IAT 265

Multimedia Programming Review

Final Exam

- Date: Monday August 15 8:30AM - 11:30AM
- Room: SUR 2600
- Coverage: Everything covered from week 1 to week 12

Note: this slides cover mainly the skeleton, you need to hang more muscles & meats on them with your own review

Final Exam Question Format

- **Question 1: Concept Matching (40 pts)**
- **Question 2: Multiple Choices (40 pts)**
  - **Concept based**
  - **Code based**
- **Question 3: Write a simple sketch (20 pts) (involving class/subclass, instantiations & method calling)**

Sample Questions for Multiple Choices

- Concept based:

1. Which of the following variables could be a primitive variable(s)?

   a) String s = "siat";
   b) int n = 10;
   c) char c ='y';
   d) both b) & c)

2. What is the fastest sorting algorithm known so far?

   a) Bubble sort
   b) Quick sort
   c) Tree sort
   d) Insertion sort

Sample Questions for Multiple Choices (1)

Code based

```
void drawCircle (float x, float y, float d) {
    ellipse(x, y, d, d );
    if (d  > 10) {
        d *= 0.75;
        drawCircle (x, y, d );
    }
}
```

```
void setup() {

  size(200, 200);
  drawCircle (width/2, height/2, width);

}
```

1. What will this code sample draw?

a) A sequence of concentric circles

b) A sequence of tangent circles

c) A sequence of overlapping circles

d) None of the above

2. What is the diameter of the fifth circle drawn?

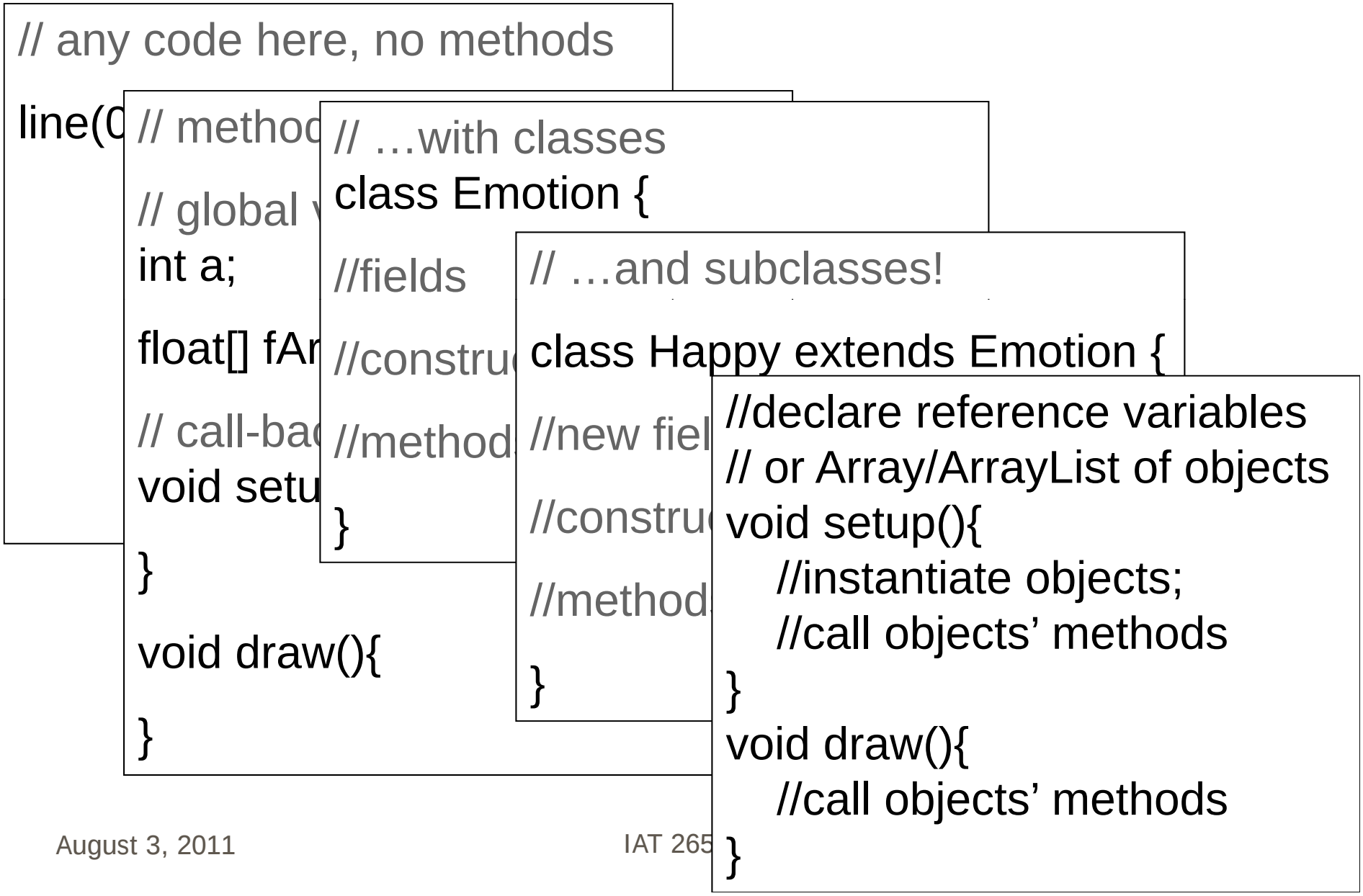
a) 25

b) 15

c) 12.5

d) 6.25

Overview: big picture



Drawing shapes

```
line(x1, y1, x2, y2);
```

```
triangle(x1, y1, x2, y2, x3, y3);
```

```
rect(x, y, width, height);
```

rectMode() – CORNER, CORNERS, CENTER

```
ellipse(x, y, width, height);
```

ellipseMode() – CENTER, CENTER_RADIUS, CORNER, CORNERS

Controlling color and line

Colors represented as Red Green Blue (RGB) values

Each one ranges from 0 to 255

`background(R,G,B);` – set the background color

`stroke(R,G,B);` – set the colors of the outline (default black)

`noStroke();` – no outline drawing around shapes

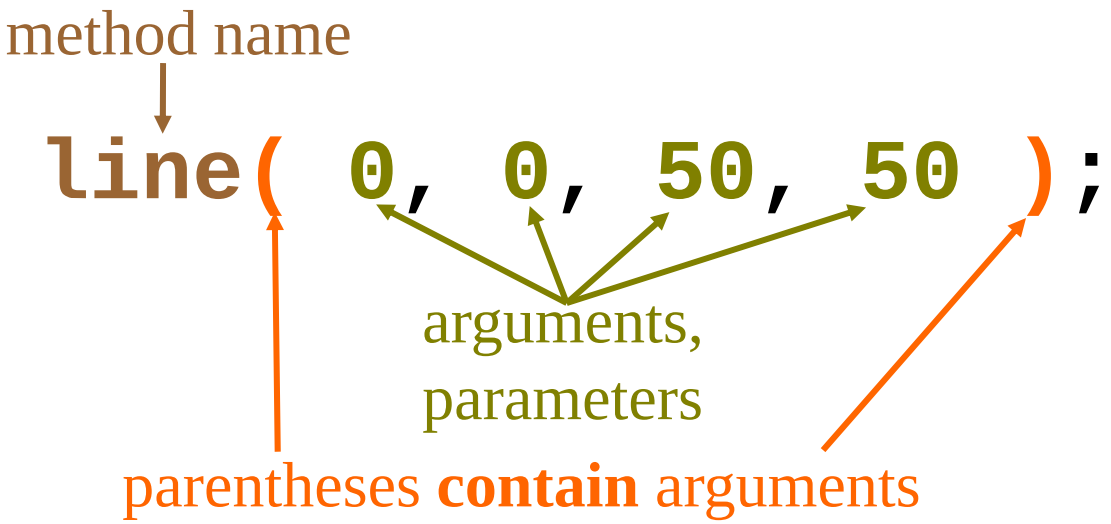
`fill(R,G,B);` – set the fill color for shapes (default white)

`noFill();` – don't fill the shapes (background shows through)

`strokeWeight(w);` – line width for outlines (default 1)

Syntax

- The study of the principles and rules for constructing sentences in natural languages



Variables

- A variable is a named box for storing a value
- You can put values in a variable by using the assignment operator (aka "=")

  e.g. `x = 1;`
- To use the value stored in a variable, just use the variable's name

  e.g. `line( x, 0, 50, 50);`

Variables have a data type

- Data type: characteristic of variable telling what kind of data it holds

```
int x;  // variable x can hold integers (int)
int y;  // variable y can hold integers (int)

x = 20;  // store 20 in x
y = 30;  // store 30 in y
point(x, y);  // use the values of x and y to draw a point
```

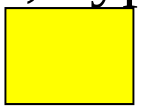
Creating an int variable

Code

```
// declare
int anInt;
```

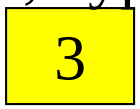
Effect

Name: anInt, Type: int



```
// initialization
// by assignment
anInt = 3;
```

Name: anInt, Type: int



- You can declare and assign at the same time:

```
int anInt = 3;
```

The "primitive" types

int – integers between 2,147,483,648 and 2,147,483,647

float – floating point numbers (e.g. 3.1415927, -2.34)

char – a single character (e.g. 'c')

byte – integers between -128 and 127

boolean – holds the values *true* or *false*

color – holds a color (red, green, blue, alpha)

color is a Processing type

Control flow

- By default Processing (Java) executes the lines of a method one after the other
  - Sequential flow
- Often we want which steps are executed to depend on what else has happened
- That is, we want *conditional* control flow

If

if statements introduce conditional execution

```
if ( <boolean expression> )
{
  // do this code
}
```

<boolean expressions> have one of two values:
true or **false**

Some boolean expressions

`anInteger == 1` true if variable anInteger is equal to 1

`x > 20` true if variable x is greater than 20

`x == 20` true if x is equal to 20, it's not so this is false

`!` is the **not** operator – reverses true and false so,
`(x != 20)←→ !(x == 20)`

This is not a boolean expression:

