Design Programming

DECO1012 & DECO2011

Rob Saunders
Decisions, Repetition
Data
What is Data?

- Data is information we record about things
  - Sometimes data is a measurement
  - Sometimes data is used to identify

- Look at your student card, look at the data it records about you, e.g.,
  - Photo: a measurement of how you look
  - Student ID: a unique number assigned to you
Data Types

• Processing can store different types of data including:
  • Boolean values (boolean)
  • numbers (int, float)
  • colours (color)
  • letters (char)
  • words (String)
  • fonts (PFont)
  • images (PImage)
## Primitive Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>truth values</td>
<td>true, false</td>
</tr>
<tr>
<td>char</td>
<td>text characters</td>
<td>a-z, A-Z, 0-9, etc.</td>
</tr>
<tr>
<td>byte</td>
<td>very small whole numbers</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>short</td>
<td>small whole numbers</td>
<td>-32768 to 32767</td>
</tr>
<tr>
<td>int</td>
<td>whole numbers</td>
<td>-2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>long</td>
<td>big whole numbers</td>
<td>-263 to 263-1</td>
</tr>
<tr>
<td>float</td>
<td>real numbers</td>
<td>1.4e-45 to 3.4e+38</td>
</tr>
<tr>
<td>double</td>
<td>high-precision real numbers</td>
<td>4.9e-324 to 1.8e+308</td>
</tr>
</tbody>
</table>
Variables

- A variable is a container for storing data, every variable has two parts:
  - Name: identifies the variable
  - Value: value stored in the variable

- Variables allow data to be reused
  - For example, if the number \textbf{21} is stored in a variable called \textit{age}, then every time the word \textit{age} is used in a program it will be represent (and be replaced with) the number \textbf{21}
Variables and Computer Memory

- Think of computer memory like a set of pigeon holes:
  - Each pigeon hole is an address in memory
  - Papers stored in a pigeon hole is data
- Using a variable is like putting a label on the front of a pigeon hole
Declaring Variables

- In Processing, variables must be declared before they are used

- Declaring a variable tells the computer the type of data that will be stored in the variable, and the variable name

```java
int x;      // Declare the variable x of type int
float y;    // Declare the variable y of type float
boolean b;  // Declare the variable b of type boolean
x = 50;     // Assign the value 50 to x
y = 12.6;   // Assign the value 12.6 to y
b = true;   // Assign the value true to b
```
Declaring Variables

- As a shortcut, variables can be declared and assigned a value at the same time:

  ```java
  int x = 50;
  float y = 12.6;
  boolean b = true;
  ```

- Alternatively, multiple variables can be declared on a single line:

  ```java
  float x, y, z;
  x = -3.9;
  y = 10.1;
  z = 124.23;
  ```
Naming Variables

- In Processing, there are some restrictions on variable names:
  - Variable names can only include a limited range of characters: a-z, A-Z, 0-9, _
  - Variable names cannot contain spaces, the underscore character “_” is often used instead of a space
  - Variable names must not start with a number, although they can contain numbers in the middle and at the end
Naming Conventions

- In Java programs, variable names typically begin with a lowercase letter and use a capital letter to start each new word
  - e.g., width, rect4, fillColour, isFilled
- Variables that refer to constant data typically use all uppercase letters and underscores between each word
  - e.g., PI, RED, MAX_VALUE
Choosing Good Variable Names

- Variables should have names that describe their content.
- This makes programs easier to read, reducing the need for lengthy comments and the potential for making mistakes.
Variable Naming Example

- Imagine you need to create a variable to store room temperature, which of the following names would you choose?

  t
  temp
  temperature
  roomTemp
  room Temperature
Built-In Variables in Processing

- Processing has built-in variables for storing important and commonly used data
  - e.g., the variables width and height store the width and height of the display window

```java
// Prints "100, 100" to the console
println(width + "", " + height);

size(300, 400);
// Prints "300, 400" to the console
println(width + "", " + height);
```
Using Built-In Variables

- Using \texttt{width} and \texttt{height} we can write sketches that scale to the size of the display window.

```plaintext
size(300, 500);
ellipse(width*0.5, height*0.5, width*0.66, height*0.66);
line(width*0.5, 0, width*0.5, height);
line(0, height*0.5, width, height*0.5);
```
Arithmetic and Functions
Arithmetic

