Review

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

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

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 (&&, ||, !)

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 "$"
- Variables can contain letters, digits, "_" and "$"

Syntax:

```java
int name = expression;
```

Example:

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

Comparing Declarations and Initializers

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

Primitive Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Range</th>
<th>Default</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>true, false</td>
<td>false</td>
<td>-</td>
</tr>
<tr>
<td>byte</td>
<td>{0..255}</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>int</td>
<td>{-2,147,483,648}</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>{-9,223,372,036,854,775,808}</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>{-3.402823478E38}</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>much larger/smaller</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>color</td>
<td>#00000000...#FFFFFF</td>
<td>black</td>
<td>4</td>
</tr>
<tr>
<td>char</td>
<td>a single character 'a', 'b',...</td>
<td>‘\u0000’</td>
<td>2</td>
</tr>
</tbody>
</table>
Operators
Symbols that operate on one or two sub-expressions.
Infix, prefix, or postfix

• Mathematical (\(+, -, *, /, \ldots\))
  - Perform standard mathematical operations (PEMDAS)

• Relational (\(<, >, ==, !=, \ldots\))
  - 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
\(+, -, *, /, \ldots\) and ... 
\(i++;\) equivalent to \(i = i + 1;\)
\(i += 2;\) equivalent to \(i = i + 2;\)
\(i -= 3;\) equivalent to \(i = i - 1;\)
\(i *= 2;\) equivalent to \(i = i * 2;\)
\(i /= 4;\) equivalent to \(i = i / 4;\)
\(i \% 3;\) the remainder after \(i\) is divided by 3 (modulo)

Examples:
\(1 + 2\)
\(i++\)

Relational Operators

\(<\) less than
\(>\) is greater than
\(\leq\) is less than or equal to
\(\geq\) is greater than or equal to
\(==\) is equivalent
\(!=\) is not equivalent

Examples:
\(10 \geq 10\)
\(mouseY > 250\)
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
    \(11/10 \rightarrow 1\)
    \(3/4 \rightarrow 0.75\)
  - Operands are promoted to the most general type
  - Modulo (\(\%\)) returns remainder after division
    \(11 \% 2 \rightarrow 1\)
    \(3 \% 4 \rightarrow 0\)
    \(5.1 \% 5 \rightarrow 0.1\) (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.
\[
\text{if} \left( \text{expression} \right) \{ \text{block of code} \}
\]
2. Add a logical expression (evaluates to true or false).
\[
\text{if} \left( \text{expression} \right) \{ \text{block of code} \}
\]
3. Add block of code that runs when expression evaluates to true.
\[
\text{if} \left( \text{expression} \right) \{ \text{block of code} \}
\]

Conditionals: if-else-statement
If-else statements allow for more than two possible outcomes.
Steps for creating an if-else-statement:
1. Set up the structure.
\[
\text{if} \left( \text{expression} \right) \{ \text{block of code} \}
\]
2. Add a logical expression (evaluates to true or false).
\[
\text{if} \left( \text{expression} \right) \{ \text{block of code} \}
\]
3. Add block of code that runs when expression evaluates to true.
\[
\text{if} \left( \text{expression} \right) \{ \text{block of code} \}
\]

Conditionals: if-statements
\[
\text{if} \left( \text{boolean_expression} \right) \{ \text{block of code} \}
\]
\[
\text{else if} \left( \text{boolean_expression} \right) \{ \text{block of code} \}
\]
\[
\text{else if} \left( \text{boolean_expression} \right) \{ \text{block of code} \}
\]
\[
\text{else} \{ \text{block of code} \}
\]

Iteration: while-loop
Repeat a block of code while an expression continues to evaluate to true.
Steps for creating while-loop:
1. Set up the structure.
\[
\text{while} \left( \text{continuation_test} \right) \{ \text{block of code} \}
\]
2. Add a logical expression (evaluates to true or false).
\[
\text{while} \left( \text{continuation_test} \right) \{ \text{block of code} \}
\]
3. Add block of code that continues to run while expression evaluates to true.
\[
\text{while} \left( \text{continuation_test} \right) \{ \text{block of code} \}
\]
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

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

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.

Functions (Cont’d)

Steps for declaring a function
1. Set up the structure.
   ```java
   returnType functionName(argument_list) {
    statements;
    return value;
   }
   ```
2. Add argument variable declarations, if any.
   ```java
   returnType functionName(int a, int b) {
   int c = a + b;
   return c;
   }
   ```
Declaring a Function vs. Calling a Function

void setup() {
  size(500, 500);
}
void draw() {
  float secret = secretFunction(mouseX, mouseY);
  // Other code...
}
void mousePressed() {
  float secret = secretFunction(mouseX, mouseY);
  float x2, y2;
  x2 = mouseX - width / 2;
  y2 = mouseY - height / 2;
  float r = sqrt(x2 * x2 + y2 * y2);
  // Other code...
}

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

A generic function to draw a happy face.

// Draw happy face
void happyFace(float x, float y, float diam) {
  // Draw happy face
  fill(255, 255, 0);
  stroke(0);
  strokeWeight(2);
  // Other code...
}

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

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, ...

```java
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;
  println(x);
  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);
}
```