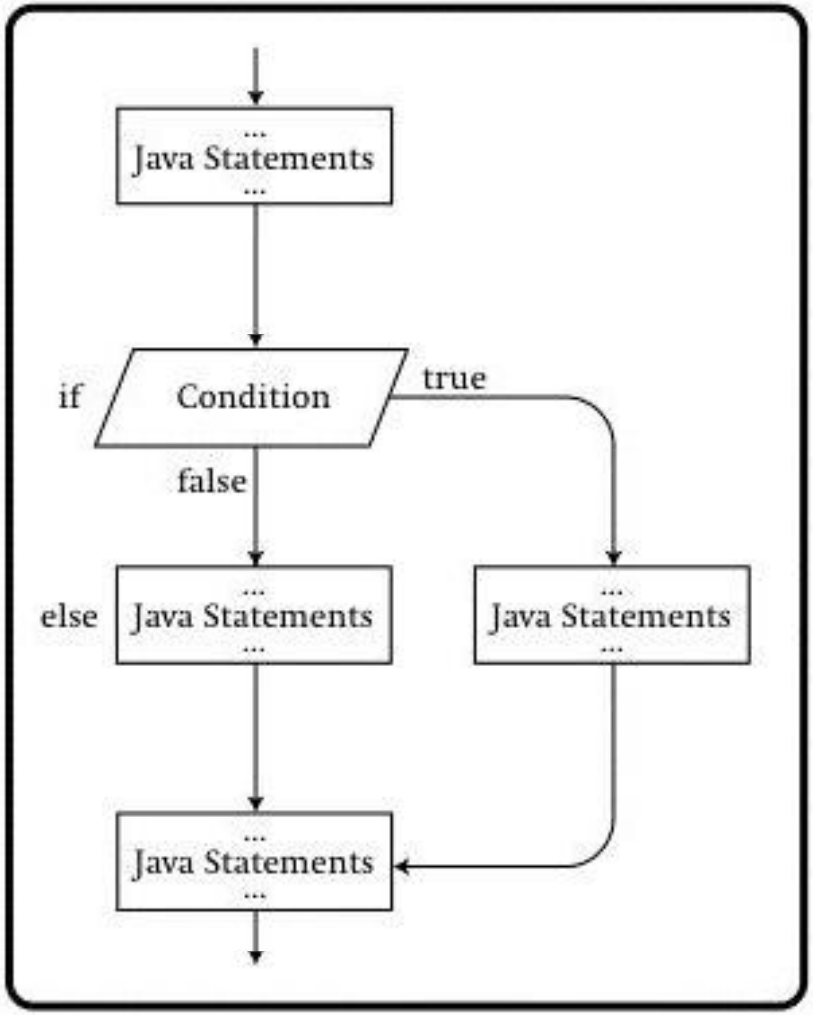
```
anInteger = 1;
```

Flowchart of the *if-else* Structure

- The *if-else* structure allows for branching code into two blocks: executing one or the other

- The syntax:

```
if (condition) {
    statements_for_true;
} else {
    statements_for_false;
}
```



Nested if-else

- Sometimes you need to have more than two branches in the flow of your program based on a set of related conditions
- That is where you can use the *nested if-else* statements. The syntax for it is as follows:

```
if (condition1) {
    statements_for_true_condition1;
}
else if (condition2) {
    statements_for_true_condition2;
}
else { //if all conditions evaluate to false
    statements_for_false_conditions;
}
```

Example *Nested if-else*

```
boolean drawRect = true;
boolean drawEllipse = true;

if (drawRect) {
    fill( 0, 200, 0 ); // fill with green
    rect( 30, 30, 40, 40 );
}
else if( drawEllipse ){
    ellipseMode(CORNER); // Draw the ellipse from the upper-
                         // left corner of its bounding box
    fill( 0, 200, 220 ); // fill with cyan
    ellipse( 30, 30, 40, 40);
}
else { //if all above false
    triangle( 30, 30,  30, 80,  80, 30 );
}

line(   0, 0, 100, 100 );
line( 100, 0,   0, 100 );
```

Compound Conditions

- You can use && or || operator to form compound conditions
- For &&: both sides of it must be true (i.e. *true && true*) for the result to be true
- Example:

```
boolean drawRect = true;
boolean drawInGreen = false;

if (drawRect && drawInGreen) {
    fill( 0, 200, 0 ); // fill with green
    rect( 30, 30, 40, 40 );
}
```

while loops

```
while( <boolean exp.> )
{
   <code to execute each time>
}
```

- Repeatedly executes the code body while the *boolean expression* is true

While Loops

- Use a while loop

```
int i = 10;
while(i <= 100) {
  line(i, 10, i, 200);
  i = i + 10; //will make i<=10
              //false finally
}
```

```
line(10, 10, 10, 200);
line(20, 10, 20, 200);
line(30, 10, 30, 200);
...
```

- A while loop will run the code inside the braces repeatedly until the condition in the parentheses is false.

for loops

```
for( <init. statement> ; <boolean expression> ;
                                    <final statement> )
{
   <code to execute each time in loop>
}
```

First executes the initialization statement

- Then tests the *boolean expression* – if true, executes the code once
- Then repeats the following:
  - execute *final statement*,
  - test *boolean expression* → execute *code* if true

Nested Loops

- Nesting of loops refers to putting one loop inside the braces of another

```
for(int i = 10; i <= 100; i += 10) {
  for(int j = 10; j <= 100; j += 10) {
    point(i, j);
  }
}
```

- For each **i**, the inside **j** loop will run through,
- then **i** will increment,
- then the inside loop will run again.

break

- break is a general control statement that lets you exit the current control structure

```
int[] intArr = new int[10] ;
for( int i = 0 ; i < intArr.length; i++ )
{
   print( intArr[i] );
   if( i == intArr.length – 1 )
   {
       break ;
   }
   print( “, “ );
}
```

Signature

- *Signature* is a term that means
  - The full specification of the method name
- Method signature = *return type + method name+ (parameters if any)*
- You may have more than one method with the **same name – overloading**
- **Not** more than one method with **same signature**

Overloading: same name diff sigs

```
Asteroid() {
    xPos = random(0, 400);
    yPos = random(0, 400);
    …
}
// another constructor!!
Asteroid( boolean big, float initX, float
  initY, long lastMillis)
{
    xPos = initX ;
    yPos = initY ;
     …
}
```

The `setup()` function

- Used to define some properties of your window or initialization of your variables:

```
int xPos = new int[20];

void setup() {

 size(640, 480);
 smooth(30);

 for(int i=0; i<arr.length; i++){
     arr[i] = random(width);
 }

}
```

`draw()`

`draw()` is a builtin Processing method **that you need to define**

`draw()` is called repeatedly by the Processing system

- Put code in draw() when you need to constantly update the display (for example, animating an object)

setup() and draw() are examples of *callbacks*

- A callback function is defined by the programmer
  - The callback gets called in response to some internal event
  - You usually don't call callback functions directly with your own code.
  - setup() and draw() are predefined within Processing as to-be-called-if-defined

Tracking Mouse Position

- Processing uses two global variables to hold the current mouse position:
  - ***mouseX***, ***mouseY***: current x and y of cursor, relative to the top-left of drawing window

```
void draw() {
 background(204);
 line(mouseX, 20, mouseX, 80);
}
```

A simple program that moves a line left or right to follow your cursor

Reacting to Mouse Clicks

- Use the variable ***mousePressed***

```
void draw() {
  if(mousePressed) {
    point(mouseX, mouseY);
  }
}
```

This simple function will draw a point in the window whenever the mouse is in the clicked position

- There are similar variables for **mouseReleased**, **mouseMoved**, and **mouseDragged**.

Mouse callback methods

- There are several built-in methods you can fill in to respond to mouse *events*

```
mousePressed()    mouseReleased()
mouseMoved()      mouseDragged()
```

Example:

```
void mousePressed()
{
  if( mouseButton == LEFT ){
    println( “Left Mouse Button was pressed” );
    loop();  // activate drawing again
  }
}
```

Arrays

- An **array** is a contiguous collection of data items of one type

- Allows you to *structure* data
  - Accessed by index number

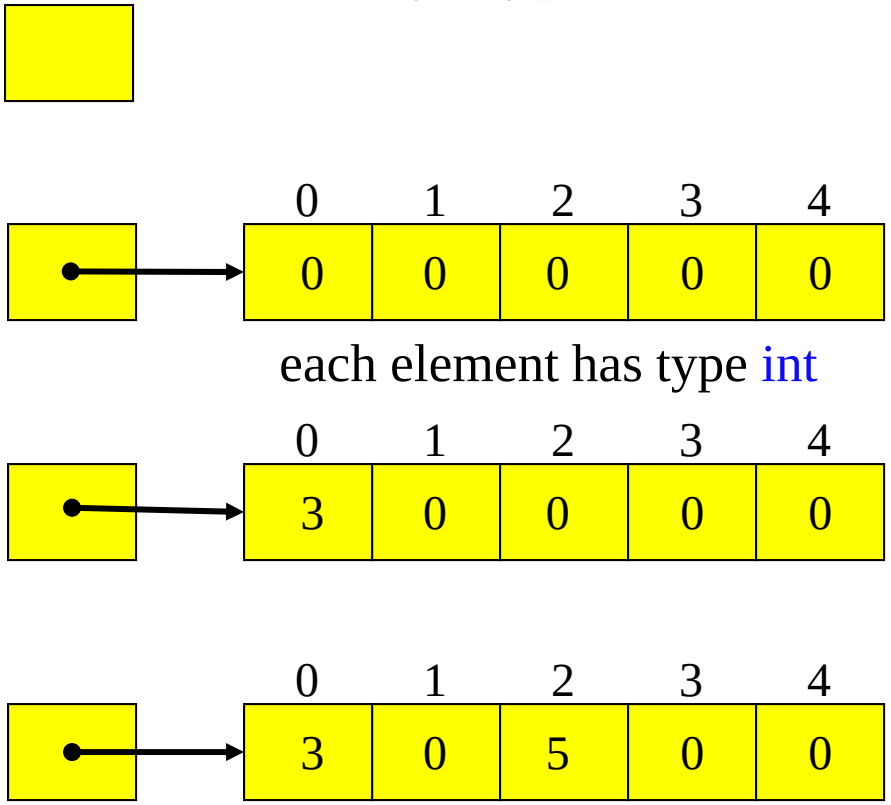
Creating an array of ints

Code

```
// declare int array
int[] intArray;

// initialize int array
intArray = new int[5];

// set first element
intArray[0] = 3;

// set third element
intArray[2] = 5;
```

Effect

Name: intArray, Type: int[]



Object Oriented vs. Procedural Languages

Procedural (e.g. C)

- We create some data representing a fish
- We write a *procedure* that can accept the data and draw the fish

Object Oriented (e.g. Java)

- We create a *class* that contains fish data AND a procedure to draw it
- The data and the procedure (ability to draw) are all in ONE "container"

What an object can offer?

- About Who you are:
  - Relevant properties/states (e.g. Fish: sizes, location, alive ...)
- About What you can do:
  - Behaviors of an object (e.g. Fish: move, collide, dodge, ...)