- Almost everything that the computer does is reduced at some point to numbers
  - All of the drawing functions take numbers to define the size and colour

- We can use arithmetic to control the creation of sketches
  - The arithmetic doesn’t have to be difficult, it can be as simple as a little addition, subtraction and multiplication
int $a = 30$;
line($a$, 0, $a$, $height$);
$a = a + 40$;
strokeWeight(4);
line($a$, 0, $a$, $height$);

int $a = 8$;
int $b = 10$;
line($a$, 0, $a$, $height$);
line($b$, 0, $b$, $height$);
strokeWeight(4);
line($a*b$, 0, $a*b$, $height$);
Drawing with Arithmetic

```c
int y = 20;
line(0, y, width, y);
y = y + 6;
line(0, y, width, y);
y = y + 6;
line(0, y, width, y);
y = y + 6;
line(0, y, width, y);
```

```c
float y = 20;
line(0, y, width, y);
y = y * 1.6;
line(0, y, width, y);
y = y * 1.6;
line(0, y, width, y);
y = y * 1.6;
line(0, y, width, y);
```
## Arithmetic Operators

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>add</td>
<td>3 + 4</td>
</tr>
<tr>
<td>-</td>
<td>subtract</td>
<td>6 - 5</td>
</tr>
<tr>
<td>*</td>
<td>multiply</td>
<td>4 * 5</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
<td>10 / 2</td>
</tr>
<tr>
<td>%</td>
<td>modulus</td>
<td>9 % 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>%</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Data Types and Arithmetic Operators

- Combining two integers will always result in an `int`, combining two floating-point numbers will always result in a `float`, but when an `int` and a `float` are combined the result is a `float`.

```java
println(4/3);       // Prints "1"
println(4.0/3);     // Prints "1.3333334"
println(4/3.0);     // Prints "1.3333334"
println(4.0/3.0);   // Prints "1.3333334"
```
Be Careful Assigning Values

- Integer values can be assigned to floating-point variables, but floating-point values cannot be assigned to integer variables.

- NOTE: The calculation is completed as an int before converting it to a float.

```cpp
int a = 4/3; // Assign 1 to a
int b = 3/4; // Assign 0 to b
int c = 4.0/3; // ERROR!
int d = 4.0/3.0; // ERROR!
float e = 4.0/3; // Assign 1.3333334 to e
float f = 4.0/3.0; // Assign 1.3333334 to f
```
Operator Precedence

- The order that arithmetic operators are applied is based on their precedence
- Highest to lowest: % / * + - =

\[ x = 3 + 4 \times 5; \quad \text{// Assign 23 to } x \]
\[ y = (3 + 4) \times 5; \quad \text{// Assign 35 to } y \]
Increments and Decrements

- Often we just want to add 1 to a value or subtract 1 from a value
- We can do this easily with the increment (++) and decrement (--) operators

```java
int x = 1;
println(x); // Prints "1" to the console
x++;       // Equivalent to x = x + 1
println(x); // Prints "2" to the console
x--;       // Equivalent to x = x - 1
println(x); // Prints "1" to the console
```
Combination Operators

To add, subtract, multiply or divide a value stored in a variable and store the result in the same variable we can use the combined operator assignments

```java
int x = 1;
println(x); // Prints "1" to the console
x += 5;     // Equivalent to x = x + 5
println(x); // Prints "6" to the console
x -= 10;    // Equivalent to x = x - 10
println(x); // Prints "-4" to the console
```
Constraining Numbers

- Processing provides a set of functions for constraining numbers
- `ceil()`, `floor()`, `round()`, `min()`, `max()`
- These functions are different from `line()` or `ellipse()` because they return values that can be assigned to a variable, or used in an arithmetic expression
Rounding Up

- The `ceil()` function rounds any floating-point number up to the nearest whole number.

```c
int w = ceil(2.0);  // Assign 2 to w
int x = ceil(2.1);  // Assign 3 to x
int y = ceil(2.5);  // Assign 3 to y
int z = ceil(2.9);  // Assign 3 to z
```
Rounding Down

• The `floor()` function rounds any floating-point number down to the nearest whole number

