Fundamental Concepts

ISE 108: Introduction to Programming
Stony Brook University

Outline of Topics

- More useful commands
- Conditional statements (chapter 5)
- Loops (chapter 6)
- Functions (chapter 7)

More Commands

Adding Interaction

- void mousePressed ()
  - If you define this function, it will be called whenever the user clicks the mouse button
- void keyPressed ()
  - If you define this function, it will be called whenever the user hits a key on the keyboard

Useful Functions

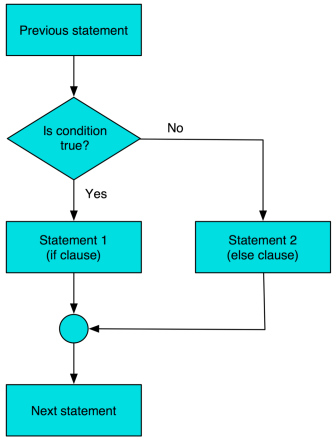
- println(*expression*)
  - Prints *expression* to the screen
  - ex. println("Hello"); / / prints "Hello" (no quotes)
- random(*range_size*)
  - Returns a random number between 0 and (*range_size* - 1)
  - ex. random(4) returns a number between 0 and 4

Setting Limits

- constrain (*variable*, *min_value*, *max_value*)
  - Restricts the value of *variable* to fall between *min_value* and *max_value*
  - constrain() returns the limited value
- frameRate (*rate*)
  - Controls the rate at which draw() redraws the screen
  - The default frame rate is 60 frames per second

Conditionals

Flow of Control



Making Decisions

- Many programs must behave differently when presented with different situations
  - e.g., choosing an action from a menu, determining what sales tax rate to charge, etc.

True and False Values

- A *boolean* expression is one whose value is either true or false (but not both!)
  - Ex. a > b or var1 == var2
- Boolean (logical) equality operator: ==
- Common programming error: using '=' (assignment) instead of "==" (equality)
  - Ex. `if (x = 5)`

Boolean Operations

- Processing makes decisions using boolean logic (i.e., values of true and false) and operations
- Two special keywords (`true` and `false`) are used for this purpose
- Relational operators are used to compare two values
- Logical operators combine smaller boolean expressions into a single, larger expression

Relational Operators

Operator	Meaning	Example
<	Less than	`age < 30`
>	Greater than	`height > 6.2`
<=	Less than or equal to	`taxable <= 20000`
>=	Greater than/equal to	`temp >= 98.6`
==	Equal to	`grade == 100`
!=	Not equal to	`number != 250`

Logical Operators

- && (logical AND)
  - true only if BOTH operands are true
- || (logical OR)
  - true if AT LEAST ONE operand is true (inclusive OR)
- ! (logical NOT)
  - true if operand is false and vice versa

The `if` Statement

- General form:

```
if ( condition )
  statement (or block of statements) to be
  executed if condition is true
```

- Ex.

```
if (length < 2)
  println("Too short!\n");
```

Examples

```
if (age < 16)
   println("Too young to drive!\n");
```

```
if (password.equals("foo"))
   accessGranted = true;
```

The `if-else` Statement

- Select one of two possible execution paths, based on the result of a comparison
- General format:

```
if ( expression )
  statement block 1
else
  statement block 2
```

More on if Statements

- An if statement executes its body when (and only when) the condition is true
  - If the condition is false, the body is skipped, and execution picks up at the first statement after the if
- By default, the body of an if statement is restricted to the first statement that follows the "if (condition)" line
  - Indentation doesn't matter
  - This can lead to trouble...

Empty Statements

- A semicolon by itself is a valid (but non-functional) statement
- Common mistake: putting a semicolon immediately after an `if` statement:

```
if (x > 5);
  println("x greater than 5!");
```

- With the semicolon, the print statement will execute regardless of the value of x

Compound Statements

- if and else only execute a single following statement
- We can get around this by enclosing multiple statements in braces
- The resulting block is called a *compound statement*
- Style suggestion: always use braces around the body of an `if` or `else` clause

Compound Statements

```
if (hours > 40)
{
   hours = hours - 40;
   overtimePay = hours * 12.0;
   totalPay = (40 * 8.0) + overtimePay;
}
```

Nested `if` Statements

- A statement block may contain another `if` statement
- Ex.

```
if (income > 25000)
  if (deductions < 3500)
    tax_rate = 1.035;
```

`if-else` Chains

```
if (expression_1)
  statement_1
else if (expression_2)
  statement_2
else
  statement_3
```

Loops

Iterative Programming

- Many programs perform the same task many times
  - Operations are repeated on different data
- Ex. Adding a list of numbers
- Ex. Displaying frames of a movie file
- Repetitive tasks are specified using loops

Loop Elements

All loop constructs share four basic elements:

1. Initialization
2. Testing the loop condition
3. Loop body (the task to be repeated)
4. Loop update

Initialization

- This section of code is used to set starting values
- For example, setting a total to 0 initially
- This can be done as part of the loop, or separately before the loop code begins

Loop Tests

- Test expressions are used to determine whether the loop should execute (again)
- Tests compare one value/variable with another
- If the test evaluates to TRUE, then the loop will execute another time

Loop Update

- This step changes the value(s) of the loop variable(s) before the loop repeats
- Ex. moving to the next item to process
- This can be done explicitly as part of the loop, or it can be done inside the loop body

`while` Loops

- `while` loops execute as long as the test condition is true
- Order of execution:
  1. Initialization
  2. Loop condition test
  3. Loop body
  4. Loop update (then return to 2)

General Form

```
initialization
while ( loop condition test )
{
  loop body
  loop update
}
```

`while` Loop Example

```
int countDown = 5;
while (countDown >= 0)
{
  println(countDown + "...");
  countDown = countDown - 1;
}
```

Loop Breakdown

```
int countDown = 5;                  Loop initialization
while (countDown >= 0)              Loop test
{
  println(countDown + "...");       Loop body
  countDown = countDown - 1;        Loop update
}
```

Loop Output

5...
4...
3...
2...
1...
0...