Parts of a class

- Classes define *fields*, *constructors* and *methods*
- Fields are the variables that will appear inside every instance of the class
  - Each instance has its own values
- Constructors are special methods that define how to build instances (generally, how to set the initial values of fields)
  - a) share the same name as the class; b) no return type
- Methods are how you *do things* to instances

Classes vs Objects

- A Class is a blueprint for fish
- An Object is a fish
- Many fishes, one blueprint

Defining a class

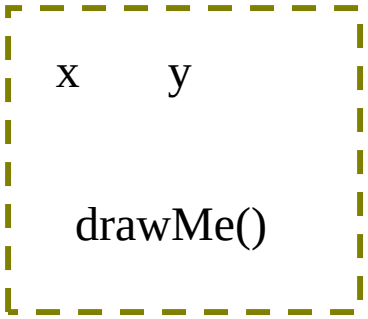
```
class MySquare {
  int xPos, yPos;

  MySquare(x, y) {
        xPos = x;
        yPos = y;
  }

  void drawMe() {
        rect(xPos, yPos, 50, 50);
  }
}
```

fields

constructor

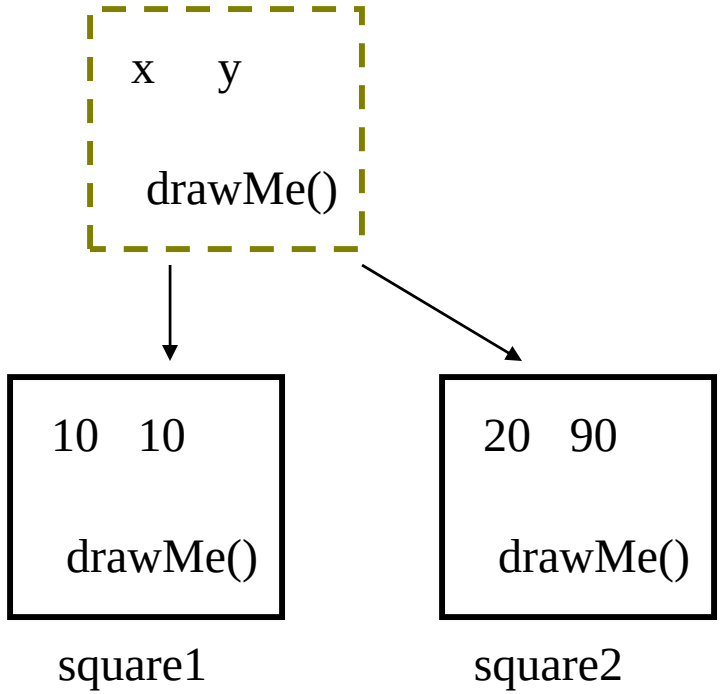
(one kind of method)

methods



Instantiation & initialization

```
class MySquare {
   int xPos, yPos;

   MySquare(x, y) {
         xPos = x;
         yPos = y;
   }

   void drawMe() {
         rect(xPos, yPos, 50, 50);
   }
}
```



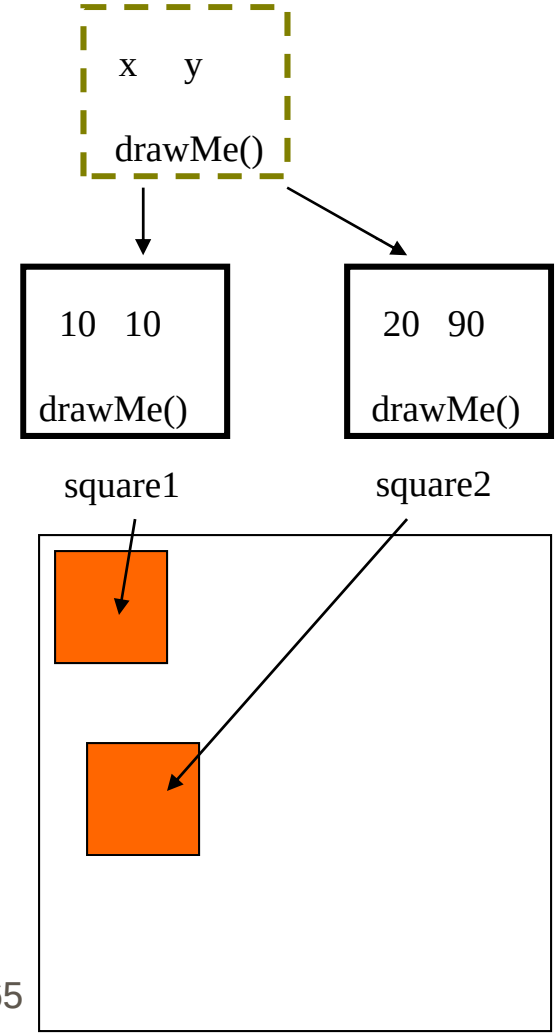

```
MySquare square1 = new MySquare(10, 10);
MySquare square2 = new MySquare(20, 90);
```

Method calling

```
class MySquare {
   int xPos, yPos;

   MySquare(int x, int y) {
         xPos = x;
         yPos = y;
   }

   void drawMe() {
         rect(xPos, yPos, 50, 50);
   }
}
```

```
MySquare square1 = new MySquare(10, 10);
MySquare square2 = new MySquare(20, 90);
```

square1.drawMe();
square2.drawMe();



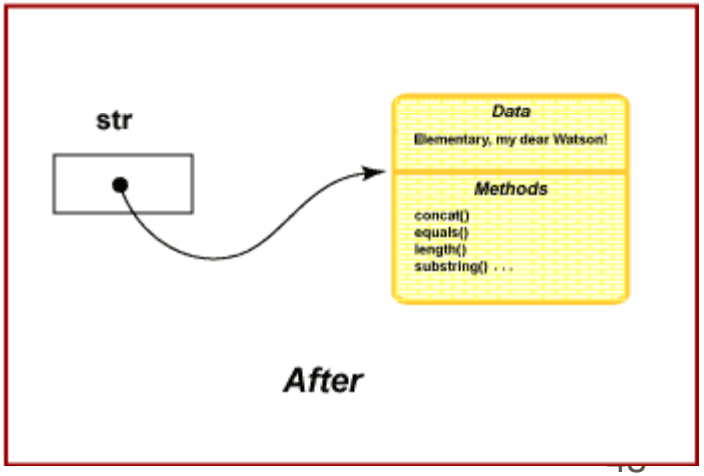
Difference between variables of Primitive and Reference types

- A primitive type variable is an *identifier for a value*
  - E.g. int num = 10;

num 10

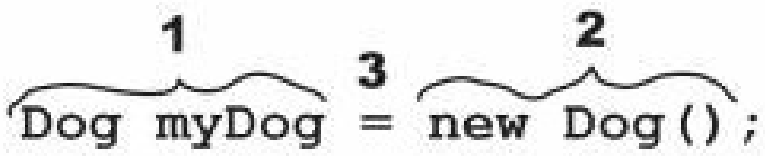
- A reference type variable is a *reference to an object's memory address* rather than a value:
  - E.g.

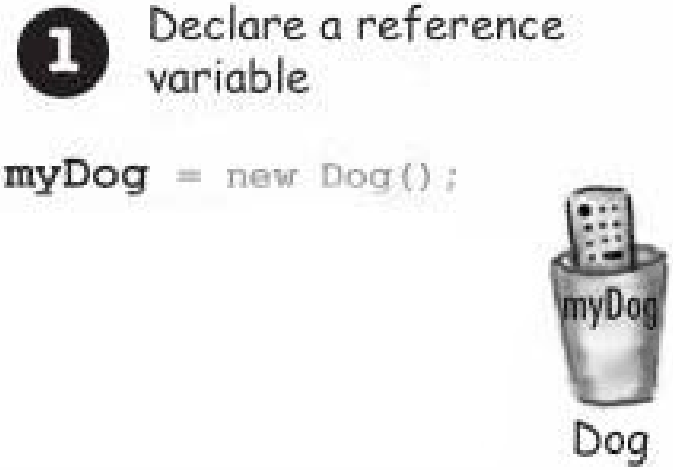
    String str = new String("Elementary, my dear Watson!");

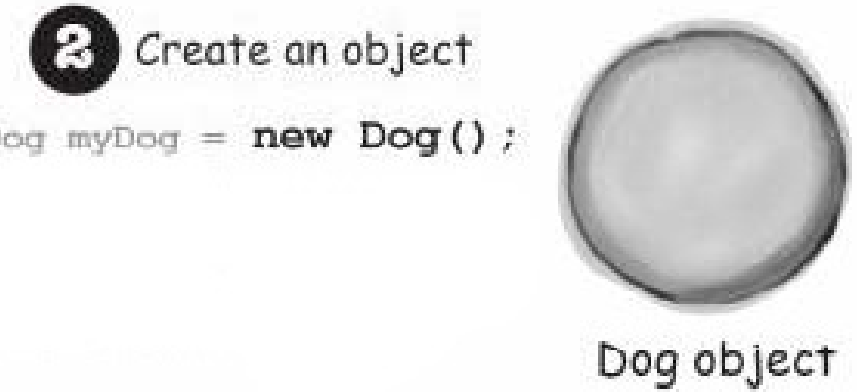


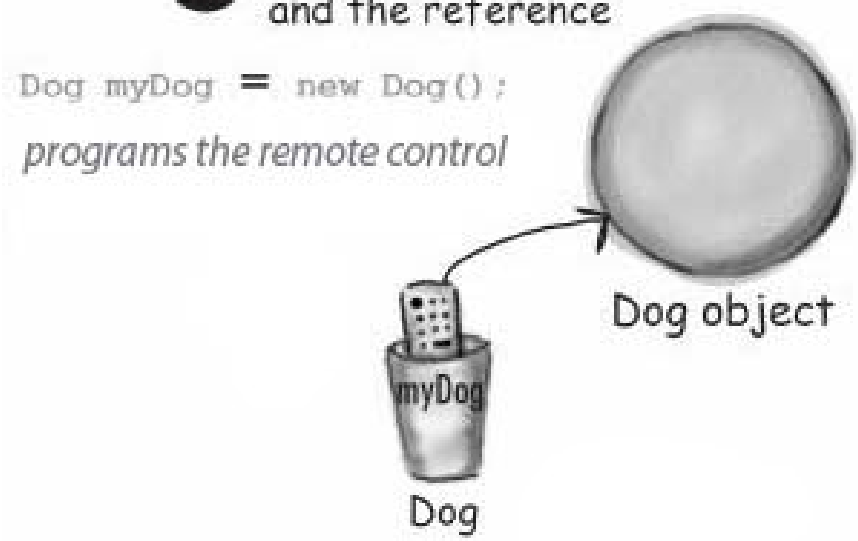
Reference

- Like a remote control
- a reference is a primitive thing that points at objects
- the assignment operator causes the reference to point at a new instance of the class


1
2
3
Dog myDog = new Dog();

