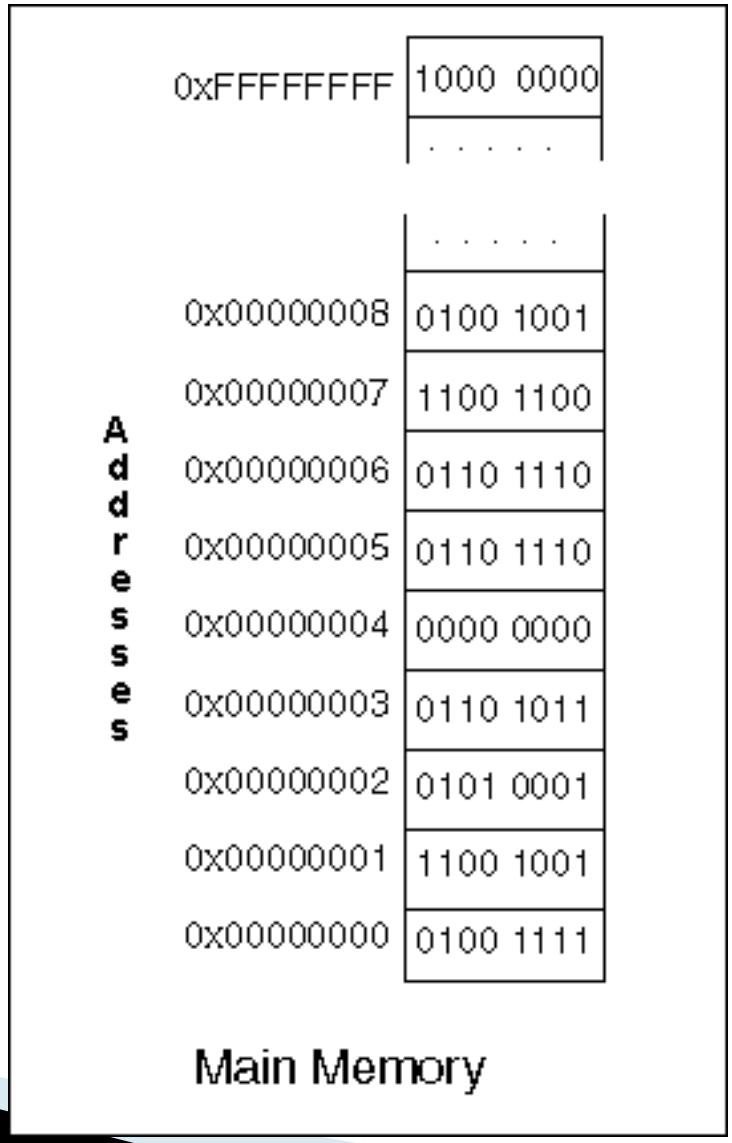
- All data and information a program uses is stored in the computer's *memory*.
- Every location in memory has a unique number called its *address*.
  - Think of memory like a line of numbered PO boxes at the post office.

Computer Memory

Computer Memory

- Each memory location can store information.
- A ***variable*** is a name given to a memory address/location. We create and use them to store, retrieve, and modify information in memory.

Computer 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

Primitive Types

- **`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;
```

Using Variables

- 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**.
  - Math Equation:

```
3 + 4 = 7
5.2 - 2.1 = 3.1
```

  - Assignment Statement:

```
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;
```

Using Variables

- 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`

Variable Names

- 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

  - Note: 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 *constant values* 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

- So… 1 / 4 is **0**
  - 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
```

Arithmetic

- 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.

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, kind of 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 and NOT a number.
  - Example: 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

Boolean Operators

- Examples:
  - 5 < 3 false
  - 4 >= 4 true
  - -14 == -13 false
  - 1 != 2 true
  - x > y depends… on the values of x and y

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)

Boolean Operators (extra!)

- 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 more complete list, highest precedence to lowest.
  - (Taken from http://en.wikipedia.org/wiki/Java_operators)

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

Boolean Expression Examples

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

For: `int x = 5, y = 6, z = -7;`

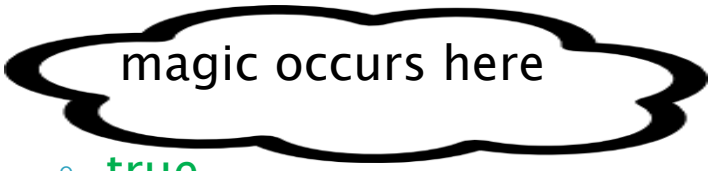
- (!(x+1 <= y) || (2*z*-1%5) > 3*x/y) && !!true

Boolean Expression Examples

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

For: `int x = 5, y = 6, z = -7;`

- (!(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, and 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);
```

In-Class Lab 1

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

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 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++;
}
```

In-Class Lab 3

CSCI 182: Introductory Programming for Media Applications

Loops

(Chapter 8.1, p. 164-174)

A 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!

Say “Hello!” to Loops!

- 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.
  - This is 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.

A Motivated Example (Redux)

```
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

`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

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

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 );
}
```

Another Conversion Example

```
int i;
size(500, 500);
colorMode(RGB, max);
i = 0;
while(i <= max) {
  stroke(i, 0, 0);
  line(i, 0, i, height);
  i++;
}
```

```
int i;
size(500, 500);
colorMode(RGB, max);
for(i = 0; i <= max; i++) {
  stroke(i, 0, 0);
  line(i, 0, i, height);
}
```

What's one of the more notable differences?

In-Class Lab 2: `for` loop

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?

Break Time

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.
  - This 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);
   }
}
```

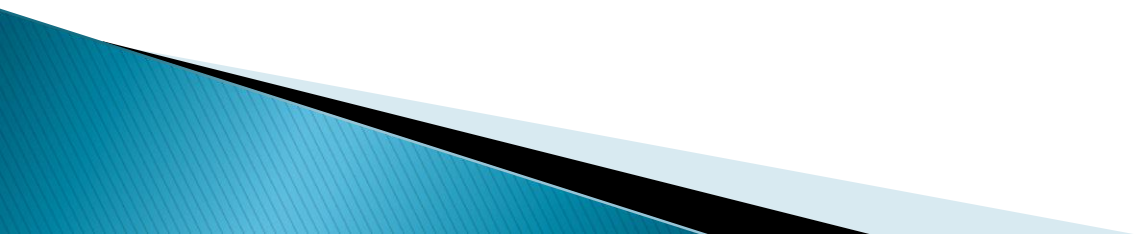
- On each iteration of the outer loop of `i`, we execute the **entire** inner loop of `j` until the inner loop conditional turns false.
- Then, we do all of that again in the outer loop's next iteration!
- The number of inner loop iterations gets **multiplied** by the number of outer loop iterations.
  - So in the example above, 100 * 100 = **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);
  }
}
```

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

In-Class Lab 3: Nested loops



CSCI 182: Introductory Programming for Media Applications

Random Numbers

(Chapter 11.1, p. 224-231)

Random Numbers

- Computers have a hard time creating truly random numbers.
- But, you can create *pseudorandom* numbers easily. They're "random enough" for most purposes.
- Create a pseudorandom floating point number **between 0 and some upper bound** by putting the bound after the word `random`, in parens:

```
float f = random(5); // from 0 to 5
float g = random(42); // from 0 to 42
```

Random Numbers

- You can specify a lower bound other than 0, if you want:

```
println(random(15, 20));
```

- That printed a floating point number between 15.0 and 20.0
- The upper bound is **exclusive.**
- Try putting that in a loop with 10000 iterations. You'll never see exactly 20.0
  - There are a **huge** amount of numbers between 15.0 and 20.0 – the chance of getting any particular number such as 18.3534 is almost none, and you will **never** get 20.0

Quick reminder about types…

- Since `random` gives you a floating point number, you can store it into a float or a double, but **not** a byte/short/int/long.

```
int i = random(6, 7); // ☹
float j = random(6, 7); // ☺
double k = random(6, 7); // ☺
int m = (int) random(6, 7); // O.o
```

- Try that last one. Use a loop to print out 100 of `(int) random(6, 7)`. What do you see?
  - What does a floating point number between 6 and 7 look like? 6.#######, right? So, what is that as an int?

Coin Flipping & Dice Rolling

- We can flip coins and roll dice by combining
  - random numbers
  - if statements
  - type casting
- To flip a coin, make an if statement with a 50% chance of evaluating to true:

```
if(  ((int)random(0, 2)) == 0   )
```

- To roll a d6 (a six sided die), assign a random int value between 1 and 6 into a variable:

```
int dieRoll = (int) random(1, 7);
//Why the upper bound of 7? Why not 6?
```

Using Random Numbers

- We can use a coin flip to decide what to do in our program.

```
if( (int) random(0, 2) == 0) {
    rect(10, 10, 80, 80);
} else {
    ellipse(50, 50, 80, 80);
}
```

- We can also make random colors, locations, or sizes. Anywhere you use a number, try using a random number instead!

```
fill(random(255), random(255),
                  random(255));//random color
ellipse(50, 50, random(10, 90),
                random(10, 90));//random size
```

Control the Randomness

- You can use random numbers to achieve subtle effects, by carefully controlling and restricting the range.
- This ellipse is always close to the center of the screen, even though its position is random:

```
ellipse(width/2  + random(-10, 10),
        height/2 + random(-10, 10), 70, 70);
```

- This color is a random shade somewhere between blue and green, but never anything wildly different:

```
fill(random(75), random(100, 255),
                 random(100, 255));
```

Control the Randomness

- This Bezier curve always smiles:

```
noFill();
strokeWeight(4);
bezier(10+random(-5, 5),   50+random(-5, 5),
       30+random(-15, 15), 90+random(-15, 15),
       70+random(-15, 15), 90+random(-15, 15),
       90+random(-5, 5),   50+random(-5, 5));
```

Let's Look at Some Examples!