• This is the default method of rounding when converting between `float` and `int`

```c
int w = floor(2.0);  // Assign 2 to w
int x = floor(2.1);  // Assign 2 to x
int y = floor(2.5);  // Assign 2 to y
int z = floor(2.9);  // Assign 2 to z
```
Rounding

• The **round()** function rounds any floating-point number to the nearest whole number

• Values with a decimal greater than .5 will be rounded up, less than .5 will be rounded down

```c
int w = round(2.0);  // Assign 2 to w
int x = round(2.1);  // Assign 2 to x
int y = round(2.5);  // Assign 3 to y
int z = round(2.9);  // Assign 3 to z
```
Minimum and Maximum

- The `min()` function returns the smallest value of two or three numbers.
- The `max()` function returns the largest value of two or three numbers.

```cpp
int u = min(5, 9); // Assign 5 to u
int v = min(-4, -12, -9); // Assign -12 to v
float w = min(12.3, 230.24); // Assign 12.3 to w
int x = max(5, 9); // Assign 9 to x
int y = max(-4, -12, -9); // Assign -4 to y
float z = max(12.3, 230.24); // Assign 230.24 to z
```
Decisions
Relational Expressions

- Relational expressions return a Boolean value, i.e., they are either true or false

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 &gt; 5</td>
<td>false</td>
</tr>
<tr>
<td>3 &lt; 5</td>
<td>true</td>
</tr>
<tr>
<td>5 &lt; 3</td>
<td>false</td>
</tr>
<tr>
<td>5 &gt; 3</td>
<td>true</td>
</tr>
</tbody>
</table>
Operators used in relational expressions compare two values to calculate a relation between them, e.g., is one greater than another.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>==</td>
<td>equivalent to</td>
</tr>
<tr>
<td>!=</td>
<td>not equivalent to</td>
</tr>
</tbody>
</table>
println(3 > 5);  // Prints "false"
println(5 > 3);  // Prints "true"
println(5 > 5);  // Prints "false"
println(3 < 5);  // Prints "true"
println(5 < 3);  // Prints "false"
println(5 < 5);  // Prints "false"
println(3 >= 5);  // Prints "false"
println(5 >= 3);  // Prints "true"
println(5 >= 5);  // Prints "true"
println(3 <= 5);  // Prints "true"
println(5 <= 3);  // Prints "false"
println(5 <= 5);  // Prints "true"
println(3 == 5); // Prints "false"
println(5 == 3); // Prints "false"
println(5 == 5); // Prints "true"
println(3 != 5); // Prints "true"
println(5 != 3); // Prints "true"
println(5 != 5); // Prints "false"
Conditionals

- Conditionals allow a program to make decisions about which lines of code.
  - They let actions take place only when a specific condition is met.
- Conditionals allow a program to behave differently depending on the values of variables.
  - For example, the program may draw a line or an ellipse depending on the value of a variable.
Conditionals

- The if structure is most commonly used in Processing to implement conditionals:

```java
if (test) {
    statements
}
```

- The test must be an expression that resolves to true or false.

- When test evaluates to true, the statements inside the { (left brace) and } (right brace) are run.
Example Conditional

```plaintext
if (x > 100) {
    ellipse(50, 50, 36, 36);
}

if (x < 100) {
    rect(35, 35, 30, 30);
}

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

---

```
x = 150
```

```
x = 50
```
Embedding Conditionals

Conditional statements can be embedded inside other conditional statements

```java
if (x > 100) {
    // First test determines ellipse/line
    if (x < 300) {
        // Second test determines which to draw
        ellipse(50, 50, 36, 36);
    } else {
        line(50, 0, 50, 100);
    }
} else {
    rect(33, 33, 34, 34);
}
```
Extending Conditionals

- To run some code when the result of a test is false, use the `else` keyword

```javascript
if (x > 100) { // If x is greater than 100,
    ellipse(50,50,36,36); // draw this ellipse
} else { // Otherwise,
    rect(33, 33,34,34); // draw this rectangle
}

line(20, 20, 80, 80); // Always draw the line
```
Logical Operators

- Boolean values can be combined or inverted by using logical operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>NOT</td>
</tr>
</tbody>
</table>
## Logical Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>true &amp;&amp; true</td>
<td>true</td>
</tr>
<tr>
<td>true &amp;&amp; false</td>
<td>false</td>
</tr>
<tr>
<td>false &amp;&amp; false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td></td>
</tr>
<tr>
<td>!true</td>
<td>false</td>
</tr>
<tr>
<td>!false</td>
<td>true</td>
</tr>
</tbody>
</table>
The Logical OR Operator