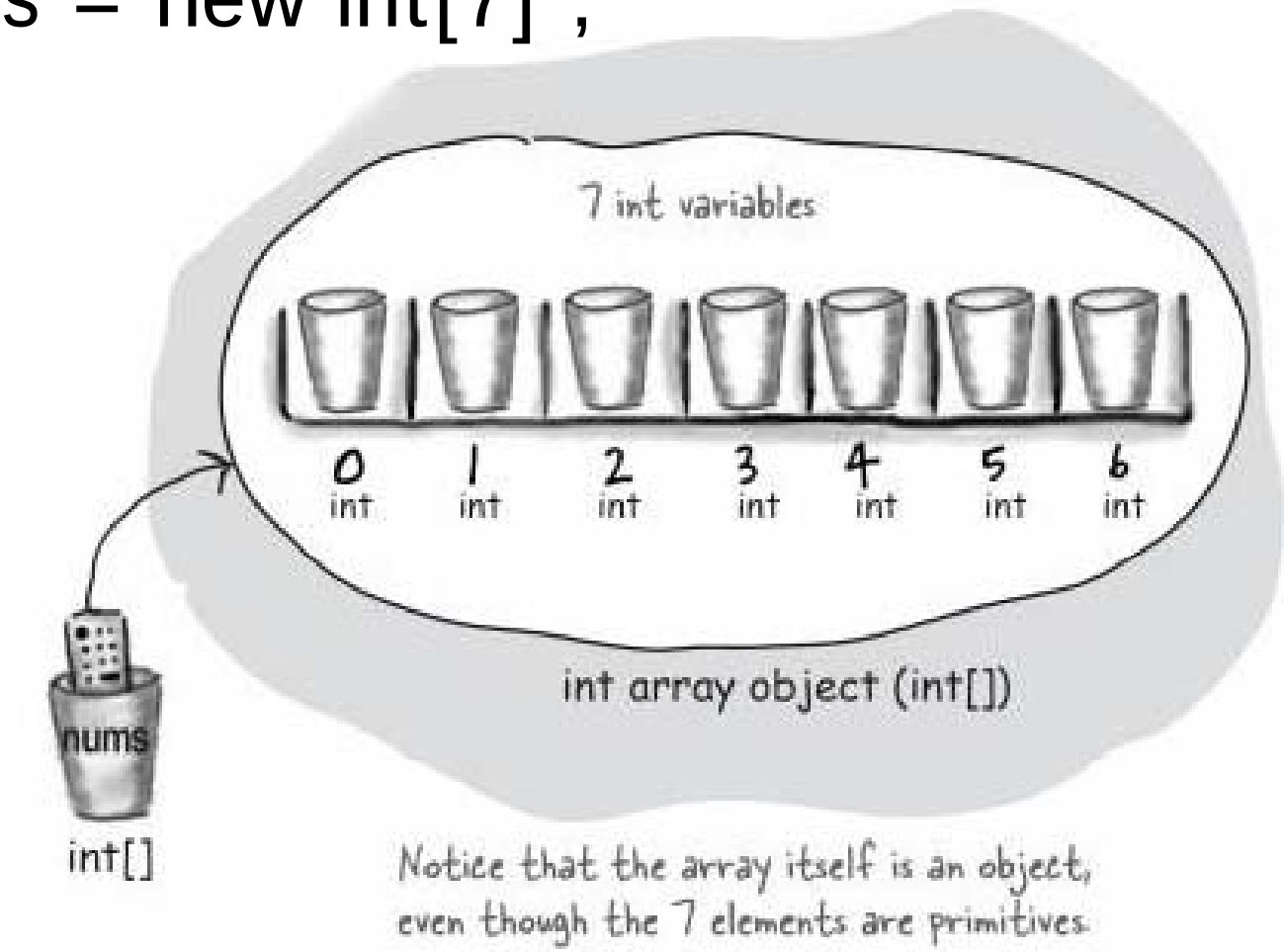

1 Declare a reference variable
Dog myDog = new Dog();
myDog
Dog


2 Create an object
Dog myDog = new Dog();
Dog object


3 Link the object and the reference
Dog myDog = new Dog();
programs the remote control
Dog object
myDog
Dog

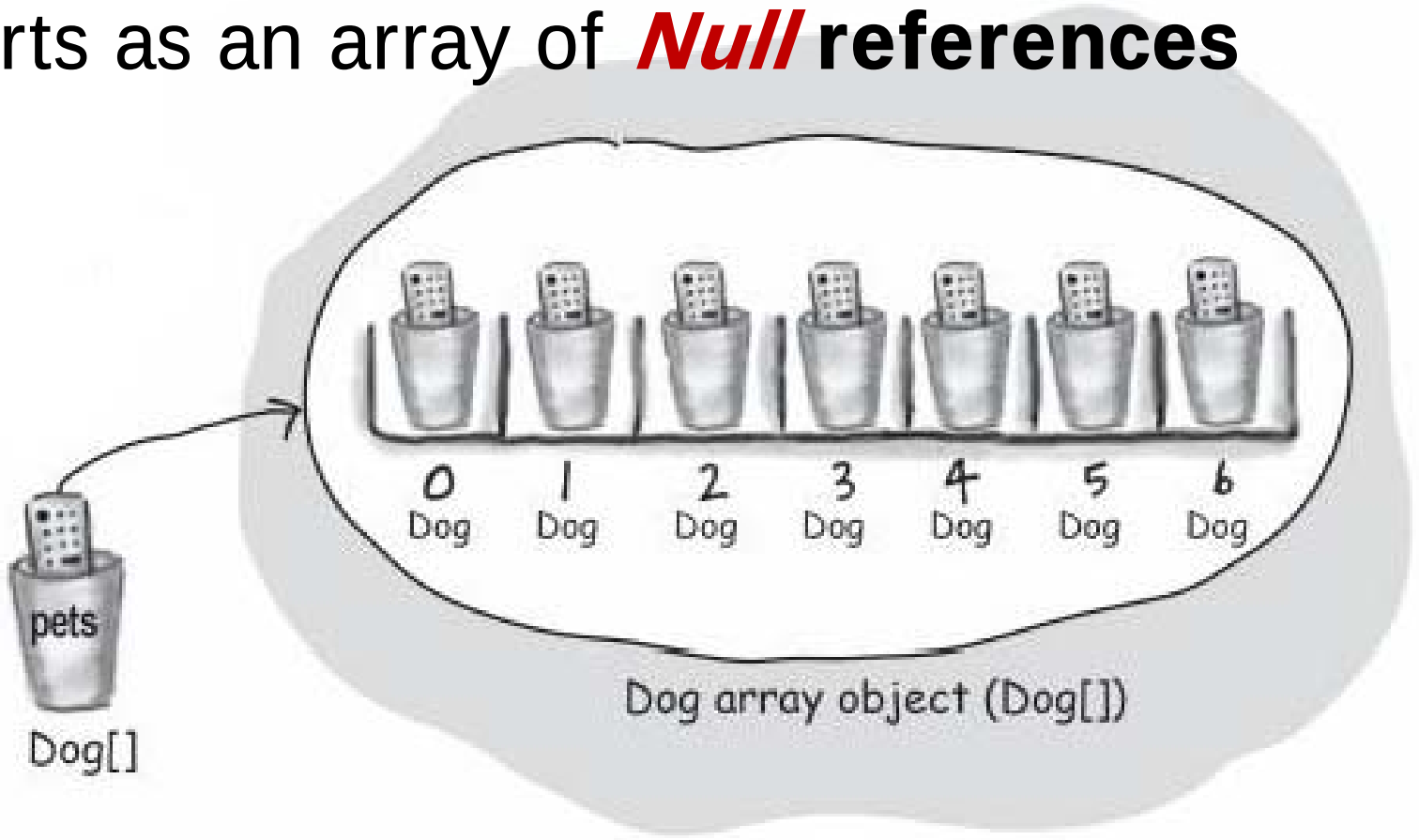
Arrays

- int[] nums = new int[7] ;



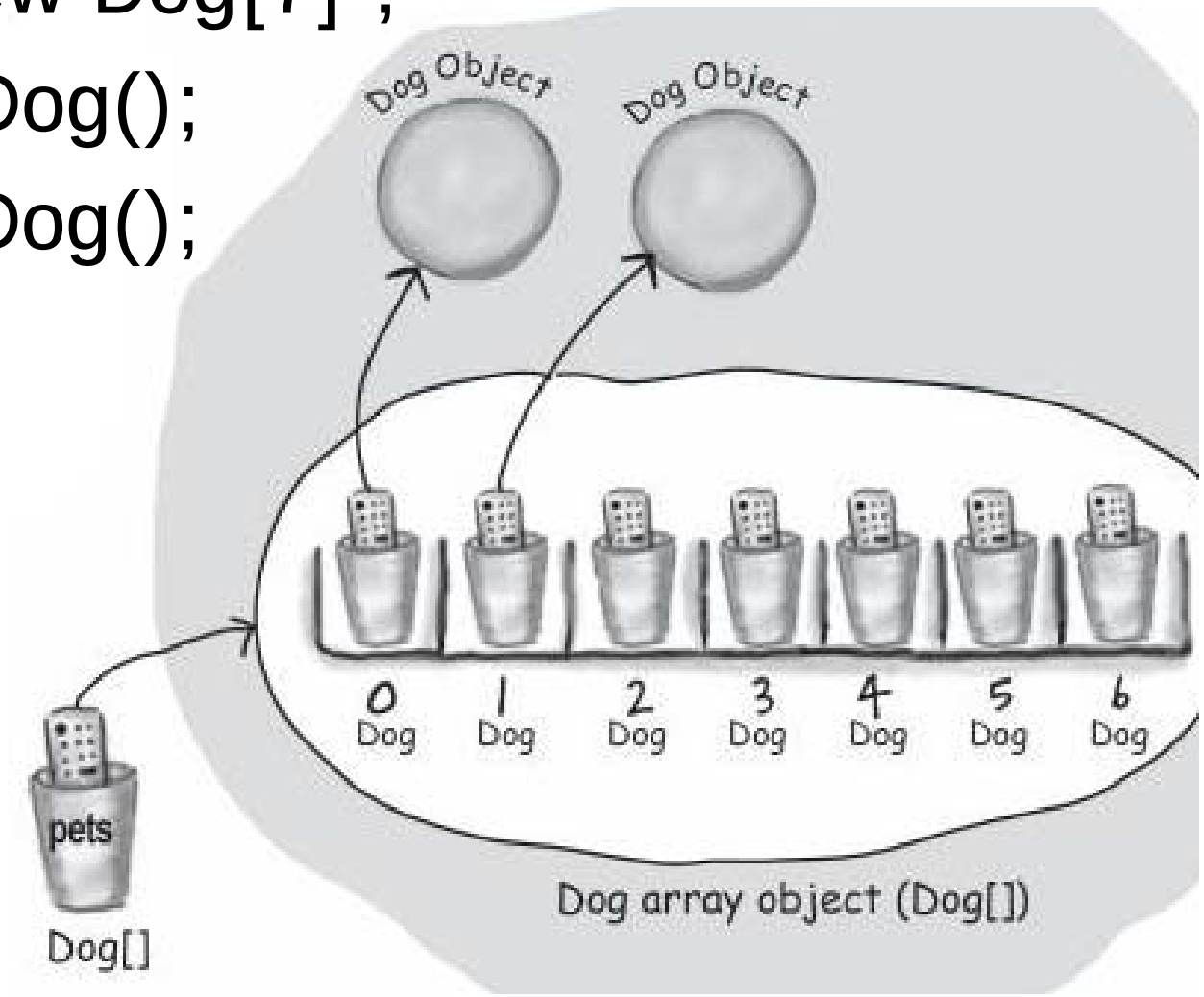
Array of objects

- Dog[] pets = new Dog[7];
- It starts as an array of ***Null*** **references**



Array of objects

```
Dog[] pets = new Dog[7] ;
pets[0] = new Dog();
pets[1] = new Dog();
```