In-Class Lab Time!

CSCI 182: Introductory Programming for Media Applications

Number Bases:
Binary, Octal, Decimal, and Hex

Number Bases

- Most math we encounter is done using base 10, also called *decimal* numbers.
- There are 10 different symbols (10 values) a single numeric digit can have: 0 through 9.
  - For any number over 9, we need more digits.

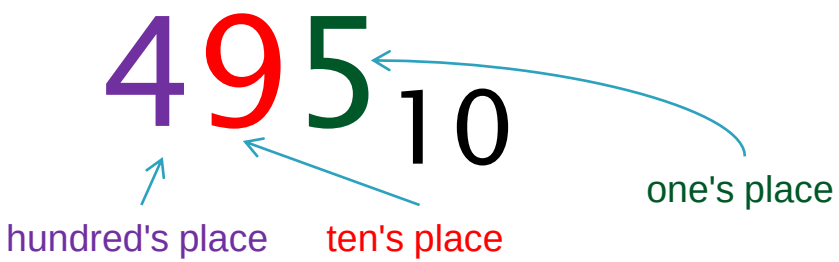
Number Bases

- Other *bases* (or *radixes)* often used in computer science include:
  - **Base 2, called *binary***. A digit's value is 0 or 1. We often call binary digits *bits*. Machine language is written in binary.
  - **Base 8, called *octal***. Digit values range from 0 to 7.
  - **Base 16, called *hexadecimal* or simply *hex***. Each digit can have sixteen different values, including:

    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f
- When writing a number, we sometimes put the base as a subscript after the number. (If no subscript, assume base 10.)
  - 17_8 is written in base 8, with the same value as 15_{10}

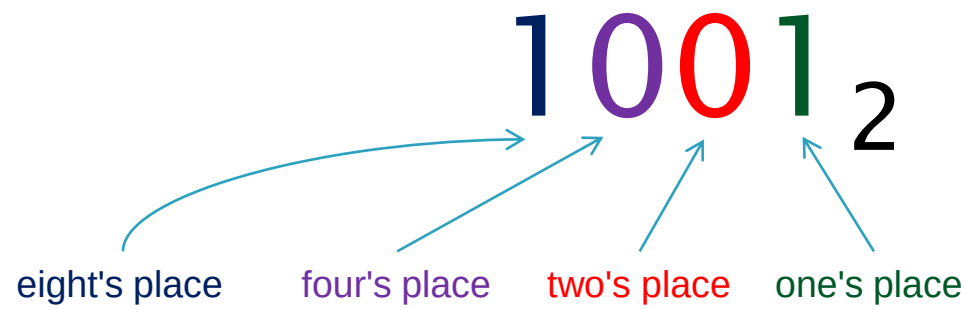
Digits

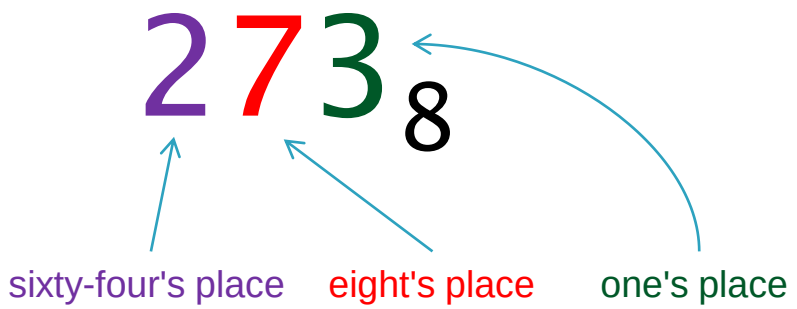
- When we look at a very large number in base 10, we're taught to look at each digit's value.
  - The rightmost digit is the "one's place", then there's the "ten's place", etc.
  - Base 10 digits are **labeled** with the powers of 10. Example:



Digits

- When looking at a number in another base, you can label the digits the same way: with the powers of that base.
- Examples:

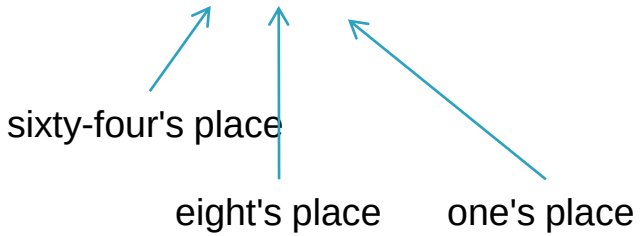




Converting To Base 10

- To convert a number to base 10:
  - 1) Look at each digit by itself. Take the digit's value and multiply it by that digit's "label".
  - 2) Sum up the results from each digit.
- Example:

$$273_8 = 2 * 64 + 7 * 8 + 3 * 1 = 187$$



Converting To Base 10

- To convert a number to base 10:
  - 1) Look at each digit by itself. Take the digit's value and multiply it by that digit's "label".
  - 2) Sum up the results from each digit.
- Example:

$$273_8 = 2 * 64 + 7 * 8 + 3 * 1 = 187$$

sixty-four eight one

Converting From Base 10

- Any number in base 10 can be converted into any other base.
- To convert a number n_{10} to some base x:
  - 1) Figure out the highest order digit of base x you'll need.
    - **Pro tip**: It's the digit one order below the smallest power of x that's > n.
    - **Another way**: Integer divide your number n by larger and larger powers of x, until you get an answer < x.
  - 2) Integer divide n by the highest order digit's label. The result goes into that digit.
  - 3) Take n % label. Repeat step 2 with that value, to get the next digit. Keep going until the "one's place."

Converting From Base 10: Example

- Let's convert 461_{10} into base 8.
- The highest order digit we'll need is the "64's place."
  - Why? Because the next higher digit (the "512's place") is larger than 461.
- 461 / 64 evaluates to 7 with a remainder of 13.
  - So, we must write a 7 in the "64's place", and use the remainder 13 to figure out the "8's place" in the same way.
- 13 / 8 evaluates to 1 with a remainder of 5.
  - So, we must write a 1 in the "8's place", and use the remainder 5 for the "1's place."

$$461_{10} = 715_8$$

Changing Bases

- So, we can convert any integer from any base x to any other base y.
  - 1) Convert from base x to base 10.
  - 2) Convert from base 10 to base y.
- Floating point numbers can be converted, too, but it's slightly more complicated.

Hexadecimal

- Base 16, called hex, has **more** values in a single digit than base 10.
  - Any number from 0… 9 in hex is the same in base 10.
  - $a_{16} = 10_{10}$
  - $b_{16} = 11_{10}$
  - $c_{16} = 12_{10}$
  - $d_{16} = 13_{10}$
  - $e_{16} = 14_{10}$
  - $f_{16} = 15_{10}$

Binary is extremely useful!

- Binary is **very** convenient for building computers. We can store a bit (binary digit) in a small circuit:
  - Store an electric charge in the circuit, which means 1_2
  - Leave the circuit uncharged, which means 0_2
- The problem with binary is it takes a **lot** of bits to store big numbers.
  - To store the number 1452795_{10}, you need 21 bits!
  - In case you're curious, that's 101100010101011111011_2
  - For all you math folks, it takes $\lceil \log_2 n \rceil$ bits to store any number n.

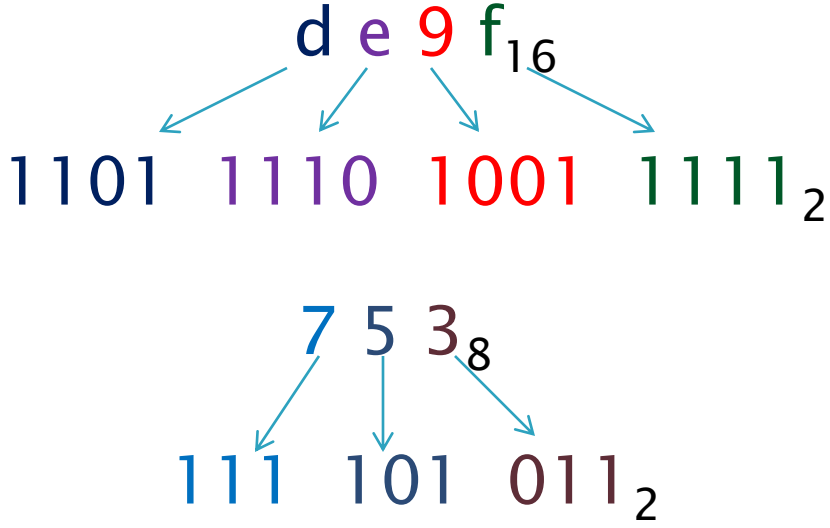
Hex is extremely useful!

- Hex is very convenient for writing and discussing binary in a more **condensed** form.
- It takes only 1 hex digit to store 4 bits.
  (Any # from 0 to 15)
  - Examples: $1001_2 = 9_{10} = 9_{16}$. $1111_2 = 15_{10} = f_{16}$.
- A *byte* takes 2 hex digits.
  (8 bits, any # from 0 to 255)
  - For all you math folks, it takes $\lceil \log_{16} n \rceil$ hex digits to store any number n. That's a lot less than binary, which requires $\lceil \log_2 n \rceil$ digits.
- Trivia:
  A hex digit used to be called a *hexit* or *nybble*.

