1 **Processing: Functions**

CST112

2 **Structuring Programs (Modularity)**

- Programmers often divide large applications into several modules within the program
- This is necessary due to the size and complexity of *professional* programs
- Hierarchies of structure allow the design to begin at the most *general* level …
- And work to the more *specific*

3 **To Find an Office in a Building**

- A city
  - has many districts or suburbs
    - each of which has many streets
      - each of which has many buildings
        - each of which has many offices

4 **Top-Down Program Structure**

- The problem
  - has some major tasks
    - each of which has many subtasks
      - each of which has many subtasks
        … and so on down as needed

6 **Structure Using Functions (Page 1)**

- Complete applications in programming consist of a *hierarchical* collection of functions
- *Working together* they perform the entire programming task
- Most complex tasks can be subdivided into procedures in this way ("divide and conquer")

7 **Structure Using Functions (Page 2)**

- Functions are written by a programmer to define specific tasks within the larger task…
  - Each task should be *clearly identifiable*
- Statements defining the functions are written once in the application and may be called from *more than one location* in the program (reusability)
- Statements and variables are *hidden* from other functions in the program (information hiding)

8 **Advantages of Using Functions**

- Pre-written *built-in* functions (part of the Processing language/library) simplify program development
  - E.g. line(), rect(), ellipse(), etc
- Each small piece can be *written and tested separately*
- *Smaller* functions usually are easier to understand
- In large professional systems individual programmers and programmer *teams* can build and share functions

9 **The Function Call (Page 1)**

- The function is *invoked* by a function call
  - The syntax specifies the function's name
  - *Transfers control* to the location of the function in the program and *executes* it
  - Optionally provides the information/input (called the arguments) that the function needs for execution

10 **The Function Call (Page 2)**

- Format:
  functionName([*argument1*, *argument2*, …]);
  - The function call will transfer control of the execution of the program to the location of the *functionName*
- Examples:
  stickMan();
  circle(x, y, radius);

11 **The Function Definition (Page 1)**

- A program function is defined by a header (which names the function—also is called its signature) and the body enclosed in {braces}
  - The format is similar to setup() or draw()

12 **The Function Definition (Page 2)**

- Simple format:

```
void functionName()
{
    variableDeclarations/statements …
}
```

- Complex format:

```
type/void functionName( [type parameter1, type parameter2, … ] )
{
    variableDeclarations/statements …
}
```

13 **The Function Definition (Page 3)**

- Examples:

```
void setup()
{
    …
}
void stickMan()
{
  …
}
```

14 **Function Control**

1. The function call *transfers control* to the named sub function
2. The body of that function is executed *entirely*
3. At the conclusion of execution of the sub function, control *returns* to point in the initial function at which the sub function was called

30 **Passing Arguments**

- In many function calls, there may be a value or values inside the parentheses (arguments) that are passed to the called function (data "sharing")
  - True for "built-in" Processing function calls as well as those defined by programmers
- Format:
  functionName(*parameter1*[, *parameter2*, ...]);
- Examples:
  rect(x, y, w, l);
  circle(x, y, diameter);

31 **The Parameter List (Page 1)**

- The parameter list is a comma-separated list of *variable declarations* inside the function header (signature)
  - The textbook calls this the "argument list"
  - However for clarity and consistency with other languages, we will use the term parameter list
- Variables *receive* the values passed from the parameters in the function call

32 **The Parameter List (Page 2)**

- Format:
  void/*type functionName*([*type parameter1*, *type parameter2*, ...])
  - Each *parameter* in the list is a declared variable, declared separately with its own type (even if two or more parameters are the same type)
- Examples:
  void circle(int x, int y, int diameter)
  void stickMan(int xCoord, int yCoord)

33 **The Parameter List (Page 3)**

- Parameters are *local* variables:
  - A means for communicating information *between* the function call and the function itself
  - When the function is called, the argument values are passed and assigned to the parameter variables in the function header

34 **The Parameter List (Page 4)**

- The number of arguments in the function call must be the same as the number of parameter variables in the called function header
- The variable *type* of each parameter in the function header must match that of the

argument value passed to it (and in the same order)

36 **Function Call and Definition Examples**

- Function calls:

```
circle(75, 125, 50);
circle(x, y, r);
circle(mouseX, mouseY, diameter);
```