Ex: Arrays of Objects

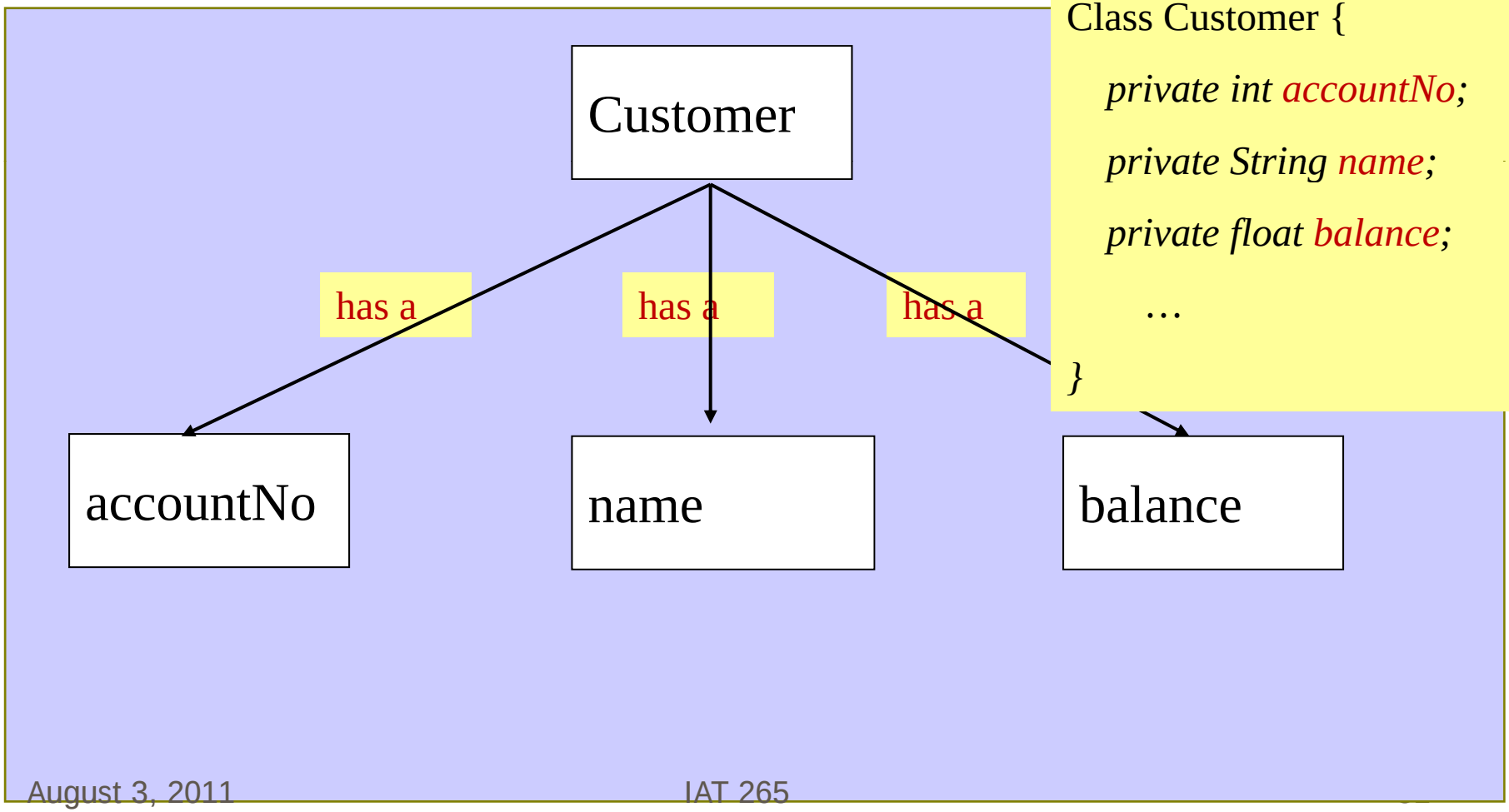
- Let's make a bunch of MySquares!

```
MySquare[] squares = new MySquare [10] ;

// initialize all of our squares.
for (int i = 0; i < 10; i ++) {
  squares[i] = new MySquare(i*10, i*10);
}

squares[4].drawMe();  // draw the 4th square.
```

Three Principles of OOP

- **Encapsulation**
- Inheritance
- Polymorphism

Encapsulation: good for 'has a' relationship

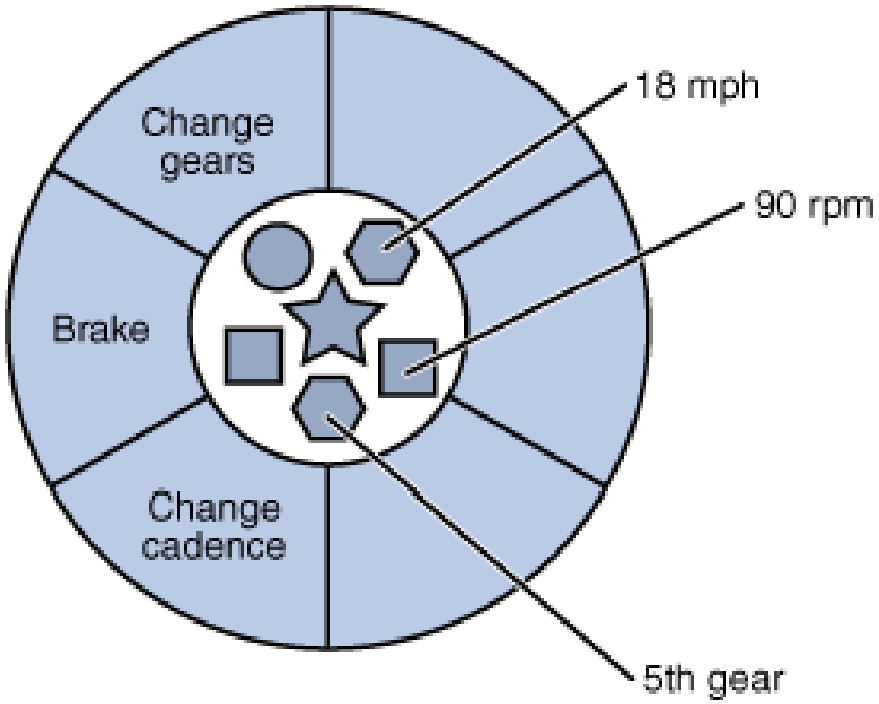


Data Encapsulation

- Hiding internal states using *private* keyword (default is *public*):
  - private fields;
- Performing all interaction through an object's
  - public methods

Bicycle as an Object

- State
  - int gear ;
  - float speed ;
  - float cadence ;
- Behavior
  - ChangeGears(int g);
  - Brake(float level);
  - ChangeCadence(float c);
  - int GetGear();
  - float GetSpeed(); ...



Encapsulation for Bicycle object

- An object's private fields can't be accessed by any objects/methods external to it

```
class Bicycle
{
      private int cadence = 0;
      private int speed = 0;
      private int gear = 1;
     //Constructor
     Bicycle () { }
} //end of Bicycle

//Tried to access private from an external method
void someMethodOutsideBicycle () {
   Bicycle bike = new Bicycle ();
   bike.gear = 5;
   print(bike.gear);
}
```

Illegal !!
You can't do these in Java

Walk around via setter and getter methods

- What can you do with private data?
  - to set it: setFieldName(varType newValue)
  - to get it: varType getFieldName()

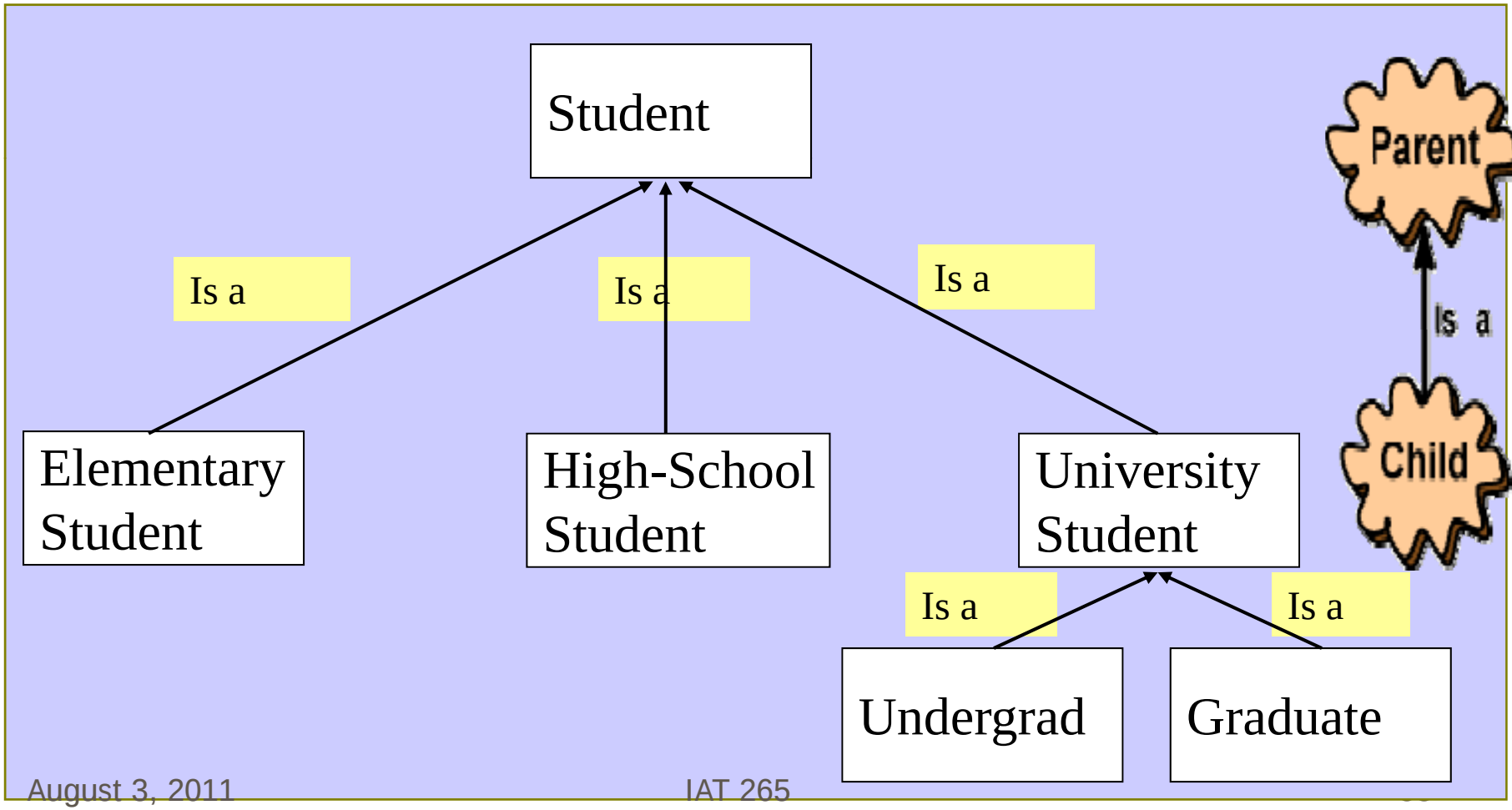
Example of Setter & Getter

```
class Bicycle
{
      ...
       private int gear = 1;
       ...
      void setGear( int g) { gear = g; }
      int getGear () { return gear; }
}
//Tried to access private from an external method
void someMethodOutsideBicycle () {
    Bicycle bike = new Bicycle ();
    bike. setGear(5);
    print(bike.getGear());
}
```