Hex ↔ Binary ↔ Octal, the easy way

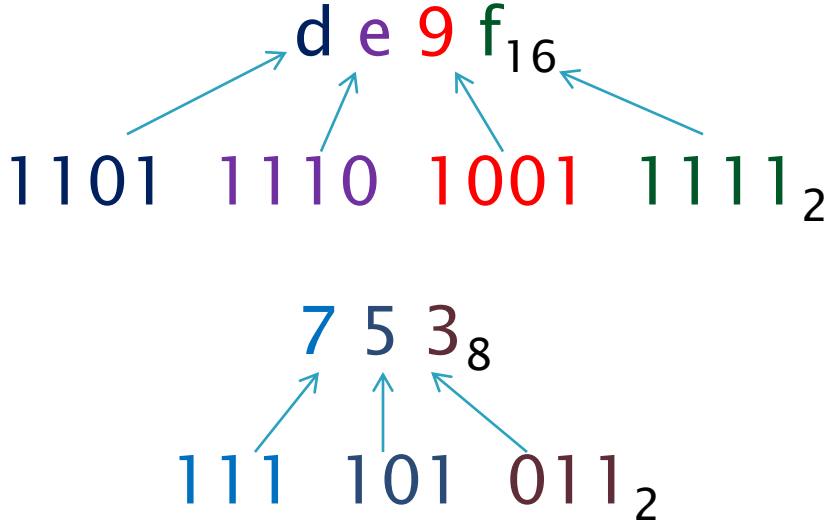
- Here's an easy trick you can use to convert between "power of 2" bases, such as binary, octal, and hex.

- Separately convert each digit into a group of bits. Examples:

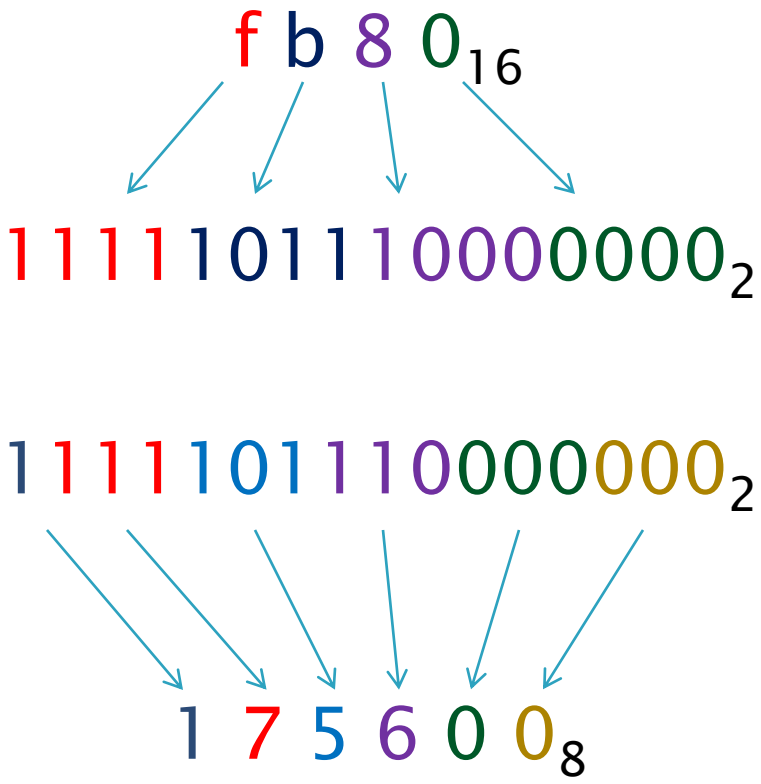


Hex ↔ Binary ↔ Octal, the easy way

- It's just as easy to convert the other way, by grouping together the bits into clumps of 3 or 4.
- Turn each clump of bits into one hex or octal digit. Examples:



Hex ↔ Binary ↔ Octal, the easy way



Number Bases in Processing & Java

- You can use numbers in binary, octal, decimal, or hex anywhere in your code!
- To write a hex number, prefix it with `0x`
  - So, this if statement is true: `if(0xf == 15)`
- To write a binary number, prefix it with `0b`
  - **Note:** This only works in Java 7 or newer.
- To write an octal number, prefix it with `0`

Number Bases in Processing & Java

- Decimal numbers **must not** start with `0`
  - The exception is the base 10 number `0` itself, which you can safely use. The number `0` is the same in every base, anyway.
  - Base 10 floating point numbers beginning with `0.` are also safe.
- You can also define an RGB color in Processing's default color mode using 6 hex digits, like this:

```
background(#ff00ff); // bright purple
```

  - That's $ff_{16} = 255$ for the red component, no green, and $ff_{16} = 255$ for the blue component.

In-Class La… I mean, examples!

- Let's practice working with number bases, **without any calculator**.
  - Just paper, pencil, and brain power!

- 1) Convert 29_{10} to octal.
- 2) Convert $1c_{16}$ to binary.
- 3) Convert 11001001_2 to decimal.
  - Brownie points if you get the nerd reference!
  - If not, just Google the number 11001001

CSCI 182: Introductory Programming for Media Applications

Animation

(Chapter 4.2, 4.7, 4.11)

Control Flow

- Up to this point, all our programs' control flow can be called *procedural*. It executes once (almost instantaneously), produces a result, and its done.
- By contrast, some programs run continuously – the program stays "running" for a long time, usually waiting for something to happen. This is called *event driven control flow*.
  - Animations and interactive programs are in this category.
  - Today, we'll discuss animations. Later, we'll look at mouse & keyboard interaction.

setup() and draw()

- To make an event driven program in Processing, you need to split your code into two parts:

```
void setup() {
  // Optional. Code in here gets executed
  // procedurally: once at the beginning
  // of the program.
}

void draw() {
  // Code in here gets executed
  // continuously: over and over,
  // 60 times every second!
}
```

`setup()` and `draw()`

- Any video or animation consists of still images called *frames* seen in rapid succession.
- The number of frames per second is called the *fps* or *framerate*. In Processing, the default framerate is 60.
- The code inside `draw` draws one frame.

Example

- Here's a simple event driven program:

```
void setup() {
  frameRate(4); // set framerate to 4
  // What'll happen if we comment out the above?
}
void draw() {
  println(frameCount); // built-in variable
}
```

- The above example has a framerate of 4, not 60, so `draw` is only executed 4 times per second.
- The `frameCount` built-in variable automatically keeps track of how many frames were drawn so far. It increments itself automatically each time we run `draw` to draw a frame.
- Note: To stop running an event driven program in Processing, click the square "stop" button next to the triangle "run" button.

Slightly More Interesting Example

```
float y = 0;
void setup() {
  size(200, 200);
  frameRate(10);
}

void draw() {
  line(0, y, width, y); // horizontal
  y++; // y gets bigger every frame
}
```

- You can declare variables outside setup and draw, but otherwise **all your code must go inside** setup and draw!

Slightly More Interesting Example

- It's drawing a line every frame, right below the last one. The background does not refresh by itself.
- Try putting this on the line below draw:

```
background(204);
```

- Now, a fresh background will **cover up** everything drawn earlier.
  - All we'll see is what's drawn this frame: a line.
  - Since the line is drawn in a different place every frame, it appears to move!
- We can put the framerate back to the default 60, giving us smoother motion by removing the

```
frameRate(10);
```

Variable Scope

- A variable declared outside of setup and draw (like, at the top of the code) has *global scope*.

```
int i; // i is a global variable
void setup() {
  i = 5; // starting value of i
}
void draw() {
  i++; // incrementing the same variable i
}
```

- A global variable can be used **anywhere** in your program:
  - Inside setup (say, to initialize it)
  - Inside draw (say, to change it every frame)
  - Anywhere!

Variable Scope

- Variables declared inside setup or inside draw have *local scope*, which means it **only exists inside there!**

```
void setup() {
  int j = 5; // Local to setup's scope
             // (in the curly brackets)
}
void draw() {
  int j = 7; // Completely different
             // variable, in draw's scope
}
```

Let's See Those Examples Again…

CSCI 182: Introductory Programming for Media Applications

Functions

(Chapter 4.2, 5.1)

Functions a.k.a. Methods

- A *function* or *method* is a name for a *block* of code.
  - A block of code is just some lines of code surrounded by { }, like the body of a loop or an if.
- Why use functions?
  - 1) Break a complex program into many simple pieces.
  - 2) Code reuse. We can execute the same block of code whenever we want, by simply saying its name.

Invoking a Function

- You *call* or *invoke* an existing function in order to use it: to execute its block of code.
- Some functions have *parameters*: variables that hold information that affect what happens.
- When invoking a function with parameters, you must provide an *argument* (value) for each parameter.
  - These values affect the result.

Invoking a Function

- To invoke a function, you need:
  - 1) The function name
  - 2) A list of arguments to each parameter, separated by commas and surrounded by a pair of parens.
  - 3) A semicolon at the end.

```
line(10, 10, 90, 90);
```

Name

Argument list

Yay for semicolons, amirite!?!

Example

- We've already been invoking *built-in functions* all semester.
  - `size, background, fill, stroke, noStroke, rect, ellipse, arc, quad,` and all the rest!
    - (FYI, loops and conditionals are not functions.)

```
ellipse(70, 40, 50, 30);
```

- The ellipse function has 4 parameters, used to determine the x,y coordinates of the center, and its size.
- We provided arguments of 70, 40, 50, and 30 to those 4 parameters.

Example

- Conversely… we have:

```
noStroke();
```

- The noStroke function has no parameters, but we still need to put the empty parens!

