Functions In Processing

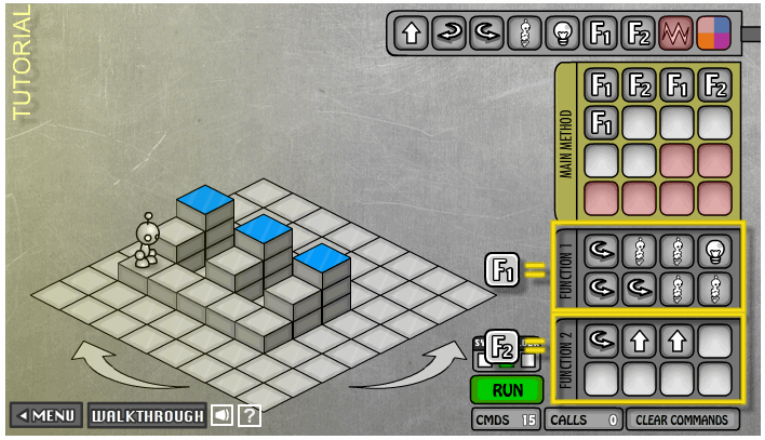
Lawrence Snyder
University of Washington, Seattle

Plan For Today

- We will review some of the datatype and variable information from last lecture, but mostly we will …
- Learn about functions in Processing

… Recall Our Love Affair With Functions

- Lightbot 2.0 we were introduced: "beautiful!"
- HW3: F.turn( ) …
- Lab4: Columns



I've said …

Abstraction … [Solid Gold Review]

- Formulating blocks of computation as a "concept" is **functional abstraction**

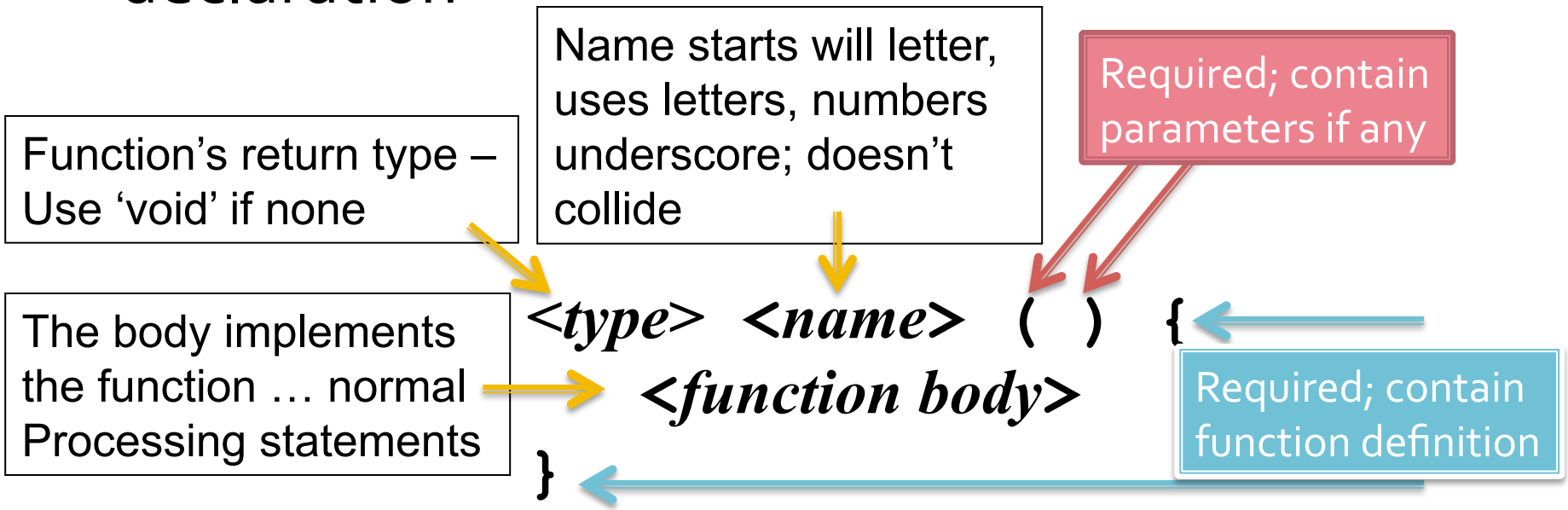
- What we did [in Lec 3] is important …
  - We spotted a coherent (to us) part of the task
  - We solved it using a sequence of instructions
  - We put the solution into a function "package", gave it a name, "process a riser," and thus created a new thing, a concept, something we can talk about & use
  - Then we used it to solve something more complicated … and then we did it again!

Abstracting [Solid Gold Review]

- Collecting operations together and giving them a name is *functional abstraction*
  - The operations perform a coherent activity or action – they become a *concept* in our thinking
  - The operations accomplish a goal that is useful – and typically – is needed over and over again
  - *Functions* implement functional abstraction: 3 parts
    - A name
    - A definition (instruction seq), frequently called a "body"
    - Parameters –stuff inside the parentheses, covered later

In Processing

- So, that's the idea … how does it look in Processing?
- Recall these components of the function declaration



Functions In Processing: Return

- Optionally, a function can return a value
- A function returns its value with the **return** statement … the stuff following return is the result
- The function is done when it reaches return

```
void draw_a_box ( ) {
                    rect(100,100, 20, 20);  // nothing to return
}
color pink (  ) {
                    return color(255, 200, 200); //give a color
}
```

Value Returning Functions …

- The returned value can be used just like any other value of its type
- Write `fill(pink( ));  //Set color to pink`
- Another example

```
float third_pi( ) {
    return 0.33333333 * PI;
}
```

