Review

- Models of Motion
  - Linear Translation
  - Bouncing
  - Rotation
  - Seeking a Target
  - Gravity and Friction
  - Accelerating toward a Target
  - Perspective (starfield)

Our Toolkit – A Review

- Graphics
  - lines, shapes, images, text, color, …
- Data of Various Types
  - Numbers - with (float, double) and without (int, long) decimal places
  - Booleans - true, false
  - Color - two color models
  - Characters and Strings
- Variables
  - Hold/name any type of data values
- Operators
  - Mathematical (+, *, ++, %, …)
  - Relational (<, >=, !=, ==, …)
  - Logical (&&, ||, !)

Our Toolkit (Continued)

- Expressions
  - Combine of data, variables, operators, functions
- Conditionals
  - if-statement
- Iterations
  - for-loop, while-loop
- Functions
  - Mathematical, Graphical, Utility, Events…
  - Of our own design
- Arrays
  - Functions that manipulate arrays
- Objects
  - State (fields), Behavior (methods / functions internal to class)
  - Of our own design using class statement

Primitive Data Types

Type	Range	Default	Bytes
boolean	{ true, false }	false	-
byte	{ 0..255 }	0	1
int	{ -2,147,483,648 .. 2,147,483,647 }	0	4
long	{ -9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807 }	0	8
float	{ -3.40282347E+38 .. 3.40282347E+38 }	0.0	4
double	*much larger/smaller*	0.0	8
color	{ #00000000 .. #FFFFFFFF }	*black*	4
char	*a single character* 'a', 'b', …	'\u0000'	2

Variables

- A *name* to which data can be assigned
- A variable is declared as a specific data type
- A variable is assigned a value using '='
- Variable names must begin with a letter, "_" or "$"
- Variables can container letters, digits, "_" and "$"
- Syntax:
  type name;
  name = expression;
  type name = expression;

```
int i;
float x;
int j = 12;
boolean bReady = true;
float fSize = 10.0;
color _red = color(255,0,0);
```

Variables are both declared and assigned a value

Comparing Declarations and Initializers

```
int      i;
int      j   = 3;
float    fac = 0.1;
float[]  Xs;
float[]  Ys  = new float[10];
float[]  Zs  = new float[] {1.2, 2.3, 3.4};
String   s1  = "abc";
String   s2  = new String("abc");
String[] s3  = new String[50];
String[] s4  = new String[] {"moe", "larry", "curly"};
Ball     b   = new Ball();
Ball[]   bs  = new Ball[200];
```

Operators

Symbols that operate on one or two sub-expressions.
Infix, prefix, or postfix

- Mathematical (+, -, *, /, …)
  - Perform standard mathematical operations (PEMDAS)
- Relational (<, >, ==, !=, …)
  - Test relationship between related expressions.
  - Always returns a boolean value (true or false).
- Logical (&&, ||, !)
  - Logical conjunction (and), disjunction (or), negation (not).
  - Always returns a boolean value (true or false).

Mathematical Operators

```
+, -, *, /      and …
i++;            equivalent to   i = i + 1;
i += 2;         equivalent to   i = i + 2;
i--;            equivalent to   i = i – 1;
i -= 3;         equivalent to   i = i - 3;
i *= 2;         equivalent to   i = i * 2;
i /= 4;         equivalent to   i = i / 4;
i % 3;          the remainder after i is divided by 3 (modulo)
```

Examples:

```
1 + 2
slope = (y2 - y1) / (x2 – x1);
i++
```

Relational Operators

<	less than
>	is greater than
<=	is less than or equal to
>=	is greater than or equal to
==	is equivalent
!=	is not equivalent

Examples:

```
10 >= 10
mouseY > 250
x % i == 0
name != "Fred"
```

Logical Operators

&& logical conjunction (and)
both expressions must evaluate to 'true' for conjunction to evaluate to 'true'

|| logical disjunction (or)
either expression must evaluate to 'true' for disjunction to evaluate to 'true'

! logical negation (not)
!true → false, !false → true

Examples:

```
mouseY > 250 && mouseX < 250
name == "Fred" || name == "Barney"
!ready
```