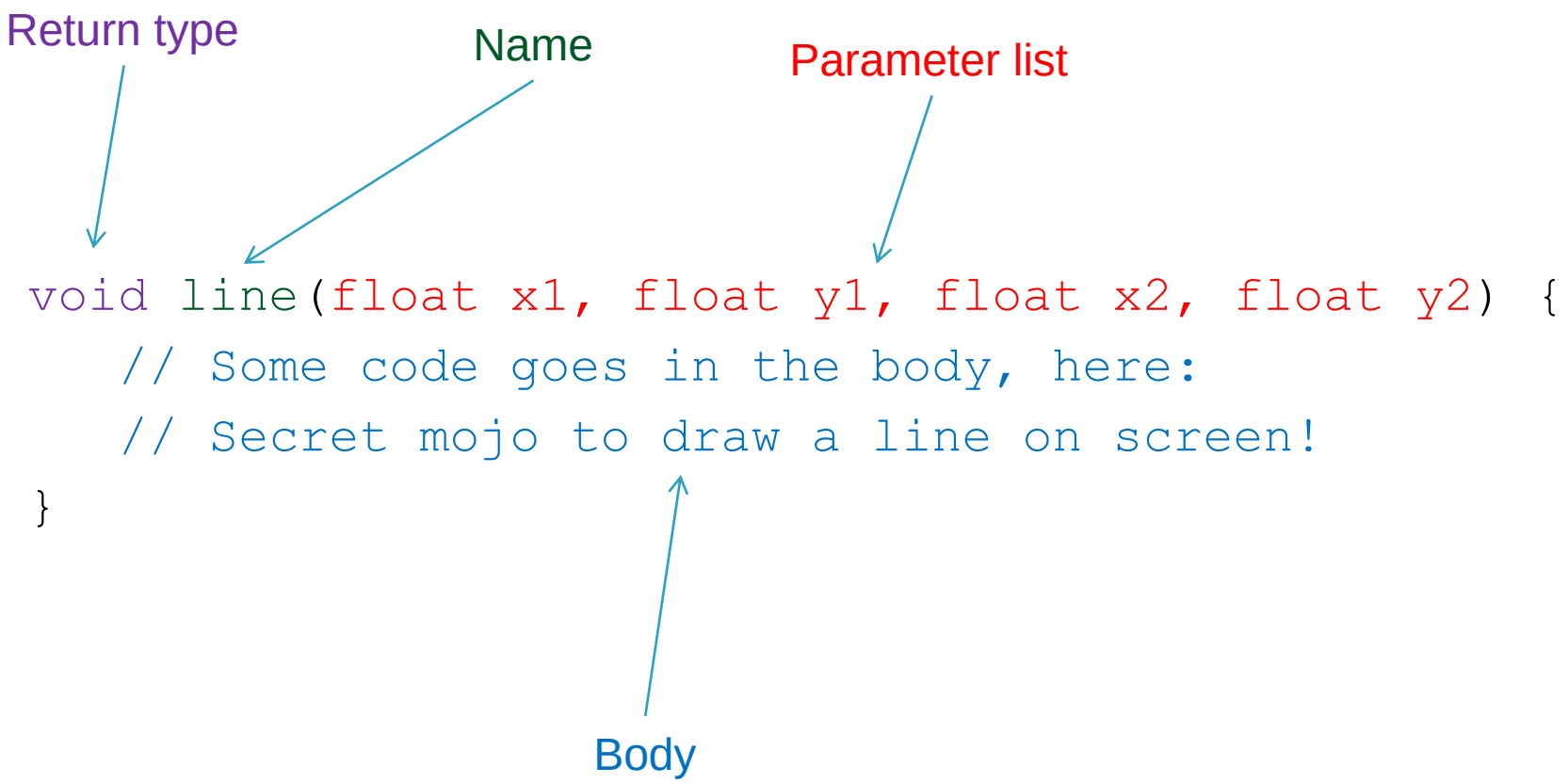
Defining a Function

- As I mentioned before, many functions are built-in to Processing. They already exist, and you can use them anywhere.
- You can also *define* a custom function of your own, to give a name to any block of code you want.
- Then, anytime you want to run that block of code, invoke your function!

Defining a Function

- To define a function, you need:
  1. A *header* or *signature* for the function, which has several parts:
     1. A *return type* (more about this later)
     2. Its name
     3. A list of the parameters (they look like variable declarations) separated by commas and surrounded by a pair of parens.
  2. A block of code surrounded by { } called the function's *body*.

Defining a Function



Example Custom Function

- Let's define a toy function called `sayHi`, with no parameters.
- Then, let's invoke it in our program.

```
void sayHi() { // define my function
   println("Hi there!");
}
void setup() {
   sayHi(); // invoke my function
}
```

Not-So Spoiler Alert: `setup` and `draw` are functions!

- In the last example, we defined the `setup` function. Its body contained just one thing: invoking the `sayHi` function.
- In order to make an event driven program, all we have to do is define the `draw` function.
  - The `setup` function is optional.
- `Setup` and `draw` are very **special** functions: The computer **automatically invokes them for you.**
  - Setup gets invoked once, first.
  - Draw gets invoked over and over, on a timer.

Let's draw a tree!

- Remember the tree from animation example 2? Here's a similar tree:

```
noStroke();
fill(50, 50, 0);
rect(50, 80, 20, 20);
fill(75, 150, 75);
triangle(35, 80, 85, 80, 60, 40);
triangle(35, 60, 85, 60, 60, 20);
```

Can't see the forest... for the trees!

- What if we wanted to make a forest of trees, say 3 trees for starters.
- We could copy-paste the code twice, but that would put 3 trees on top of each other, and it would still look like one tree.
- We could painstakingly edit the numbers on each tree to put them in different places.
  - How tedious! You couldn't see the forest for the trees! (ba dum tsssss)
- Let's make a function to draw a tree.

A makeTree Function

- We can put the code to draw a tree into a function, then invoke it.

```
void setup() {
  makeTree();
}
void makeTree () {
  noStroke();
  fill(50, 50, 0);
  rect(50, 80, 20, 20);
  fill(75, 150, 75);
  triangle(35, 80, 85, 80, 60, 40);
  triangle(35, 60, 85, 60, 60, 20);
}
```

A makeTree Function

- Note:
  - If you're going to define your own functions, you have to make it a "setup and draw" style program, even if you don't animate anything.
  - If you're making a procedural program, you can omit draw

makeTree with Parameters

- If we invoke `makeTree` 3 times in setup, it will draw 3 trees, but we're back to the same problem as before – they're in the same place!
- If we put in parameters for the `makeTree` function, we can use them to say **where** to put the tree!

makeTree with Parameters

```
void makeTree (float x, float y) {
  noStroke();
  fill(50, 50, 0);
  rect(x-10, y+60, 20, 20);
  fill(75, 150, 75);
  triangle(x-25, y+60, x+25, y+60, x, y+20);
  triangle(x-25, y+40, x+25, y+40, x, y);
}
```

- Think of the topmost point of the tree as x,y.
- Phrase all coordinates **relative to x,y.**

makeTree without Parameters

```
void makeTree () {
  noStroke();
  fill(50, 50, 0);
  rect(50, 80, 20, 20);
  fill(75, 150, 75);
  triangle(35, 80, 85, 80, 60, 40);
  triangle(35, 60, 85, 60, 60, 20);
}
// This slide is here for comparison.
```

Function Overloading

- `makeTree` with the two float parameters is more useful than the one without, right?
- But, it's still possible to define **both** of them!
- They are considered different functions because their headers are different.
  - So, they are invoked differently. You and the compiler can tell them apart by **how they are invoked!**
  - One of them must be invoked with 2 arguments. The other with 0 arguments.

Function Overloading

- This is called *overloading*.
- We say that `makeTree` is *overloaded*, because two different functions named makeTree exist.
- In reality, the two functions are completely separate.

In-Class Lab 1

Tips for Writing Functions

- Make each function short and to the point.
- Every function should have a single, easy to explain purpose.
- Name the function and name each parameter something descriptive.
- **Put a comment above every function you ever define**, briefly saying what it does and what its parameters are for.

Let's write some visual functions!

- Draw a house around a given (x,y) coordinate pair, with a given r g b color for the paint.
- Draw a horizontal street of houses at a given height on the screen, with a road in front.
- Draw a town of streets of houses.

In-Class Lab 2

Return Values

- Some functions actually *return* a value once they finish running.
  - For example, the random function returns a value of type float.
- There are actually two overloaded random functions in the library, and their definitions look like this:

```
float random (float upper) {
  …
}
float random (float lower, float upper) {
  …
}
```

- Notice the *return value* / *return type* is float. That means when the function completes, whoever invoked it gets a float number calculated by the function.

Return Values

- You can define custom functions with return values.
- Here's a function that computes the square of its parameter.

```
int squareIt(int a) {
  return (a * a);
}
```

- The *return* statement ends the function immediately (kinda like a break statement immediately ends a conditional), and returns the specified value to the invoker.
- This function doesn't draw anything. It's only purpose is to calculate and return the result.

Return Values

- Here's an example of a function that has two int parameters, and returns an int value equal to the larger of the two:

```
int maximum(int a, int b) {
  if(a > b) {
    return a;
  } else {
    return b;
  }
}
```

Expressions Using Functions

- You can use a function with a return value anywhere you can use a number or a variable value.
- The expression `1.0 + 2.0` evaluates to 3.0
- The expression `random(1) + 2.0` gets evaluated by:
  1. Invoking random(1) and obtaining the return value – a float number, and
  2. Using that returned number in its place, adding it to 2.0. The answer is between 2.0 and 3.0.

Examples on Moodle

- `makeTree` function definition, both versions
- `maximum`
- `squareIt`
- Many invocations of these functions, to demonstrate how they can be used.

Let's write a boolean function!

- Design a function to determine if a given coordinate pair is on screen and in the top half of the display window.
- This function should return the answer: whether it is or isn't.
  - In setup, use a loop to make twenty circles of diameter 5 at random locations on the screen.
  - Each circle in the top half of the screen should be green, and each circle in the bottom half red.
    - Use that first function to decide each circle's color.