Three Principles of OOP

- Encapsulation
- **Inheritance**
- **Polymorphism**

Inheritance: good for '*is a*' Relationship



Inheritance

- **Inheritance:** child class extends the functionality of a parent class while inheriting all of its attributes and behaviors

  - Subclass inherits all the fields and methods from its super class

  - Subclass can define its own fields and methods

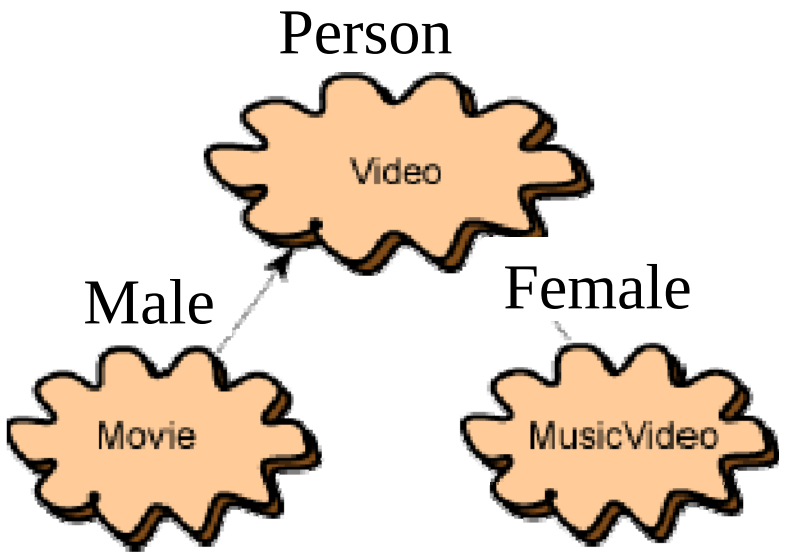
Why Inheritance?

- Mainly code reuse and extension:
  - allows classes to *inherit* commonly used attributes and behaviours from other classes, rather than reinvent the wheel
  - extend existing classes to add more functionality
    - Allows subclass' code to focus exclusively on features that are unique to itself

Example: Person vs. Male, Female

```
class Person {
    //Properties common to both Male/Female
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }

    // Behavior common to both Male / Female
    public void haveLunch(){
        println(this.name + " is having lunch");
    }
}
```

When field and parameter share the same name, use 'this.' to differentiate



Example: Person vs. Male, Female

- **class Male extends Person** {

```
        private String beardType;  //property unique to male
        public Male(String name, int age, String beardType) {
            super(name, age);
            this.beardType  = beardType;
        }
        public void showBeardType () {
            println(super.getName() + " is a male with beard of " + beardType);
        }
}
class Female extends Person {
        private String hairStyle;          //property unique to female
        public Female(String name, int age, String hairStyle) {
            super(name, age);
            this.hairStyle = hairStyle;
        }

        public void showHairStyle (){
            println(super.getName() + " is female with hair style of “ +
hairStyle);
        }
}
```

Super and *this*

- *this* is a keyword that always refers to the current object: Useful to refer to something of yourself within a class
  - this.field – refers to a field of yourself (useful to differentiate when you use the same names for fields and parameters)

- *super* is a keyword that always refers to the superclass portion of an object
  - super.method() – calls the superclass' method (but normally you just directly call the method without super.)
  - super(parameter if any) – calls the superclass' constructor

Inheritance: case study

- Subclasses inherit fields and methods from parent

```
class EatingBug extends Bug{
  …
}
```

Our subclass needs a constructor & something extra

- We want EatingBug objects to have a center dot with randomized color → need a field for that

```
color dotColor;
```

- We want the **EatingBug** constructor to do the same work as the Bug constructor as well as initialize **dotColor**

```
EatingBug(float x, float y, float chgX, float chgY,
float sz) {
  super(x, y, chgX, chgY, sz);
  dotColor = color(random(255), random(255),
    random(255));
}
```

super() here is to call the parent's constructor. Please note super() must be the 1st statement in children's constructors

Now we have EatingBug

- We can use EatingBug now in our example
- But, it's basically just a copy of Bug so far
- The only reason to define an EatingBug is to add new capabilities or to *override* old ones

Add an **eat()** method

- We want an EatingBug object's **eat()** method to eat smaller bugs and grow itself

```
void eat(Bug otherBug) {
    if(bSize > otherBug.bSize) {
      //grow itself by 10%
      bSize *= 1.1;
      //kill the other bug
      otherBug.alive = false;
    }
}
```

Put EatingBug objects into action

```
EatingBug[] bugs = new EatingBug[count];

void setup() {
    ...      //anything the same as before

  for(int i=0; i<count; i++) {
    bugs[i] = new EatingBug (
      random(gardenW), random(gardenH),
      random(-1,1), random(-1,1),
      random(12,36));
  }
}
```

```
void draw() {
    …      //anything the same as before

    //if bug i and bug k collide
    if(bugs[i].detectCollision(bugs[k]) {
      bugs[i].eat(bugs[k]);
    }

    …      //anything the same as before
}
```

Then what ...

- Can EatingBug has its own drawBug() method, so that it draws an EatingBug object differently (e.g. with a bigger dot at its center with randomized color)?
  - We know it inherits a drawBug() method from Bug, then which gets called at runtime?

- Given Bug is the parent of EatingBug, can we use it as the type for EatingBug objects?

These questions relate to OOP's third principle

- Encapsulation
- Inheritance
- **Polymorphism**
  - the ability to create a variable, a method, or an object that has more than one form
  - Two types:
    - **Overriding polymorphism**
    - **Inclusion polymorphism**

Overriding polymorphism

- A subclass replaces the implementation of one or more of its parent's methods (with same signatures)

- Can EatingBug has its own drawBug() method, so that it draws an EatingBug object differently?
  - Yes, and when drawBug() gets called at runtime, it is the one to be called (i.e. overrides its parent version)

Override Bug's drawBug()

```
//Override parent's drawBug method
  void drawBug() {
    //call parent's drawBug() method to draw
    // a regular bug
    super.drawBug();

    //Draw a center bigger dot on top of
    //parent's version
    pushMatrix();
    translate(bugX, bugY);
    if(alive) {    //draw only if the bug is alive

      //make the bug rotate
      if( changeX <0 ) {
        rotateY(PI);
      }

      //draw the center dot with dotColor
      fill(dotColor);
      ellipse(bSize/2, bSize/2, bSize/4, bSize/4);

      //redraw the body line
      stroke(160, 0, 0);
      line (0, bSize/2, bSize, bSize/2);
    }
    popMatrix();
  }
```

Inclusion polymorphism

- You can use a superclass as the type to declare a name that references to objects of its subclasses
  - The name could be a *variable* or a *parameter*

Example of Inclusion polymorphism

- Remember that we defined two children classes (**Male** & **Female)** of class **Person**
- Remember that classes are types
  - So **Person** is a type, so are **Male** and **Female**

- So, here are some legal assignments
  - `Male p1 = new Male("Mark", 17, "moustache");`
  - `Person p2 = new Male("John", 20, "none");`
  - `Person p3 = new FeMale("Linda", 18, "longhair");`

- But this is illegal
  - `Male p4 = new Person("Ken", 22);`

- So it is perfectly legal to do this:
  - `Bug bug = new EatingBug (random(gardenW), random(gardenH), random(-1,1), random(-1,1), random(12,36));`

Same goes for parameters as well...

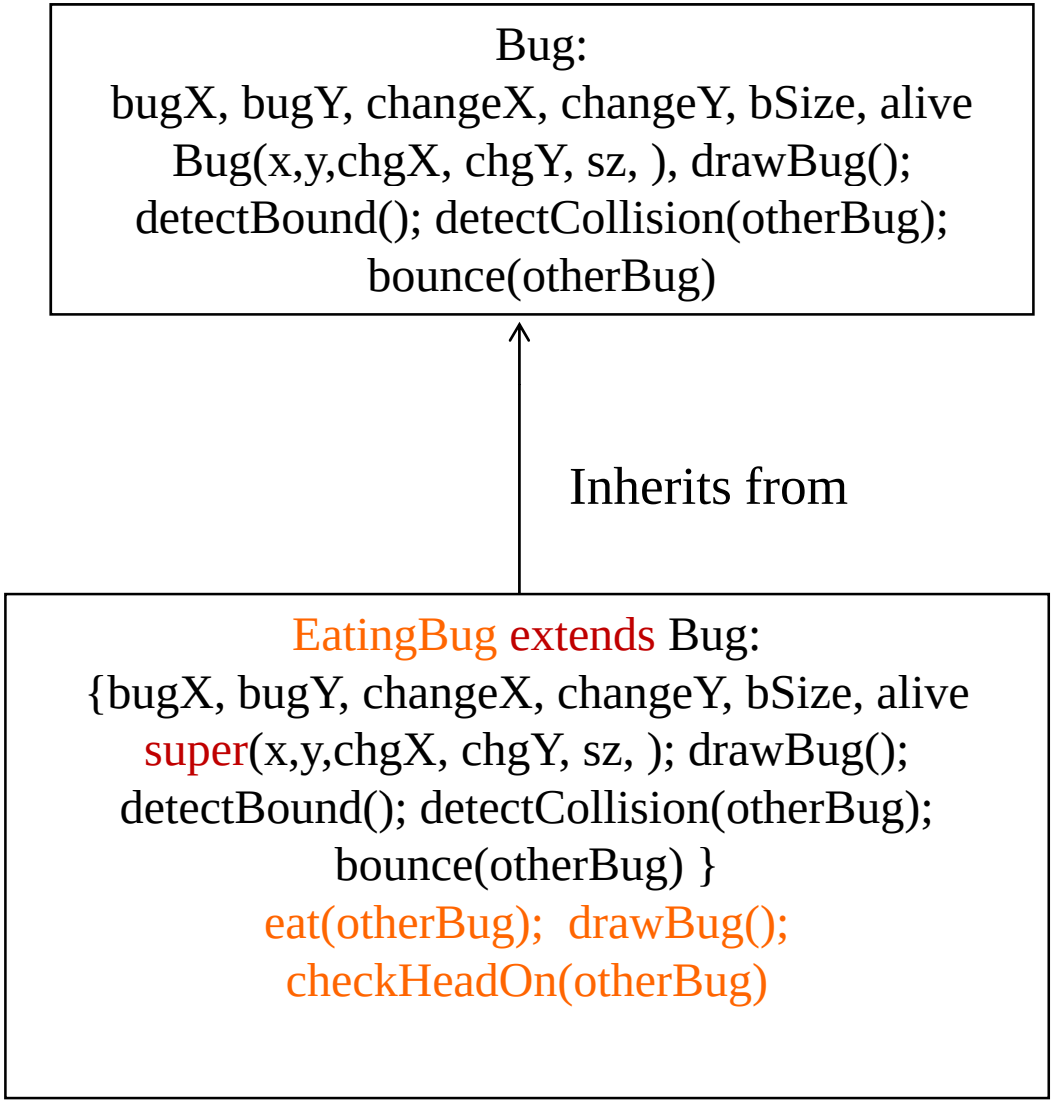
- A parameter of a superclass type can accept its subclass objects as arguments
  - This is useful when you have more than one subclass

```
void eat(Bug otherBug) {
  if(bSize > otherBug.bSize) {
    //grow itself by 10%
    bSize *= 1.1;
    //kill the other bug
    otherBug.alive = false;
  }
```