- Function header (signature) and body

```
void circle(int x, int y, int diameter)
{
   ellipse(x, y, diameter, diameter);
}
```

50 **Passing Arguments "ByValue"**

- Passing an argument by value means that a *copy* of the argument is stored as private in the called function at a separate RAM address …
  - Changing the value of the variable in the called function does not change the original value
- The opposite of passing by value is passing by reference which means that the address of the values is passed to the called function …
  - Changing the value of the variable in the called function *will change* the original value

51 **Return Values (Page 1)**

- Programmer-defined functions:
  1. May have arguments passed to them (the *input*)
  2. Process that data (the *code*)
  3. May have return values (the *output* from the function that is sent back to the location of the function call)

52 **Return Values (Page 2)**

- Many functions "calculate" a return value (*result*) that is passed back to the calling function
- Statements using the keyword return are used to *return a value* from function to the call location
- Any number of return statements may be coded within the function, any of which:
  1. Terminate execution of function at that point
  2. Return control to the location at which it was called and *passes the result back* to the call

53 **Return Values (Page 3)**

- Format for functions that return a result:

```
return expression;
```

  - Returns *expression* value to *calling* function
- Examples:

```
return 10;
return x;
return mouseX * mouseY;
return random(255);
```

54 **Return Value Example**

- Function call:
  ```
  int r = rgbRandom();
  ```
  - Execution of all functions take *precedence* over the assignment operator
  - A function that returns a value is an expression that "behaves" like a calculation (formula)
- Function definition:
  ```
  int rgbRandom()
  {
     return (int) random(255);
  }
  ```

56 **Using the Return Value**

- The function call should use the returned value as *part of another statement* …
  - Assignment statement, output statement, or some other *data related* operation
- Examples:
  ```
  r = rgbRandom();
  println( rgbRandom() );
  if (rgbRandom() < 127) …
  fill( rgbRandom(), rgbRandom(), rgbRandom() );
  ```

57 **Invalid Function Calls**

- Therefore functions that *return a value* should not stand alone
- The rgbRandom() function does return a value so the following is *invalid*:
  ```
  rgbRandom();
  ```
- The circle(*int*, *int*, *int*) function does not return a value so the following is *valid*:
  ```
  circle(mouseX, mouseY, rad);
  ```

60 **Function Types**

- Functions that *return a value* have a type
  - A function's type must be the same as the type of the value that is returned
- Format:
  type/void *functionName*([*parameterList*])
- Example:
  ```
  int rgbRandom()
  ```
- If a value is not returned, the return type is void
  ```
  void circle(int xCoord, int yCoord, int radius)
  ```

61 **The key System Variable**

- A Processing system variable that stores the last ASCII character that was used on the keyboard (either keyed or released)
- Special system constants that can be tested for include BACKSPACE, TAB, ENTER, ESC, etc.
- Examples:

```
if (key == 'A')
```

  - Differentiates between upper and lower cases

```
if (key == ENTER) // The Enter key
```

62 **The keyCode System Variable**

- A Processing system variable that stores the last *non-ASCII* character that was pressed on the keyboard (either keyed or released)
- The special non-ASCII system constants that can be tested for include UP, DOWN, LEFT, RIGHT, SHIFT, ALT, CONTROL, etc.
- Example:

```
if (keyCode == UP) // The Up arrow key
```

67 **The sq and sqrt Functions**

- These are two Math functions:
  - The sq() function returns the square of a value, (the product of multiplying it by itself), e.g.

```
square = sq(9);  // square will be 81
```

  - The sqrt() function returns the square root of a value, e.g.

```
root = sqrt(9);  // root will be 3
```

69 **Using the Return Value**

- A function call that returns a value can be used to represent values in a larger expression (not just in assignment statements)
- Examples:

```
println( rgbRandom() );
circle(x, y, radius(x, y) * 2 );
fill( rgbRandom(), rgbRandom(), rgbRandom() );
```

75 **Final Review of Function Calls**

- So remember that there are three distinct elements involved in calling functions:
  1) The function call transfers control to the called function (and automatically *transfers back* to the call location when done executing)
  2) One or more *argument* values may be passed to *parameter* variables that are defined in the called function's header
  3) A value (the result of function's processing) may be *returned* after its execution