Another Example

```
int root = 0;

while (root < 10)
{

  root = root + 1;

  println( root + " * " + root + " = "
           + (root * root) );

}
```

Loop Output

1 * 1 = 1
2 * 2 = 4
3 * 3 = 9
4 * 4 = 16
5 * 5 = 25
6 * 6 = 36
7 * 7 = 49
8 * 8 = 64
9 * 9 = 81
10 * 10 = 100

`for` Loops

- `for` loops execute a fixed number of times
- Order of execution:
  1. Initialization
  2. Loop condition test
  3. Loop body
  4. Loop update (then return to 2)

General Form

```
for ( initialization ;
      loop condition test ;
      loop update )
{
  loop body
}
```

`for` Loop Example

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

Loop Breakdown

Loop initialization

Loop test

Loop update

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

Loop body

Loop Output

```
0
1
2
3
4
5
6
7
8
9
```

Aside: Math Shorthand

- Processing (like Java) provides "shorthand" for common arithmetic expressions
- Any expression of the form *x = x operator value* can be rewritten as *x operator = value*
  - ex. i = i + 3 can be rewritten as i += 3
- "++" after a variable means "Add 1 to the variable"
- "--" is the same, except it subtracts 1
  - ex. i++ is the same as i = i + 1

Fancier `for` Loop Headers

- We can include multiple initialization or update statements in a loop header
  - Separate each statement with a comma
  - Ex. for (i = 0, j = 1; i < 5; i++, j--) { }
- Loop update statements don't have to increase by 1
  - Ex. for (i = 1; i < 101; i = i + 5) { }

Advanced `for` Loop Headers

- Loop headers can also include calls to methods
  - Ex. for (int i = 0; i < numPlayers() ; i++)
- A for loop can also omit part of the header
  - The missing piece must be supplied somewhere else
  - Ex. for (; i < 5; i++) { }
  - Pathological example: for (; ;) { }

Choosing a Loop Type

- For a fixed number of iterations:
  - `for` loops are generally considered the way to go
- For a variable number of iterations:
  - `while` loops can execute 0 or more times
- However, each type of loop can be rewritten as the other type

Nested Loops

- The body of a loop can contain any other type of statement(s)
  - This includes other loops
- If the outer loop executes ***n*** times, and the inner loop executes ***m*** times, the body of the inner loop will execute (***n*** × ***m***) times

Nested Loop Example

```
for (int i = 0; i < 4; i++)
{
   for (int j = 0; j < 4; j++)
   {
      // print() is like println(), except it doesn't
      // go to the next line at the end
      print("*");
   }
   println(); // go to next line for next row
}
```

Example Output

```
****
****
****
****
```

Another Example

```
int numStars;

for (numStars = 1; numStars < 11; numStars++)
{

  for (int i = 0; i < numStars; i++)

    print("*"); // part of inner loop

  println(); // part of outer loop

}
```

Loop Output

```
*
**
***
****
*****
******
*******
********
*********
**********
```

Functions

What is a Function?

- Remember that laziness is a virtue (for programmers)
  - Always try to avoid writing duplicate code
- A function (sometimes called a *method*) lets us group related statements together so that they can be reused
  - Synonyms: procedure, or subprogram
- Calling a method tells Processing to execute that code

Calling Functions

- To call (invoke) a function, write its name, followed by left and right parentheses
  - The function's input (its *arguments*) should be placed inside the parentheses
  - The parentheses are always needed, even if there is no input
  - Function calls are terminated by a semicolon

Defining a Function

- A function definition consists of a *header* and the function body
- The header specifies the function's return type, name, and parameter list (in that order)
  - Ex. int doSomething (int value)
- The body is a brace-enclosed set of 0 or more program statements

All About Arguments

- *Arguments*: pieces of data that are passed to a function
  - Inside the function, they are called *parameters*
- Different input can produce different results
- Parameters can be manipulated, like variables
- Primitive type arguments (int, double, boolean) are passed as copies — changes are not sent back.

Function Example

```
int multiply (int first, int second) // header
{
   return (first * second);          // body
}
// Return type: int
// Function name: multiply
// Arguments: two int variables
```

Return Types

- A function may *return* a value to whomever called it
  - This can be any type of value (int, double, etc.)
  - General form: `return` *value* `;`
  - If a function returns a value, that value must be the same type as what was declared in the header
  - NOTE: a function ends IMMEDIATELY after executing a return statement

Return Type Example

```
int timesThree (int value) // return type: int
{
   // the type of the value that is returned must match
   // the type that was declared in the function header
   return (value * 3);
}
```

A Second Example

```
int nextEvenMultiple (int i, int j)
{
   return i + j - i % j;
}
```

`void` Functions

- A function can also be *void*
  - This means that the function does not return any value to its caller
  - A void function has `void` as its return type
  - void functions may have *side effects*
    - This means that the function does something visible (like printing something to the screen), but does not explicitly return any value

`void` Function Example

```
void printLineOfStars()
{
   println("**********"); // side effect: prints to the screen
}
// NOTE: void functions don't need a return statement
```