Let's write some math functions!

- Input a number grade from 0 to 100, and output the corresponding letter grade. (e.g. an input between 90-100 would return the char value 'A')
- Decide whether a single input number is a multiple of 7, or not.
- Calculate the Euclidean distance between two pairs of points – based on HW2.
  - Also: Make a “Manhattan” distance function.

In-Class Lab 3

Recap: Defining a Function

- A function is like a recipe. *Define* the function to write down the recipe of what to do. *Parameters* are used to control what the function does. Remember, the code won't actually run until it is *invoked*.
- When defining a function, ask yourself:
  - What am I trying to do with this function?
  - What should the return type be?
  - What should we name the function?
  - What are the parameters?
    - For each one: what type, what name, and what purpose?
  - What code should we put in the body, and how do we use the parameters there?
  - What value is returned, if anything?

Recap: Invoking a Function

- *Invoke* a function to run the code in its body – to "make it happen." *Arguments* (input values) given to each parameter customize what the function does **this** time.
- When invoking a function, ask yourself:
  - Why am I invoking this function? What do I want it to do for me?
  - How many parameters does it have, and what types?
  - Does the function use each parameter for?
  - What arguments (values) do I want to give its parameters?
  - What will the function do when I invoke it with those arguments?
  - What return value will the function give, if any?
    - What am I going to use the return value for?

CSCI 182: Introductory Programming for Media Applications

ASCII Characters, Strings, and Display Window Text

ASCII

- The `char` variable type holds a single symbol – letters, numbers, punctuation, etc.
- There are 256 different symbols. An *ASCII Chart* or *ASCII Table* lists all the symbols. (Google it)
- Each symbol is numbered.
  - For example, @ is number 64, and the tilde ~ is 126.
- In fact, each symbol **is** its number!
  - The `char` data type is numeric, the same size as a byte variable. The *ASCII value* number is what's stored.

ASCII

- Use single quotes around character values such as `'a'`, `'Z'`, and `'$'`.
  - `x` is a variable name.
  - `'x'` is a character value – a symbol
  - `char symbol = '#';`
- You can use the ASCII value, directly:

```
char ch = 65;
println(ch); // prints capital letter A
```

ASCII

- Remember, the character **is** its ASCII value.

```
if('a' == 97) // this is true
```

- You don't have to memorize the ASCII table for this class, but I **strongly recommend** familiarizing yourself with it.
  - At least remember that 'A' is 65, and 'a' is 97.

In-Class Lab 1

String Variables

- A *String* (capitalize the S!) is another variable type.
  - It's actually a very special kind of variable called an *object*, but more on those later!
- It can hold a sequence of ASCII characters of almost any length: a word, a sentence, an essay.
- Use double quotes around String values.
  - Also, notice the capital S in String, here:

```
String s = "Heidely-ho, neighborino!";
println(s);
```

String Variables

- A String has a bunch of extra features beyond primitive types. They're "built-in" to every String.
- The + operator is used to add numbers, but…
- Use + on two Strings to ***concatenate*** them (glue their contents together side by side).
  - The + operator is overloaded for different types of parameters, just like a method.
  - Spoiler alert: Operators **are** methods!

String Variables

```
println("Adam" + " Reagan");
```

- As long as **at least one** operand of + is a String, the other will be implicitly type converted into a String, then concatenated. (Remember, it's temporary!)
- Notice the space before Reagan…

String Variables

- Here's another example using String concatenation:

```
int i;
for(i = 0; i < 100; i++) {
   println("Iteration #" + i);
}
```

String Methods

- Strings have a **lot** of built-in features!!!
  - http://docs.oracle.com/javase/7/docs/api/java/lang/String.html
- There are dozens of methods **inside** every String, to do useful things.
  - (Objects are cool, like that!)
- Some of the more useful String methods:
  - `Substring(...)`
  - `length(...)`
  - `charAt(...)`

String Methods

- To invoke a String method, put (1) the name of a String variable, (2) a dot, then (3) the invocation of a String method.
- The method will be invoked **on that String!**
- The `length` String method is straightforward. Returns the number of characters in the String.

```
String s = "Hello";
String t = "Goodbye";
println(s.length()); // 5
println(t.length()); // 7
```

String Methods

- The `charAt` String method has a return type of char. It returns the character at the given position a.k.a *index*.
  - Remember, here in the CS dept we count from 0!

```
String s = "Adam"; // A is at index 0
char c = s.charAt(3);
println(c); // prints the letter m
```

String Methods

- The first character (at index 0) of any String named `x` is always `x.charAt(0)`
- The last character of any String `x` is `x.charAt(x.length()-1)`

String Methods

- The `substring` String method (actually, two overloaded methods) cuts out a piece of a String, and returns just the piece specified.
  - The one parameter is the **first index you want** (inclusive). Returns a String containing everything from there to the end.

```
String s = "Adam Reagan";
String t = s.substring(2);
println(t); // am Reagan
```

String Methods

- If you provide a 2nd argument, that's the **end** of the piece you want (exclusive). You'll get everything **up to but not including** that index. (How would I get all but the last char?)

```
println("Hello".substring(1, 3)); // el
```

- Oh, yeah – you can invoke String methods on String variables, **and** String literal values like `"Hello"`.

In-Class Lab 2

Text in the Display Window

- You can put text in the display window, in any color, size, location, and font (see Example code).
- Beforehand, you have to add a font to your program using the Tools menu. A file containing the font gets placed by the editor into a folder named data, inside your sketch folder.
  - A Processing program is really a folder of files, including the source code .pde file, font files, etc.

Text in the Display Window

- The font file has a built-in default font size, conveniently written into the filename!
  - You can use any size font with that file – it's a default.
  - If you don't specify a font size in your source code, the default font size in the file is used.

Pro Tips

- `PFont` is another variable type, used to hold an entire font.
  - Spelled with a capital P and a capital F.
- Remember: you can make text animate & move just like ellipses, and all other shapes.
- The coordinates to the `text` method are the **lower left corner**.
- Fill color affects the color of fonts **and** the insides of shapes.
- Choose the `textFont` once, and it'll be the active font used to write all further text, until you change it.
  - It's a setting just like stroke, strokeWeight, fill, noStroke, etc.

In-Class Lab 3

CSCI 182: Introductory Programming for Media Applications

Interactive Animation
(Chapter 7)

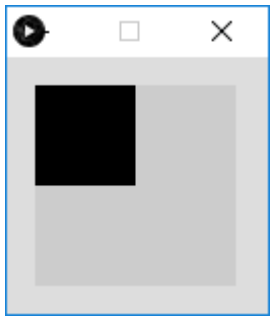
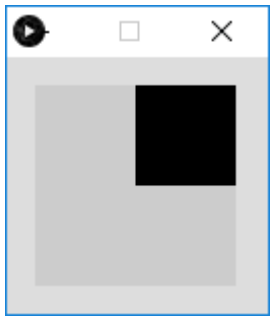
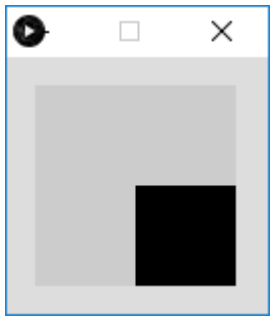
Built-In Variables

- The built-in variables `mouseX` and `mouseY` are the (x,y) coordinates of the mouse pointer in the display window.
  - Mouse coordinates are not recorded if outside the window.
- Here's an example where one end of a line follows the mouse:

```
void setup() {
  size(400, 400);
  frameRate(60); // try different numbers
}
void draw() {
  background(255, 150, 100); // move to setup?
  line(mouseX, mouseY, 50, 30); // try ellipse?
}
```

Built-In Variables

- `mouseX` and `mouseY` are numbers, so we can use them in boolean and math expressions.
- By doing "range of values" boolean expressions, we can highlight areas of the screen. Something like…


Built-In Variables

- There are two more built-in variables: `pmouseX`, `pmouseY`. They record the mouse location from one frame earlier!
  - Try modifying our earlier example to use:

```
line(mouseX, mouseY, pmouseX, pmouseY);
```

- We can't see further back in time without doing it manually using custom variables (or arrays…).
- `mouseX`, `mouseY`, `pmouseX`, `pmouseY` are all int variables.

Built-In Variables

- The built-in variable `mousePressed` is a boolean. It tells whether you're holding down a mouse button.
- It is **always false**, unless we hold down a mouse button to make it true.
  - It goes back to false when you let go.
- The best way to use `mousePressed` is in an if statement's boolean expression, inside the draw function definition

Mini-Lab 1

- Have the car from your Homework 5 follow the mouse pointer around the screen.
  - This is an animation, so you'll need to be inside draw() rather than setup().
  - Invoke your drawCar method with mouseX and mouseY.
    - If you haven't finished that part of your homework yet, then just make a car with a rectangle and a couple of circles, for now.

Built-In Variables

- The built-in variable `mouseButton` stores a value indicating **which** mouse button is being held down right now.
- Compare it to the constant values `LEFT`, `RIGHT`, or `CENTER` in a boolean expression, like this:

```
if(mouseButton == LEFT)
```

- Once you release the mouse button, it loses that value.
  - Technically, you have to release the mouse and then move the cursor, then it will lose that value.
  - This behavior changes subtly with each version of Processing.

Special Functions

- We can also do mouse interaction with some "special" functions.
- Remember `setup()` and `draw()` are "special", because they are **invoked for you** at certain times.
  - You never manually invoke them yourself.
- Another special function is `mousePressed()`, not to be confused with the `mousePressed` built-in variable.
  - They're totally different, despite the similar names!