Evaluating Expressions

- PEMDAS
- Things to watch out for:
  - Integer division discards decimal places

```
11/10 -> 1
3/4 -> 0
3./4 -> 0.75
```

  - Operands are promoted to the most general type
  - Modulo (%) returns remainder after division

```
11 % 10 -> 1
3 % 4 -> 0
5.1 % 5 -> 0.1 (Actually 0.099999905)
```

Evaluating Modulo Expressions

Evaluate: a % b

1. r1 = a / b Perform integer division (regardless of type)
2. r2 = r1 * b
3. r3 = a - r2

a % b -> r3

```
// Both produce same result : 0.1
println( 6.3 % 3.1 );
println( 6.3 - int(6.3 / 3.1) * 3.1 );
```

Conditionals: if-statements

Conditionally execute a block of code.
Steps for creating an if-statement:

1. Set up the structure.

```
if  (                  )
{

}
```

2. Add a logical expression (evaluates to true or false).

```
if  ( mouseY > 250   )
{

}
```

3. Add block of code that runs when expression evaluates to true.

```
if  ( mouseY > 250   )
{
   rect( mouseX, mouseY, 50, 50);
}
```

Conditionals: if-else-statement

```
if ( boolean_expression ) {

   //statements executed when boolean_expression is true;

} else {

   //statements executed when boolean_expression is false;

}
```

```
// What does this do?
void draw() {
   if ( mouseY < 250 ) {
       println("the sky");
   } else {
       println("the ground");
   }
}
```

Conditionals: if-statements

```
if ( boolean_expression_1 ) {

      //statements;

} else if ( boolean_expression_2 ) {

      //statements;

} else if ( boolean_expression_3 ) {

      //statements;

} else {

      //statements;

}
```

Optional

Conditionals: If-statement examples

```
if (j < i) { … }

if (true) { … }

if (keyCode == 38) { … }

if (mouseX > 250 && mouseY > 250) { … }

if (speed > 100.0 && bMoving == false) { … }

if (speed > 100.0 && !bMoving) { … }

if (x < 10 || x > 20) { … }
```

Iteration: while-loop

Repeat a block of code while an expression continues to evaluate to true

```
while( continuation_test )
{
   statements;
   // continue;
   // break;
}
```

Iteration: while-loop

Steps for creating while-loop

1. Set up the structure.

```
while (                  )
{

}
```

2. Add a logical expression (evaluates to true or false).

```
while (  i < 10          )
{

}
```

3. Add block of code that continues to run while expression evaluates to true.

```
int i = 0;
while (  i < 10          )
{
   println( i );
   i = i + 1;
}
```

Iteration: for-loop

```
for ( initialization; continuation_test; increment )
{
   statements;
   // continue;
   // break;
}
```

- A kind of iteration construct
- initialization, continuation test and increment commands are part of statement
- To break out of a while loop, call **break;**
- To stop execution of statements in block and start again, call **continue;**

Iteration: while-loop <-> for-loop

```
float diameter = 500.0;
while ( diameter > 1.0 ) {
 ellipse( 250, 250, diameter, diameter);
 diameter = diameter - 10.0;
}
```

Initialize (runs only once)
Test to continue
Update

```
for (float diameter = 500.0; diameter > 1.0; diameter = 10.0 )
{
  ellipse( 250, 250, diameter, diameter);
}
```

Tracing while-loops and for-loops

```
for (int i=0; i<4; i++) {
  println( i );
}
```

Convert to a while-loop

```
int i = 3;
while ( i > 0 ) {
  println( i );
  i--;
}
```

Convert to a for-loop

Functions

- A function names a block of code, making it reusable.
- Arguments can be "passed in" to function and used in body.
- Arguments are a comma-delimited set of variable declarations.
- Argument values are ***copies*** of passed values, not originals.
- Function must return a value that matches function declaration.

```
return_type function_name( argument_decl_list )
{
   statements;
   [return value;]
}
```

Functions

Steps for declaring a function

1. Set up the structure.

```
addOne(                    )
{
}
```

2. Add argument variable declarations, if any.

```
addOne( int a              )
{
}
```

Functions (Cont'd)