```java
int a = 10;
int b = 20;

if ((a > 5) || (b < 30)) {
    line(20, 50, 80, 50);
}

if ((a > 15) || (b < 30)) {
    ellipse(50, 50, 36, 36);
}
```
int a = 10;
int b = 20;

if ((a > 5) && (b < 30)) {
    line(20, 50, 80, 50);
}

if ((a > 15) && (b < 30)) {
    ellipse(50,50,36,36);
}
Repetition
Iteration

Iteration can be used to greatly reduce the amount of code required to accomplish repetitive tasks.

Original code

```plaintext
size(200, 200);
line(20, 20, 20, 180);
line(30, 20, 30, 180);
line(40, 20, 40, 180);
line(50, 20, 50, 180);
line(60, 20, 60, 180);
line(70, 20, 70, 180);
line(80, 20, 80, 180);
line(90, 20, 90, 180);
line(100, 20, 100, 180);
line(110, 20, 110, 180);
line(120, 20, 120, 180);
line(130, 20, 130, 180);
line(140, 20, 140, 180);
```

Code using iteration

```plaintext
size(200, 200);
for (int i = 0; i < 150; i += 10) {
    line(i, 20, i, 180);
}
```
Iteration using `while` loops

- The `while` structure is very much like the `if` structure, except that the statements inside the brackets are executed again and again until the test becomes `false`.

```plaintext
while (test) {
    statements
}
```
A Simple while Loop

- In this example the statements inside the curly brackets will be executed as long as the value of x is less than 10.

- Notice that if the value of x did not increase inside the while loop, it would run forever.

```java
int x = 0;
while (x < 10) {
    println(x);
    x++;
}
```
Using **do...while** Loops

- A do...while loop is similar to a while loop, except that the test is performed after the code inside the brackets is run.
- Consequently, the code inside the brackets is always run at least once.

```java
int x = 10;
do {
    println(x);
    x--;  
    x--;  
} while (x > 1)
```
Iteration using **for** loops

- Often we need to initialise, test and update a variable to complete a loop
- The **for** structure allows these tasks to be put together and is structured like this:

```java
for (init; test; update) {
    statements
}
```

// An example for loop
for (int x = 0; x < 10; x++) {
    println(x);
}
An Example for Loop

Using a for loop we can draw repetitive patterns

```java
size(200, 200);
int y = 20;
for (int x = 20; x < width; x += 20) {
    rect(x-5, y-5, 10, 10);
}
```
An Example for Loop

Using embedded for loops we can draw repetitive patterns

```java
size (200, 200);
for (int y = 20; y < height; y += 20) {
    for (int x = 20; x < width; x += 20) {
        rect(x-5, y-5, 10, 10);
    }
}
```
Using Variables Inside Loops

```java
fill(0,76);
noStroke();
smooth();
for (int y = -10; y <= 100; y += 10) {
  for (int x = -10; x <= 100; x += 10) {
    ellipse(x + y/8.0, y + x/8.0, 15 + x/2, 10);
  }
}
```
Using Conditionals Inside Loops

```c
for (int y = 20; y <= 80; y += 5) {
    for (int x = 20; x <= 80; x += 5) {
        if ((x % 10) == 0) {
            line(x, y, x+3, y-3);
        } else {
            line(x, y, x+3, y+3);
        }
    }
}
```
Vertices
Vertices

- Vertices are points used to define shapes
- Vertices allow programmers to create complex shapes using many points

- Shapes are defined by putting a number of calls to the `vertex()` function between calls to the functions `beginShape()` and `endShape()`
Simple Shapes

```javascript
noFill();
beginShape();
vertex(30, 20);
vertex(85, 20);
vertex(85, 75);
vertex(30, 75);
endShape();
```

```javascript
noFill();
beginShape();
vertex(30, 20);
vertex(85, 20);
vertex(85, 75);
vertex(30, 75);
endShape(CLOSE);
```
Creating Vertices Using Loops

We can use loops to create complex shapes with vertices