Special Functions

- `mousePressed()` is automatically invoked **once** each time you click any mouse button, at the instant you press down.
  - It doesn't matter if you hold it down or not. This function only reacts at the instant you click down.
    - (By contrast, the `mousePressed` variable is true as long you hold a mouse button, then false when you release the mouse.)
- To use it, define a zero-argument `mousePressed()` function with a return type of void.

```
void mousePressed() {
  // Code here is executed once each click.
}
```

Special Functions

- The `mouseReleased()` special function is automatically invoked once every time you **release** a mouse button – when you "unclick".
  - It's the opposite of mousePressed().
- Try changing our last example. Replace the definition of `mousePressed()` with `mouseReleased()`, then run it again & try it out.

Special Functions

- In the real world, programs almost always react when you release mouse buttons, **not** when you press them. (Menu buttons, for example.)
  - So, you should usually use `mouseReleased()` instead of `mousePressed()`. Users tend to expect it.

Special Functions

- The `mouseMoved()` and `mouseDragged()` special functions are similar to each other.
- `mouseMoved()` is automatically invoked once every frame, but **only** if you moved the mouse that frame **without** holding any buttons down.
- `mouseDragged()` is automatically invoked once every frame, but **only** if you moved the mouse that frame while **holding** a button down.
- These two functions are **mutually exclusive!!!** On any single frame, they won't both be invoked.
  - If you're holding down a mouse button when you move the pointer, you're "dragging" the mouse.
  - If you're **not** holding down a mouse button when moving the pointer, you're "moving" the mouse.

Summary of Mouse Interaction

- Built-in variables:
  - `mouseX`, `mouseY` – location of mouse pointer now
  - `pmouseX`, `pmouseY` – location mouse pointer 1 frame earlier
  - `mousePressed` – true if we're holding down a button, false if we're not
  - `mouseButton` – tells us which mouse button is currently being held down. Compare to `LEFT`, `RIGHT`, or `CENTER`
- Special functions you can define:
  - `mousePressed()` – invoked once per click, as you click down
  - `mouseReleased()` – invoked once per click, as you let go
  - `mouseMoved()` – invoked every frame you've been moving the mouse without holding down any button
  - `mouseDragged()` – invoked every frame you've been moving the mouse while holding down a button
  - **Note**: Those last two never get invoked on the same frame!

Mini-Lab 2

- Modify Mini-Lab 1 so that the car is only visible while you hold down the mouse, and invisible when you let go.
- Two different ways to do this:
  - 1) Make the drawCar invocation conditional on the mousePressed variable being true.
  - 2) Cut-paste the drawCar invocation from inside draw() to inside the definition of mouseDragged().
    - If you do it this way, the car won't show up if the mouse stays still – you have to be dragging the mouse to see it.

Keyboard Interaction

- Built-in variables:
  - `keyPressed` – boolean. True iff (if and only if) a keyboard button is being held down.
  - `key` – char. The most recently typed ASCII value.
    - If the most recently typed keyboard button was non-ASCII, it has the special value `CODED`.
  - `keyCode` – The most recently typed non-ASCII value.
    - Usually examined only when `key == CODED`.
    - Compare it to values such as `UP, DOWN, LEFT, RIGHT, SHIFT, ALT`

Keyboard Interaction

- Special functions you can define:
  - `keyPressed()` – invoked each time you type any keyboard button.
  - `keyReleased()` – invoked each time you release any keyboard button.
- **Note:** keyPressed() and keyReleased() get repeatedly invoked while holding down a keyboard button. (~20/sec)
  - But, the keyPressed boolean variable stays true the whole time!

Animation Control

- Built-in functions you can invoke:
  - `noLoop()` – Stops `draw()` from being automatically invoked.
    - Pauses the animation. Similar to `frameRate(0)`
  - `loop()` – Resumes automatically invoking `draw()` several times per second (based on the frame rate).
    - Reverses the effect of `noLoop()`
    - **Do not** attempt to invoke `loop()` inside `draw()`! It's a chicken-and-egg issue. Invoke it from another special function such as `mousePressed()`
  - `redraw()` – Invokes `draw()` exactly once, to manually make one frame. Notice the lowercase d in the name.
    - Usable even when the animation is stopped with `noLoop()` or `frameRate(0)`.
    - **Do not** attempt to invoke `redraw()` inside `draw()`!

Mini-Lab 3

- Add a pause button p and a resume button r to your Homework 4 animation.
- Here's how:
  - Define the keyPressed() function. Inside it:
    - If the user pressed p (you can examine the key variable), then noLoop() will pause the animation.
    - If the user pressed r, then loop(); will resume the animation.
- Remember to check out the **huge** number of example programs on Moodle!

CSCI 182: Introductory Programming for Media Applications

Arrays

(Chapter 11.2 – 11.4, p. 232 – 252)

Arrays

- An ***array*** is a collection of variables of the same type (primitives like int, or objects like String).
- Each *element* (i.e. each variable or item) in the array has a number called it's *index* or *offset*.
- Indexes start at 0, and go up to the length of the array – 1.
  - Think of a long row of post office boxes, each with a box number written on the front.
  - Very similar to the characters in a String – they also have indexes.

Arrays

- Arrays are **really useful**, any time you want to keep track of lots of the same kind of thing:
  - Pixels of an image
  - Homework grades in a course
  - Snowflakes or stars in the sky
  - Days of the month
  - Names of everyone in a class, etc.

Declaring Arrays by Initializing

- There are two ways to declare an array:

  1) Initialize each of the array's elements to an initial value.

```
int [ ] numbers = {1, 2, 3, 4, 5};
```

variable type of elements

square brackets mean it's an array

name of the array

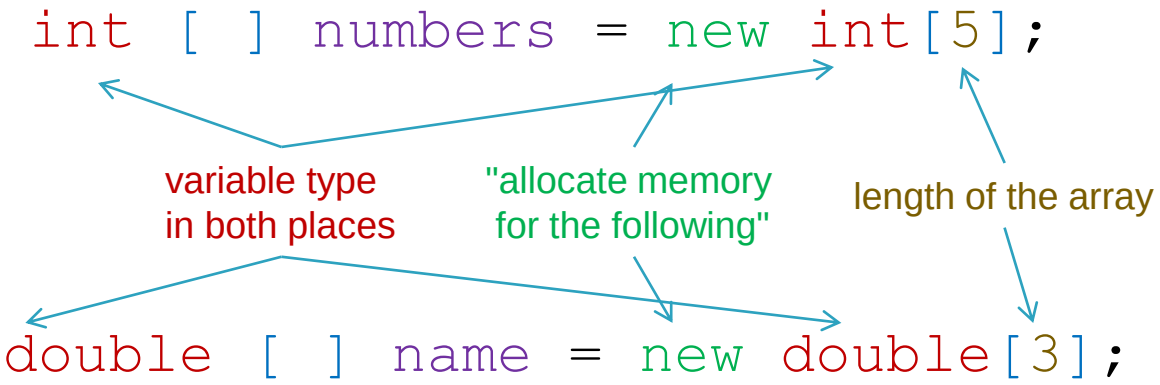
initial values of elements

```
String [ ] name = {"Adam", "L", "Reagan"};
```

- The length of the array is **implied** by how many initial values you give.
  - So, `name` is length 3 and `numbers` is length 5.

Declaring Arrays by Length

- 2) Specify the length, leaving element values unspecified.
  - Elements will get "default" values, such as 0 for int, 0.0 for float.
  - Don't do it this way with objects like String! (Learn why later!)



- The length of the array is explicitly specified.
- You have to specify the variable type twice.

`char` Arrays vs. Strings, & Examples

```
char [] myName = {'A', 'd', 'a', 'm'};
```

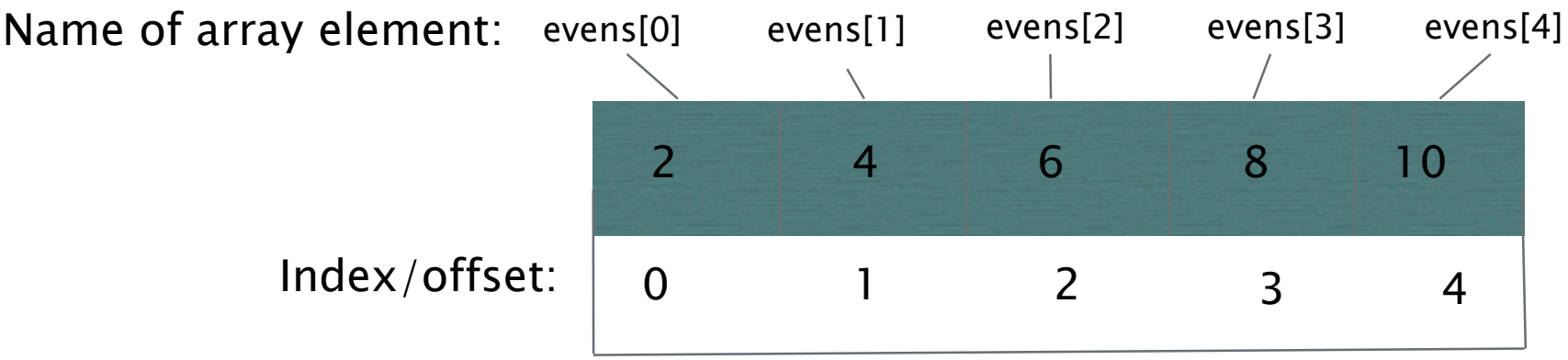
- A `char` array is similar to a String – both are an ordered sequence of chars, with indexes. But, Strings also have other methods built in, like `substring()`.
- Every array also has a built-in int **variable** called `length` inside it, in addition to having all its elements. `length` tells you how many elements are in the array.
  - Not to be confused with the similar `length()` method inside a String!

```
float [] nums = {124.34, 397.2, 8562.3};
println(nums.length);  //3
String s = "Adam";
println(s.length());   //4 but notice the parens!
```

Accessing Array Elements

- Each element of an array has a "box number" called its *index* or *offset*. These elements are normal variables, and you can use each one separately.