Here you can pass in objects of any Bug's subclasses as its argument,

e.g. an object of EatingBug

Bug Inheritance



String details

- A string is *almost* like an array of **char**s
  - char someletter = 'b';
  - String somewords = "Howdy-do, mr. jones?";
  - Note the use of double-quotes (vs. apostrophes)

- Like the objects we've created with classes, it has several methods, too...

String methods

- From http://processing.org/reference/String.html
  - length()
    - returns the size of the String (number of letters)
  - charAt(*number*)
    - returns the char at an index *number*
  - toUpperCase() *and* toLowerCase()
    - returns a copy of the String in UPPERCASE or lowercase respectively.
  - substring(*beginIndex*, *endIndex*)
    - returns a portion of the String from *beginIndex* to *endIndex-1*

```
String howdy = "Hello!";        String expletive = howdy.substring(0,4);
```

String concatenation

- Concatenation means – appending a string to the end of another string
- With Strings, this is done using the **+** symbol
- So, if you have:

```
String s1 = "She is the";        String s2 = "programmer." ;

String sentence = s1 + " awesomest  " + s2;
```

- You'll get out:

```
println(sentence);  // sentence = "She is the awesomest programmer."

// outputs:  She is the awesomest programmer.
```

Special characters

- Special characters
  - tab: "\t"
  - new line: "\n"

(\ tells the computer to look to the next character to figure out what to do)

String twolines = "I am on one line.\n I am \ton another."

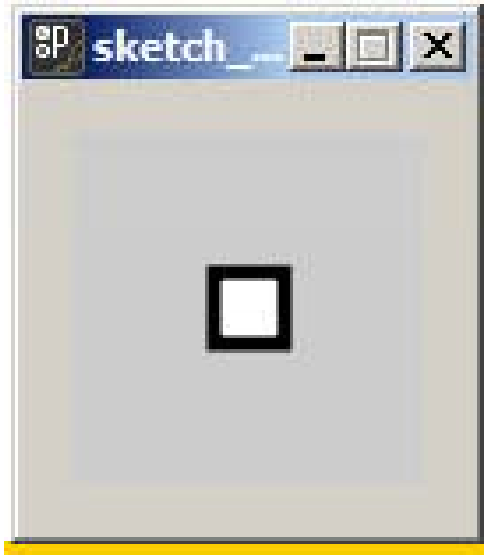
I am on one line.
I am on another.

  - other *escape characters* include "\\" "\""

Keyboard Interactions

- Processing registers the most recently pressed key and whether a key is currently pressed
  - The boolean variable ***keyPressed*** is *true* if a key is pressed and *false* if not
  - ***keyPressed*** remains *true* while the key is held down and becomes *false* only when the key is released

```
//draw a rectangle while any key is pressed

void setup()
{
  size(100,100);
  smooth();
  strokeWeight(4);
}
void draw()
{
  background(204);
 if (keyPressed==true)
 {

   rect(40,40,20,20);
 }
   else
   {
     line(20,20,80,80);
   }
 }
```



key variable – which key is pressed?

- The key variable (of char type) stores the most recently pressed or released key
  - Commonly used for keys included in the ASCII specification (e.g. a~z, A~Z, ENTER/RETURN, ESC,...)

```
void draw() {
    if(keyPressed) {
        if (key == 'b' || key == 'B' ) {
               fill(0);
        }
     } else {
        fill(255);
     }
     rect(25, 25, 50, 50);
}
```

keyCode variable – which coded key is pressed?

- The keyCode variable is used to detect special keys such as the UP, DOWN, LEFT, RIGHT arrow keys and ALT, CONTROL, SHIFT

  - When checking for these keys, it's necessary to check first if the key is coded, with the conditional "if (key == CODED)"

```
if (key == CODED) {
        if (keyCode == UP) {
                fillVal = 255;
        } else if (keyCode == DOWN) {
                fillVal = 0;
}
```

Key events callbacks

- ***keyPressed():*** called once every time a key is pressed

- ***keyReleased():*** called once every time a key is released

Key Mapping for Multi-key Interactions

```
// Key Mapping for Multi-key interactions
boolean downKey, upKey, leftKey, rightKey;
void keyPressed() {
  if (key == CODED && keyCode == RIGHT)   rightKey = true;
  if (key == CODED && keyCode == LEFT)   leftKey = true;
  if (key == CODED && keyCode == UP)   upKey = true;
  if (key == CODED && keyCode ==DOWN)   downKey = true;
}
void keyReleased() {
  if (key == CODED && keyCode == RIGHT)   rightKey = false;
  if (key == CODED && keyCode == LEFT)   leftKey = false;
  if (key == CODED && keyCode == UP)   upKey = false;
  if (key == CODED && keyCode ==DOWN)   downKey = false;
}
```

Case study: Key-controlled Avatar

```
class AvatarBug extends Bug {
  AvatarBug(float x, float y, float chgX, float chgY, float sz) {
    super(x, y, chgX, chgY, sz);
  }
  //method eat: eat the other bug if bigger otherwise kill itself
  void eat(Bug otherBug) {
    if(bSize > otherBug.bSize) {
     bSize *= 1.1;  //grow itself by 10%
     otherBug.alive = false;   //kill the other bug
    }   else  {
      this.alive = false;        //otherwise kill itself
      drawWaves();               //draw waves to show being killed
    }
  }
```

Case study: Key-controlled Avatar (2)

```
//method for drawing waves
 void drawWaves() {
  stroke(200, 0, 0);
  noFill();
  for(int i=1; i<=2; i++) {
    ellipse(bugX, bugY, i*bSize, i*bSize);
  }
 }

 //methods checkHeadOn() & drawBug()
 //are the same as EatingBug
 ...
```

```
//methods for move left, rigth, up & down
void moveRigtht(){
  if(changeX < 0)   changeX *= -1;
  bugX += changeX;
}

void moveLeft(){
  if(changeX > 0)    changeX *= -1;
  bugX += changeX;
}

void moveUp(){
  bugY -= changeY;
}

void moveDown(){
  bugY += changeY;
 }
}
```

Case study: Instantiation in the setup & draw functions

```
Bug[] bugs = new Bug[count];
AvatarBug avtBug;

  //Key Mapping for Multi-key Interactions
  //(Exactly the same as on page 14)
  boolean downKey, upKey, leftKey, rightKey;
  void keyPressed() { ...
  }
  void keyReleased() { ...
  }

//method respawn itself at the center
AvatarBug respawn() {
  return new AvatarBug(width/2, height/2, 4,
     4, 20);
}
```

```
void setup() {
  ...
  avtBug = respawn();
}
void draw() {
  ...
  //move avatar based on keypressed
  if (rightKey) avtBug.moveRigtht();
  if (leftKey) avtBug.moveLeft();
  if (upKey) avtBug.moveUp();
  if (downKey) avtBug.moveDown();

  //nested for loops i & k
  ...
  if(bugs[k].alive &&
      avtBug.detectCollision(bugs[k])  &&
      avtBug.checkHeadOn(bugs[k])) {
       avtBug.eat(bugs[k]);
       if(!avtBug.alive) {
         avtBug = respawn();
       }
  } }
```

How do we make objects disappear when destroyed?

- So far we have used conditional drawing
  - E.g. based on Bug's *alive* status, draw the bug only when it is true

```
void drawBug() {
   pushMatrix();
   translate(bugX, bugY);
   if(alive) {    //draw only if the bug is alive
       fill(0, 0, 60);
       ellipse(0, 0, bSize, bSize);
            ...
    }
    popMatrix();
}
```

- This is actually not the best approach, as the destroyed objects, although invisible, still sit in the memory

ArrayList

- It's a resizable list
  - Can add and delete things without worrying about declaring the size
- The main methods we care about are add(), get(), and remove(), and size()
- Steps in using ArrayList
  - Declare a variable of type ArrayList
  - Create a new ArrayList and assign it to the variable
  - Call add(), get() and remove() and size() on ArrayList as you need them

Using **ArrayList.add()**

- The argument type of the add method is Object
  - Object is the parent class of *all classes* in Java
  - With a parameter of Object type, you can pass in an object of any class

- So, to initialize our asteroids...

```
ArrayList bugs = new ArrayList();
for(int i = 0; i < count; i++){
   bugs.add(new Bug(
    random(width),random(height), random(1,1),
    random(-1,1), random(12,36));
}
```

Getting things out of an ArrayList

- **`ArrayList.get(int i)`** – returns the *ith* object (starting with 0)

- But this doesn't work!

  `bugs.get(i).drawBug();`

  Why?