```python
noStroke();
fill(0);
beginShape();
vertex(40, 10);
for (int i = 20; i <= 100; i += 5) {
    vertex(20, i);
    vertex(30, i);
}
vertex(40, 100);
endShape();
```
Beziers Vertices

We can use Bezier vertices to create complex curved shapes

smooth();
noStroke();
beginShape();
vertex(90, 39);
bezierVertex(90, 39, 54, 17, 26, 83);
bezierVertex(26, 83, 90, 107, 90, 39);
endShape();
Colour
Setting Colours

- Colours are set using the **background()**, **fill()** and **stroke()** functions

  ```
  background(value1, value2, value3)
  fill(value1, value2, value3)
  fill(value1, value2, value3, alpha)
  stroke(value1, value2, value3)
  stroke(value1, value2, value3, alpha)
  ```
Setting Colours

- By default, colours are defined using the RGB (red, green, blue) colour space
  - `value1` is the red component
  - `value2` is the green component
  - `value3` is the blue component
  - If provided, `alpha` is always the opacity value
- The meaning of the values can be changed with the `colorMode()` function
Setting the Background

background(242, 204, 47);

\background(174, 221, 60);
Filling with Colour

background(255);
noStroke();
smooth();
fill(242, 204, 47, 160);  // Yellow
ellipse(47, 36, 64, 64);
fill(174, 221, 60, 160);  // Green
ellipse(90, 47, 64, 64);
fill(116, 193, 206, 160); // Blue
ellipse(57, 79, 64, 64);
Colour Selector

- Processing provides a colour selector to help you choose your colours
Colour Data

- Processing provides the `color` data type to store colour values in variables
- Colour values are created using the `color()` function

```plaintext
color(gray)
color(gray, alpha)
color(value1, value2, value3)
color(value1, value2, value3, alpha)
```
Using **color** values

color ruby = color(211, 24, 24, 160);
color pink = color(237, 159, 176);
background(pink);
noStroke();
fill(ruby);
rect(35, 0, 20, 100);
Setting the Colour Space

- The `colorMode()` function sets the colour space for a program
  - The mode used can either be RGB or HSB
    - HSB = hue, saturation, brightness

```
colorMode(mode)
colorMode(mode, range)
colorMode(mode, range1, range2, range3)
```
RGB vs HSB

colorMode(RGB);
for (int i = 0; i < 100; i++) {
    stroke(i*2.5, 255, 255);
    line(i, 0, i, 100);
}

colorMode(HSB);
for (int i = 0; i < 100; i++) {
    stroke(i*2.5, 255, 255);
    line(i, 0, i, 100);
}
Lab Exercises
Lab Exercises (Variables)

- Create a composition that scales with different window sizes.
  - Put different values into `size()` to test.

- Use a single variable to set the position and size for three ellipses.

- Create a few relational expressions.
  - Print evaluations to the console with `println()`.
  - Combine relational expressions using `&&` and `||`
Lab Exercises (Conditionals)

- Create a composition with a series of lines and ellipses that changes depending on the size of the display window.
- Use an if structure to select which lines of code to run based on width and height. Use the size() function to test the code.
- Try adding an else to the code to change which code is run.
- Try adding some logical operators to change which code is run.
Lab Exercises (Vertices)

› Draw a shape of your own design using a `for` loop to create the vertices.

› Use different parameters for `beginShape()` to change the way a series of vertices are drawn.

› Draw a complex curved shape of your own design using `bezierVertex()`.
Lab Exercises (Colour)

- Use HSB colour space and a for structure to design a gradient between two colours.
Assignment 1

Percentage of Final Mark: 20%
Static Variations

- Design and develop a static sketch that can generate different images each time it is run
  - Use if conditions to change the behaviour of the sketch based on either:
    - `random()` or `noise()`
    - `year()`, `month()`, `day()`, `hour()`, `minute()`, and `second()`
  - Sketch should be 400x400 pixels in size
Submission

- Submission will be via an online portfolio, that will include
  - A series of experimental sketches showing how you developed your idea—include at least 3 experimental sketches. **THIS IS IMPORTANT!!!**
  - All source code should be commented
  - The final sketch that you developed further based upon your experimental sketches
  - A description of your sketch including references to inspirational work by others