Steps for declaring a function

3. Add block of code that runs when function is called.

```
addOne( int a              )
{
  int b;
  b = a + 1;
}
```

4. Add returned value (if any) and set function type to be the same.

```
int addOne( int a          )
{
  int b;
  b = a + 1;
  return b;
}
```

5. Test by calling it.

```
println( addOne(7) );
```

Declaring a Function vs. Calling a Function

```
void setup()
{
  size(500, 500);
  background(255);
}

void draw() { }

void mousePressed()
{
  float secret = secretFunction( mouseX, mouseY );
  fill(0);
  background(255);
  text(secret, mouseX, mouseY);
}

float secretFunction( float x, float y )
{
  float r, x2, y2;

  x2 = x - (0.5*width);
  y2 = y - (0.5*height);
  r = sqrt(x2*x2 + y2*y2);

  return r;
}
```

Names of passed variables do not have to match names of variables in function. Values are copied.

The single value returned by the function is preceded by the 'return' keyword.

A generic function to draw a happy face.

```
// Draw happy face
void happyFace( float x, float y, float diam )
{
  // Face
  fill(255, 255, 0);
  stroke(0);
  strokeWeight(2);
  ellipseMode(CENTER);
  ellipse(x, y, diam, diam );

  // Smile
  float startAng = 0.1*PI;
  float endAng = 0.9*PI;
  float smileDiam = 0.6*diam;
  arc(x, y, smileDiam, smileDiam, startAng, endAng);

  // Eyes
  float offset = 0.2*diam;
  float eyeDiam = 0.1*diam;
  fill(0);
  ellipse(x-offset, y-offset, eyeDiam, eyeDiam);
  ellipse(x+offset, y-offset, eyeDiam, eyeDiam);
}
```

Draw a happy face at mouse position when mouse or 'h' key is pressed.

```
void setup() {
  size(500, 500);
  background(0);
  smooth();
}

void draw() { }

// If mouse pressed, draw large happy face
void mousePressed() {
  float diam = random(30, 60);
  happyFace( mouseX, mouseY, diam );
}

// If h-key pressed, draw small happy face
void keyPressed() {
  if (key == 'h' || key == 'H') {
    float diam = random(10, 30);
    happyFace( mouseX, mouseY, diam );
  }
}
```

Scope

- An enclosing context in a program where values and expressions are associated.
- A way to separate variables in different parts of your program from one another.
- To a first approximation, the scope of a section of your code is demarcated by { and }.

Kinds of Scope

- Global
- Function
- Block

```
// Global scope.
// Variables declared here are accessible by all.
int x1 = 10;

void setup() {
  // Inside setup function scope.
  int x2 = 20;

  for (int i=0; i<10; i++) {
    // Inside the scope of the for-loop
    // 'x2' is accessible here
    // 'i' is not accessible outside the block
  }

  if ( x2 > 10 ) {
    String s = "blah";
    // Inside the scope of the if-statement
    // 's' is only accessible within this block
  }
}
```

Scope and Variable Access Rules

1. A variable declared within a given scope (global, function, block) is accessible (read, write) from within that scope, as well as all nested (inner) scopes.
2. A variable declared in an inner (nested) scope cannot be accessed from code executing in an outer scope or an adjacent scope.
3. When a variable name is accessed from code, the local scope is checked for the variable first. If it is not found, the next outer (containing) scope is checked for the variable. This continues until all outer scopes are searched, in order.

Shadowing

- If a variable is declared within an inner (nested) scope has the same name as a variable declared in an outer scope, the inner-scope variable "shadows" (hides) the outer-scope variable. The inner declared variable is a distinct variable with the same name. It does not replace the outer variable or change it in any way.

* Note: All rules apply to variables of any type: int, float, String, boolean, …

```
float x = 1.2;
void setup() {
  println(x);
}
```

```
float x = 1.2;
void setup() {
  float x=3.4;
  println(x);
}
```

```
float x = 1.2;
void setup() {
  float x=3.4;
  printIt();
  println(x);
}
void printIt() {
  println(x);
}
```

```
float x = 1.2;
void setup() {
  x=3.4;
  printIt();
  println(x);
}
void printIt() {
  println(x);
}
```

```
float x = 1.2;
void setup() {
  float x=3.4;
  printIt(x);
  println(x);
}
void printIt(float y) {
  println(x);
}
```