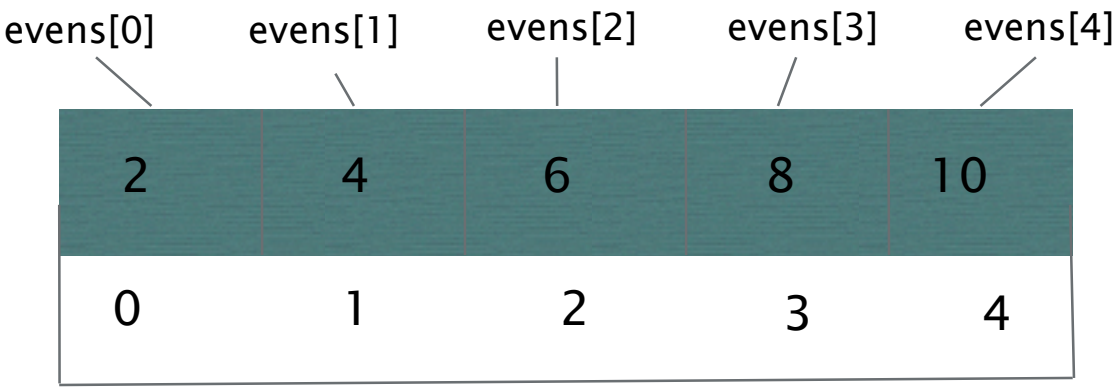
```
int [ ] evens = {2, 4, 6, 8, 10};
```



Accessing Array Elements

- Each element of an array has a "box number" called its *index* or *offset*. These elements are normal variables, and you can use each one separately.

```
int [ ] evens = {2, 4, 6, 8, 10};
println(evens[0]); // prints 2
println(evens[4]); // prints 10
```



Mini-Lab 1: Declaring & Accessing

- Make variables for 5 coordinate pairs, using arrays. Here's how:
  - Declare an array of five int variables called xCoords. Initialize the elements as you declare the array.
  - Declare another array of five int variables called yCoords. Initialize the elements as you declare the array.
- Use your drawCar method again to draw 5 cars: one at each of those 5 coordinate pairs. Here's how:
  - Paste your definition of drawCar into this program.
  - Five invocations of drawCar in setup() will be enough: one for each coordinate pair you made.
    - Each invocation of drawCar will need a coordinate pair as arguments. Give it an element of xCoords, then give it an element of yCoords, then give it more arguments for any color you want. (See Slide 8 as a reference)

`for` loops and arrays go together like peanut butter and jelly!

- `for` loops and arrays are made for each other.
- You can use a loop to access each element of an array. Use the loop variable as an array index!
  - Example of putting the even whole numbers into an array:

```
int [ ] numbers = new int[100];
for (int i = 0; i<100; i++) {
  numbers[i] = 2*i;
}
```

Or, use `numbers.length` here!

Changing an Array's Length

- You can change the length of an array, by removing or adding a new element at the high end.
- The `shorten` method removes the last array element and shortens the array. It returns a new array identical to the old one, but one index shorter!
- The `append` method lengthens the array by one, and puts its argument into the newly created element.

Changing an Array's Length

```
String[] trees = { "persimmon", "coconut", "fig"};

trees = append(trees, "guava");
// What is the length and contents of trees, now?

trees = shorten(trees);
// What about now?

println(trees); // println is cool w/ arrays of Strings
println(trees.length);
println(trees[1].length());
```

Mini-Lab 2: Arrays with Loops

- Take your code from Mini Lab 1, and modify the code so it makes 50 cars instead of 5 cars! You can do it all from inside setup().
  - Make sure your display window is pretty big – 600 by 600 at least!
  - Your arrays will have to be length 50. Change the declarations to declare by length, instead of by initializing.
  - Run through xCoords with a loop, and set the current element of xCoords to a random number from 0 to `width`.
  - In the body of that same loop, you can also set the current element of yCoords to a random number from 0 to `height`.
  - Finally, run another loop (use loop variable i) through xCoords, and each iteration invoke drawCar with element i of xCoords and element i of yCoords. Make all the cars the same color, for now.
- (You could've done this without an array, by invoking drawCar in a loop with random arguments for x and y. But, don't!)

Parameters: Passing by Value

- When you pass a variable as an argument to a function, you are making a **copy** of the argument variable into the parameter variable. This is called *pass-by-value*, or a *value parameter*. (Everything we've done thus far!)
- Changes to the parameter value do **not** affect the original argument variable, because we made a copy.

```
void setup() {
  int x = 5;
  foo(x); // value of x copied into q
  println(x); // 5, still. Not 10.
}

void foo(int q) {
  q = 10; // modifying q, not the argument
}
```

Parameters: Passing by Reference

- Array parameters work differently!
- If you pass an array into a function, that array is **not copied**. Instead, a reference to the same array is given to the function. This is called *pass-by-reference* or a *reference parameter*.
- So, changes to the array within that function call **do** affect the original argument array. It's the same array!

Parameters: Passing by Reference

```
void setup() {
  int [] x = {5, 10, 15};
  foo(x);
  println(x); // x[0] is now 99
}

void foo(int [] q) {
  q[0] = 99; // modifying the array passed
}
```

In-Class Examples of Array Functions

- A function named addToElements. It should have:
  - No return value
  - Two parameters: a float array, and also a float.
  - The function should add the float's value to every element of the array. Use a loop.

In-Class Examples of Array Functions

- A function named arrayMax.
  - It's return type is int.
  - It's single parameter is an int array.
  - It's purpose is to find the maximum value element in the provided int array, and return that value.
  - How? Here's the idea:
    - Use a local variable biggestSoFar to hold the biggest value seen so far. Initialize it to the value of element 0 of the array.
    - Iterate through the array checking each element to see if its value is bigger than biggestSoFar. If so, then this value should replace biggestSoFar, because it's bigger! (Illustration on the board.) At the end, biggestSoFar will be the very biggest value.

In-Class Examples of Array Functions

- A function named arrayCopy. It returns a newly created copy of the given array of the same length with copies of all the values.
  - It's return type is an int array.
  - It's single parameter is an int array.
  - How? Here's the idea:
    - Make an int array local variable of the same length as the parameter array.
    - Use a loop, where the loop variable iterates through the new array from the first index to the last index. Make sure it works for **any length**!
    - Each iteration, copy the current value (index given by the loop variable's value) from the old array into the same index in the new one.
    - Once that loop is done, return the newly made array.
      - Since the new array is a reference, it will persist after the function ends!

Mini-Lab 3: Array Functions

- Take your code from Mini Lab 2, and make all 50 cars move to the right one pixel per frame!
  - xCoords and yCoords will need to be global variables, so move them to the top of the program.
  - Define a new function named moveRightOne. It'll move the x coordinates of any given array to the right by one pixel. Here's how:
    - Return type void
    - Name moveRightOne
    - Parameter is an array of int variables, with any name you want. You could call it x
    - In the body, run a loop through the parameter array (loop variable i), and increment element i of the array. You can use ++ operator
  - We're making an animation, and we'll draw all the cars every frame. Make the background every frame, then cut-paste the drawCar loop from setup to draw.
  - After the drawCar loop, invoke moveRightOne on the xCoords array. This makes sure the cars keep moving every frame.

CSCI 182: Introductory Programming for Media Applications

Abstraction: Objects & Classes
(Chapter 14)

Objects

- An object is a complex variable, like a container, holding:
  - ***Data fields*** (i.e. variables), storing information about the current state
  - ***Methods***, which are invoked to manipulate & use the object.
- Object Oriented (OO) Programming is an approach / style of writing code with objects. An OO program is a collections of these objects, all communicating with each other.

Objects

- You can think of an object as an abstract black box. It's a conceptual entity that you can use (via method calls) to do something useful.
- You don't need to "peek inside the box" or understand how it works. All you need to know is how to **use it**: how to invoke its methods, and what they can do for you.

Information Hiding / Encapsulation

- The internals of an object are actually hidden from its user. This is called ***information hiding*** or ***encapsulation***.
  - I don't know the actual code definitions of all the String methods. I don't need to! I know how to use them.
- The **only** way to interact with an object is with well-known, documented *public* (usable by anyone) methods. The internal details are *private* (access denied).
  - Again, try to think of the object as an abstract thing. Who cares how the thing actually works – just use it! (Car analogy.)

Classes

- **A class is a variable type.**
  - It defines data fields (a.k.a *fields*, member variables, or members) as well as methods (a.k.a. *class methods*).
- **Each object is an instance of a class.**
  - It's a variable whose type is a class. The class definition describes how to use the object.
- Every object contains a copy of everything defined in its class.
- For example, you could define a class named Dog, and then declare several objects of that type: Fido, Rover, and Lassie.

Classes

```
String s = "Adam";
```

- `String` is a class, and s is an object (a variable) of type `String`. The data fields of s store `"Adam"`
- PFont is another class we've learned about. There are **many** more classes defined in the Java Standard Library that we can use.
- You can also define your own class!

Syntax of a Class

```
class DayOfYear {
    int dayOfMonth; //Class data fields
    int dayOfYear;

    //Constructor – We'll get to this

    //Other Class Methods – Coming Soon!
}
```

- Let's look at the Example Programs 1-3 on Moodle.