- can be used

```
coneArea = third_pi() * r * r * h;
```

Parameters

- Parameters are the values that go inside the parentheses in a function
- For example

```
float coneArea ( float baseRad, float hite) {
    return third_pi * baseRad * baseRad * hite;
}
```

- Notice:
  - The datatype of the parameter must be given
  - Parameters are separated by commas
  - Parameter names like other names – no conflicts

Parameters And Arguments

- We've seen the function definition … now for the call

```
area = coneArea( 297.5, 100);
```

- The *call* causes the function to run
- Strange Terms:
  - When they are in the function definition, items in parentheses are called *parameters*

```
( float baseRad, float hite)
```

  - When they are in the function call, they are called *arguments*

```
( 297.5, 100)
```

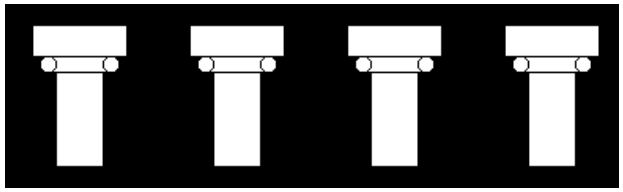
The Function Declaration & Calls

- The result:

```
void drawCol( int offset) {          // Draw a Column
  rect(20+offset, 250, 60, 20);      // Top stone
  rect(30+offset, 270, 40, 10);      // Stone below it
  ellipse(30+offset, 275, 10, 10);   // Left curl
  ellipse(70+offset, 275, 10, 10);   // Right curl
  rect(35+offset, 280, 30, 60);      // Actual column
}
```

- The calls

```
fill(255);
drawCol(0);
drawCol(100);
drawCol(200);
drawCol(300);
}
```

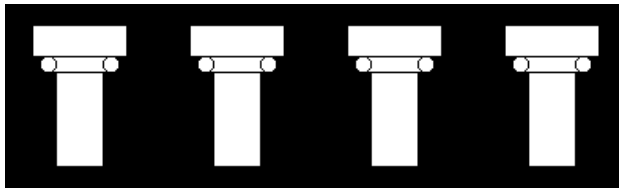


Better Than Copy/Paste Edit!

Just Do It!

Parameters and Arguments

- The value of the argument [from the call] is the value used for the parameter [inside the function definition]



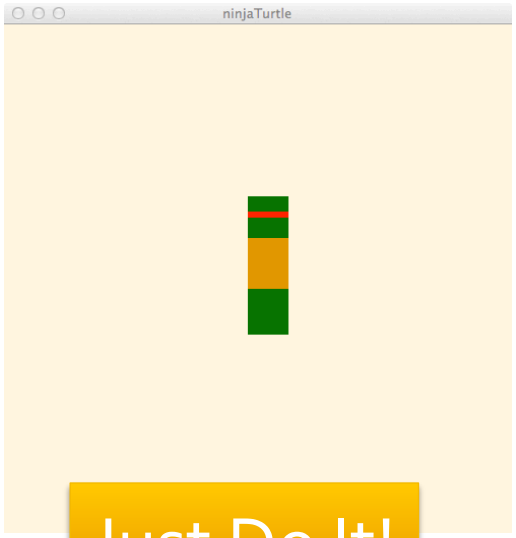
```
void drawCol( int offset) {
  rect(20+offset, 250, 60, 20);
  rect(30+offset, 270, 40, 10);
  ellipse(30+offset, 275, 10, 10);
  ellipse(70+offset, 275, 10, 10);
  rect(35+offset, 280, 30, 60);
}
```

```
fill(255);
drawCol(0);
drawCol(100);
drawCol(200);
drawCol(300);
}
```

General Turtle

- Turtles are a concept that deserves a function
  Parameters: x-position, y-position, goggles

```
void draw() {
  background(255, 245, 220);
  turtle(0, 0, color(255,0,0) );
}
void turtle( float xoffset, float yoffset, color goggles) {
  fill(0,100,0);
  rect(240+xoffset,260+yoffset, 40, 45);
  fill(219,136,0);
  rect(240+xoffset,210+yoffset, 40, 50);
  fill(0,100,0);
  rect(240+xoffset,190+yoffset, 40, 20);
  fill(goggles);
  rect(240+xoffset, 184+yoffset, 40, 6);
  fill(0,100,0);
  rect(240+xoffset, 169+yoffset, 40, 15);
}
```



Just Do It!

Define The Whole Team

- So a turtle's now defined, lets use it!

```
void raff( float xpos, float ypos) {
     turtle( xpos, ypos, color(255,0,0));
}
void mike( float xpos, float ypos) {
     turtle( xpos, ypos, color(250,122,0));
}
void leo( float xpos, float ypos) {
     turtle( xpos, ypos, color(0,0,255));
}
void don( float xpos, float ypos) {
     turtle( xpos, ypos, color(128,0,128));
}
```

Good News / Bad News

- The use of a function to define the four turtles has advantages and disadvantages
- Good news
  - Much shorter specification
  - The items that change are explicitly noted, params
  - Ninjas are easy to reposition, and we could give them a standard position, like upper left corner:

```
turtle(-240+xpos,-169+ypos,color(255,0,0));//Raff at (0,0)
```

- Bad news
  - Can't drop-and-reassemble – need separate offsets

Parameters

- Parameters are automatically declared (and initialized) on a call, and remain in existence as long as the function remains unfinished
- When the function ends, the parameters vanish, only to be recreated on the next call
- It is wise to choose parameter names, e.g. o-f-f-s-e-t that are meaningful to you
  - I chose offset as the orientation point of the figure in the x direction
  - Notice that I used that name a lot, and the meaning to me remained the same