Mini-Lab 1: Defining a Class

- Define a class of your own to represent a physical object.
  - Name it after what it represents: Tree, Car, Frisbee, etc.
- Declare data fields in your class for the information is needs to hold.
  - For starters, give it a coordinate pair of its location.
  - Give it at least one other data field for some other property it will have, like color, size, tallness, shape, etc.
- Define a method named void display() to cause the object to appear in the display window.
  - Use the data fields to depict it.
  - Use at least 6 graphics primitives.

Constructor

- A *constructor* is an important kind of class method.
- It describes how to create new objects of its class, and how to initialize their data fields.
- When you declare a new object variable, you've merely created a handle. It's an empty shell that can be used to refer to an object.
- For example:
  - In order to create a new object, you must invoke its constructor. Only then does the handle refer to an object.
  - If you don't define a constructor, a default constructor is automatically made; it sets data fields to default values.

Constructor

- Some constructors have parameters, allowing you to specify how to make the object.
- Here's how you **define** a constructor:

```
class ClassName { // class definition
  // This is a constructor definition
  ClassName(parameter_list_here) {
    // initialize data fields
  }
}
```

- Here's how you invoke a constructor to create a new object:

```
ClassName objectName = new ClassName(args);
```

See Example 4!

Mini-Lab 2: Constructors

- Define a constructor for your Mini-Lab 1 class!
  - Give the constructor one parameter for each data field.
  - In the body of the constructor, assign from the parameter to the data field.
- Use your new variable type!
  - Outside your class, define void setup().
  - Declare and construct at least 3 different objects of your class.
  - Invoke display on each one of your objects.

In-Class Example

- A class called `DayOfYear`, to hold a calendar date like June 17th.
  - To keep things simple, let's pretend every month has 31 days and there's no such thing as a leap year!
- The data fields are:
  - `int dayOfMonth`, which needs to stay between 1 and 31
  - `int monthOfYear`, which needs to stay between 1 and 12
- The constructor will have a parameter meant to be assigned into each data field.
  - So, when constructing a new `DayOfYear` object, you specify a day and a month.
- The other class methods:
  - `getDay`. Returns the day of the month stored.
  - `getMonth`. Returns the month of the year stored.
  - `advanceDay` method, to move the day forward. Jan 31 goes to Feb 1, etc.
  - `printDay` method to print the day's name in "January 15" format.
- More examples: Example Programs 5 – 7!

Mini-Lab 3: Array of Objects

- Modify your Mini-Lab 2 code to make an array of 50 objects of your class.
  - Instead of 3 objects, declare an array of 50 objects by length.
  - Instead of separately constructing 3 objects, loop through the elements of your array and construct `array[i]`
  - Instead of invoking display on each of 3 objects, loop through the elements of your array and invoke display on `array[i]`

CSCI 182: Introductory Programming for Media Applications

Introduction to Java with NetBeans

Java

- As we discussed at the beginning of the semester, Java is an OO programming language that Processing is built upon.
  - (See the Introduction slides for more on Java)

- To create Java programs on your computer, you must install the *Java Development Kit* (JDK), free at:

  http://www.oracle.com/technetwork/java/javase/downloads/index.html
  - Download the "Java Platform (JDK)", which contains:
    - The Java Compiler, for compiling Java source code into Java Byte Code.
    - The Java Runtime Environment, for running Java Byte Code.
    - … and so much more!

Java vs. Processing: Similar

- Java is *similar* to Processing. The basics are ***mostly the same***:
  - Primitive types, arithmetic, assignment statements
  - Variable scope (local, global, and data field variables)
  - Boolean arithmetic, if-else & switch statements, loops
  - Arrays
  - Functions (call them methods, in Java)
  - Classes, handles, objects

Java vs. Processing: Dissimilar

- Some things are **not the same**, such as:
  - Java applications begin at the special method called `void main(String[] args)` instead of `setup` & `draw`.
  - By default, you don't have the display window to use.
    - Java does have its own graphics libraries, different from Processing.
    - Later, I'll also show you how to get an actual Processing display window through a Java program.
  - Most built-in Processing variables/functions like width, height, frameCount, random, and mouseReleased don't exist in Java.
    - The Java Standard Library has everything you need, though!
  - Numeric constants in Java like `5.0` are considered type double, not float.

Java vs. Processing: Dissimilar

- Some more things are **not the same**, such as:
  - **Every function must be a class method**. You can't simply define a function on its own, not even main. You have to put it in a class.
  - **Each class definition goes in its own file**, named after the class, with .java on the end.
  - You can enforce information hiding, by making data fields & methods private. (More on this later!)
  - Java does not support the `color` primitive type, nor HTML-style colors like `#ff00ff`
  - When typecasting, you must parenthesize the type being cast.

```
int x = (int) 7.5;
```

Java vs. Processing: Dissimilar

- Even more things are **not the same**, such as:
  - print and println statements are written like this:

```
System.out.print("Hello world");
System.out.println("!!!");
System.out.println("Oogabooga!");
```

NetBeans

- NetBeans is a program for writing and running computer programs in many languages, including Java.
- NetBeans is installed on all the lab computers.
- Download NetBeans on your computer for free:
  https://netbeans.org/downloads/
  - Get the "Java SE" bundle.
- The built-in **tutorial is highly recommended**!
  - On the Start Page, click "Java SE Applications" under the "Demos & Tutorials" heading.

Tips for using NetBeans

- To start writing a new Java program:
  1. File → New Project.
  2. Select the Java category at the top, then click Java Application.
  3. Click the Next button, type in a Project Name, then click the Finish button. Don't edit anything else in those menus.
- Type code into the big window on the right.
- Syntax errors are automatically highlighted in red on the left.
- **Alt – Shift – F to auto-format your code.**
- Ctrl – S to save & compile your code at the same time.
- **F6 to Run your program**, or click Run → Run Project.
- Use the dot operator . on an object to see a giant context menu of all your choices!

← And this is awesomesauce!

Mini-Lab 1: Porting Over the DayOfYear Example

- First, make a new Java Application.
  - File → New Project. Click Java Application, then click Next. Type a name for it, then click Finish.
  - This will automatically create a file with a class with an empty main method.
- Then, add the DayOfYear class to your Java Application.
  - File → New File. Click Java Class on the right, then click Next at the bottom. Type in the Class Name DayOfYear, then click Finish.
  - Open the DayOfYear class definition in Processing, and copy-paste the **body** of the DayOfYear class from Processing into the body of DayOfYear in your new file in NetBeans.
- Paste the body of void setup() from the Processing program into the body of the main method in Java.
- Fix the print and println statements.

Porting Processing to Java Using Display Window

- 0) Create a new Java application.
  - File → New Project. Click Java Application, then click Next. Type a name for it, then click Finish.
- 1) Put the Processing library into your Java application.
  - In the Projects viewer, click the + to open your project.
  - Right-click on Libraries, and click Add JAR/Folder.
  - Find core.jar from your Processing installation, and click open. It's normally found in your Processing installation folder under core/library.

Porting Processing to Java Using Display Window

- 2) Edit the top of your class to say

```
public class ClassName extends PApplet {
```

  - ClassName is the name of the class you made.

- 3) At the top of the same file, add this line below the package.

```
import processing.core.*;
```

- 4) Paste this line in the body of the main method.

```
PApplet.main(new String[]{"packageName.ClassName"});
```

  - `ClassName` is the name of the Java class with main in it.
  - `packageName` is the name of the folder that file is in.

Port Processing to Java Using Display Window

- 5) A Processing program can be written **inside** the body of your class. Think of this class as a Processing program.
  - You can even paste an existing Processing program into the body of this class, with setup and draw, etc. **But, remember:**
  - Everything in your Processing program, even global variables, go inside your Java class. Except that…
  - Each class needs its own file! So, if you defined a class in your Processing program, put it in another file. You will need to put

```
import processing.core.*;
```

  - Remember to fix print statements, and a couple of other small adjustments…

Port Processing to Java Using Display Window

- Put the word `public` to the left of every method definition.

```
public void makeTree(float x, float y) {
   // method body
}
```

- Put `@Override` on the line above every **special** method definition, such as draw and keyPressed.

```
@Override
public void keyPressed() {
   // method body
}

@Override
Public void draw() {
   // method body
}
```

Port Processing to Java: Processing Class

- If your Processing program had defined any classes that use the display window, you have to fiddle with it a bit more.

- This example Processing code…

```
// All in the same file: WhateverName.pde
class Spot {
  void display() {
    ellipse(50, 50, 50, 50);
  }
}

void setup() {
   Spot s = new Spot();
   s.display();
}
```

Port Processing to Java: Processing Class

- …becomes this Java code.

```
// In the file Spot.java
public class Spot {
  public void display(PApplet p) { // new param
    p.ellipse(50, 50, 50, 50); // use the param
  }
}

// In the file Example.java
public class Example extends PApplet {
  @Override
  public void setup() {
    Spot s = new Spot();
    s.display(this); // new argument
  }
}
```

More Examples – Port to Java

- Port the original snowy scene animation to Java in NetBeans.
  - (Example 2 from the Animation example code pde)
  - This Processing program didn't define a class – easy to port.
- Port the Kapow interactive animation to Java in NetBeans.
  - (Example 7 from the Objects and Classes example code pde)
  - This Processing program defined a class – complicated to port.

Mini-Lab 2: Animation in Java

- Port any Processing animation you've made (such as the homework 3 animation) to Java in NetBeans.
- Your entire Processing program can be pasted in the body of the same Java class with main. Remember the steps we went over!
- If the Processing animation defined a class, it's a lot more complicated (see the last couple of slides).