Need to cast back from Object

- Since things are put in an ArrayList as Object, they come back out as Object
  - It's like they forget their more detailed type
  - So, when using ArrayList (or any Java collection class), you need to cast back to the more detailed type
- For our Bug example:

```
Bug bugi = (Bug)bugs.get(i);
```

```
//For the rest of our previous case study,
//just replace all bugs[i] with bugi,
//and it will do the same job as before
```

Destroying bugs

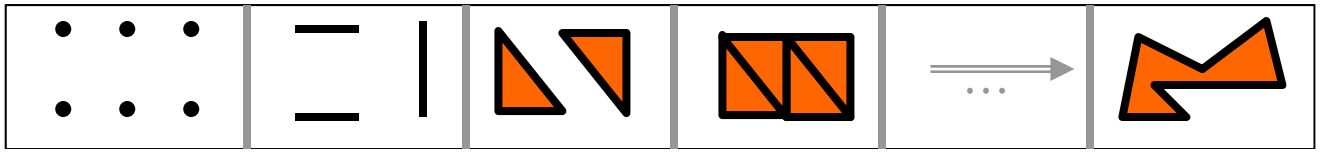
- When a Bug is eaten by an AvatarBug, we need to destroy it
  - This was the major reason for using ArrayList

```
void destroy(ArrayList bugs) {
  bugs.remove(this);
}
```

- By doing this, we don't need to check alive status for Bug objects, as any dead bug would be removed → doesn't exist anymore
  - AvatarBug still needs to check alive status to respawn, so make it a field of AvatarBug only

Building Special Shapes

- The `beginShape()` and `endShape()` functions allow us to draw irregular shapes from any number of points we define.
- Many types of Shape:
  - `POINTS, LINES, TRIANGLES, TRIANGLE_STRIP, TRIANGLE_FAN, QUADS, QUAD_STRIP, POLYGON`
  - *POLYGON* will be the most useful.



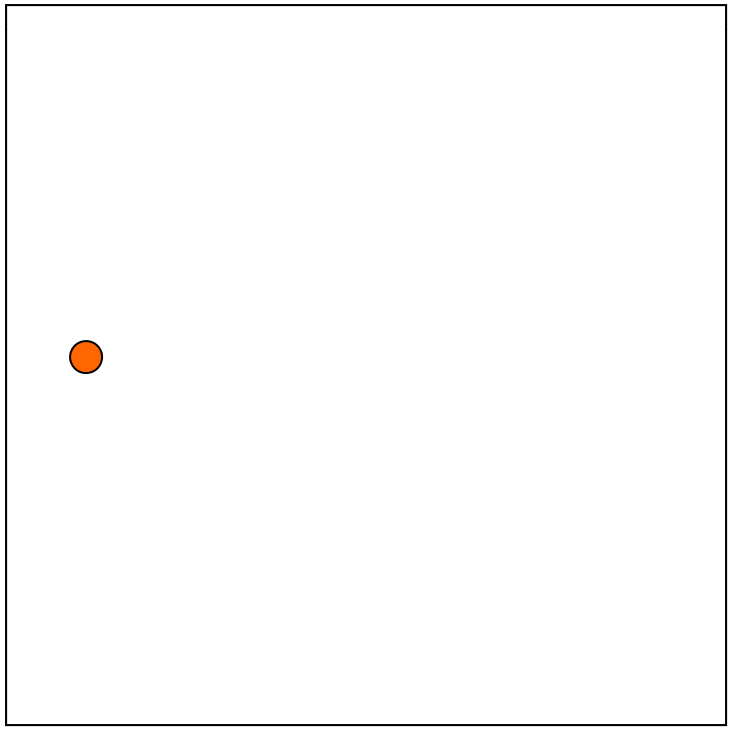
Building Polygons

- `beginShape( POLYGON );`
  - Tells the program to start the polygon.
- `vertex(x, y);`
  - Make as many calls to this as you have vertices in your polygon.
- `endShape( CLOSE );`
  - Finishes the shape, connecting the last vertex to the first vertex to close the polygon, then colors it with the current `fill()` color.

Building Polygons

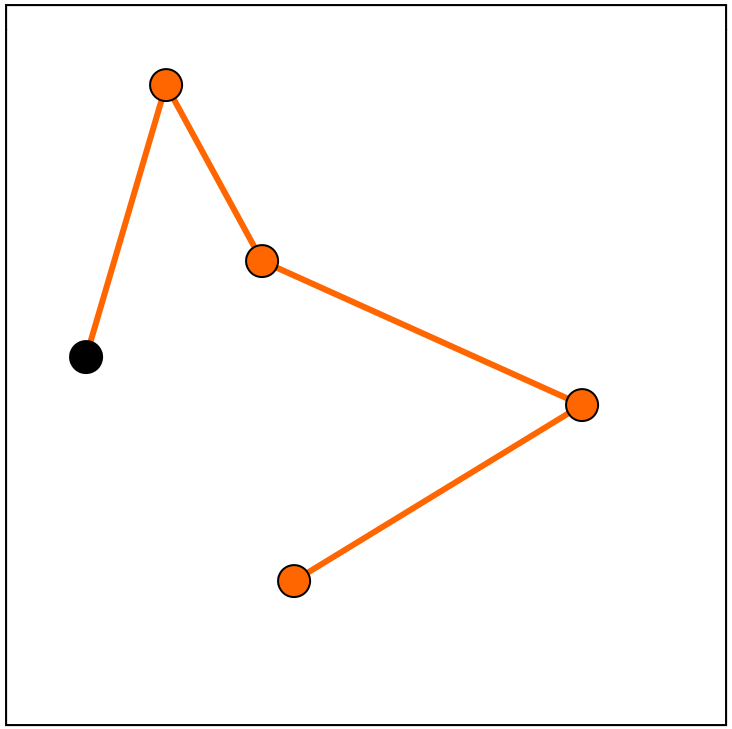
```
beginShape();
vertex(10, 50);
```

(starts a new polygon, and begins at point (10, 50).)



Building Polygons

```
vertex(20, 10);
vertex(30, 40);
vertex(80, 60);
vertex(40, 80);
```

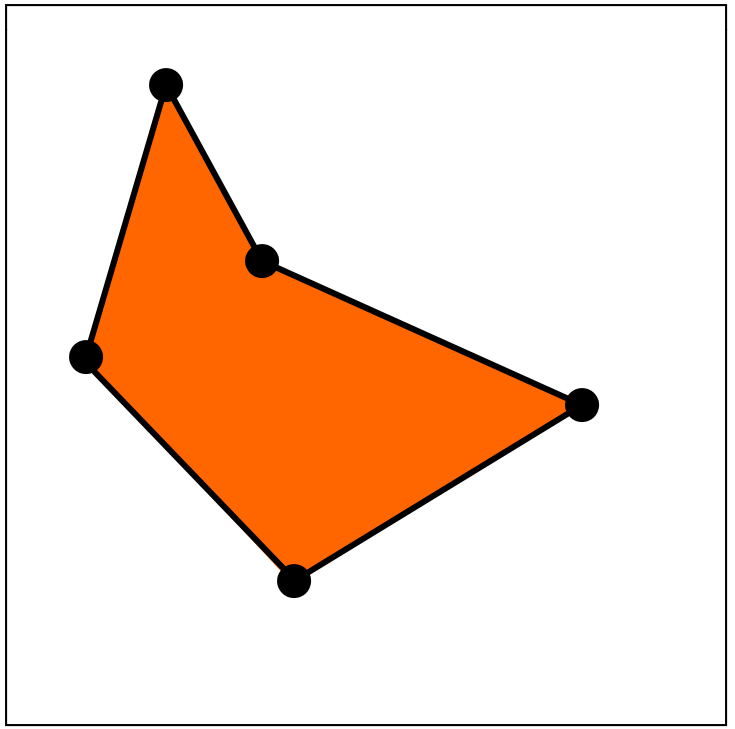


(adds 4 more points to the polygon, and connects them in the order they are called.)

Building Polygons

```
endShape(CLOSE);
```

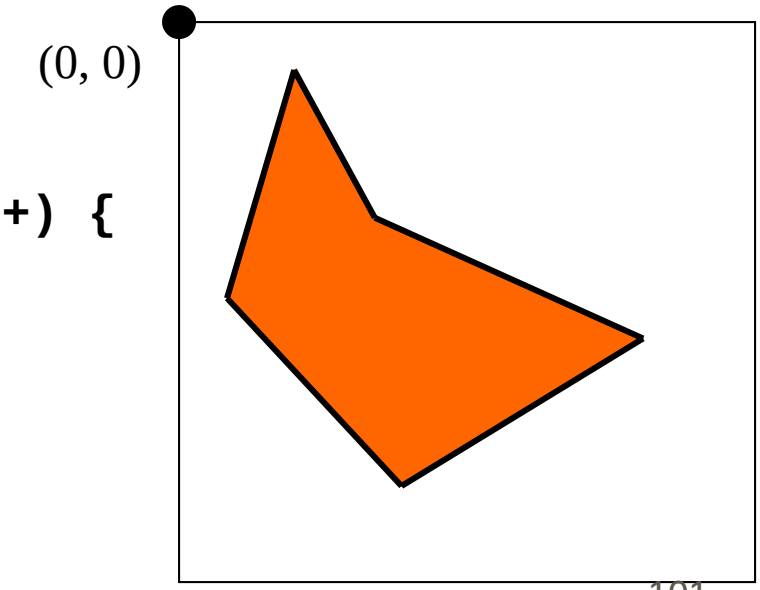
(connects the last point to the first point, and fills the polygon.)



Translation

- Translation gives us another way of drawing in a new location. It in essence, moves the point (0, 0) in our window.

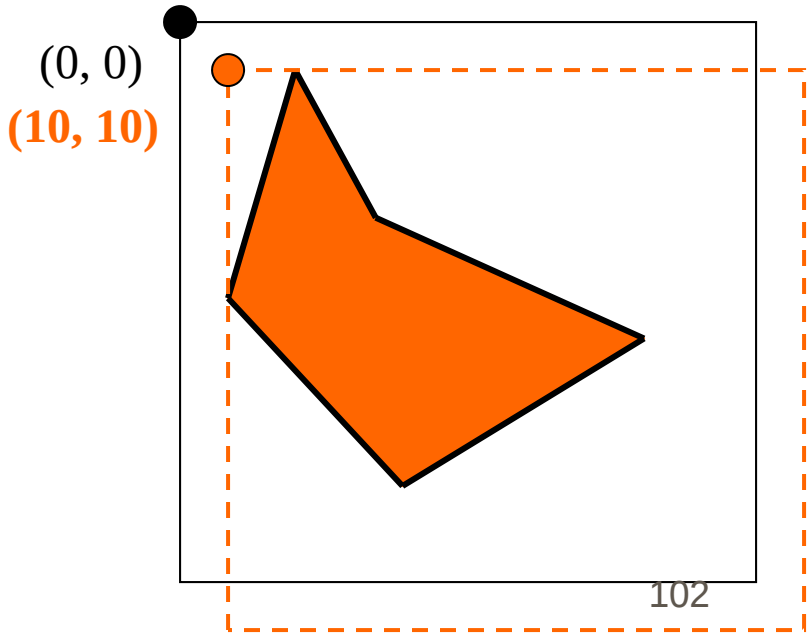
```
beginShape();
for(int i = 0; i < xvals.length; i++) {
  vertex(xvals[i], yvals[i]);
}
endShape(CLOSE);
```



Translation

- After the call to `translate()`, any drawing functions called will treat our new orange point as if it were (0, 0).

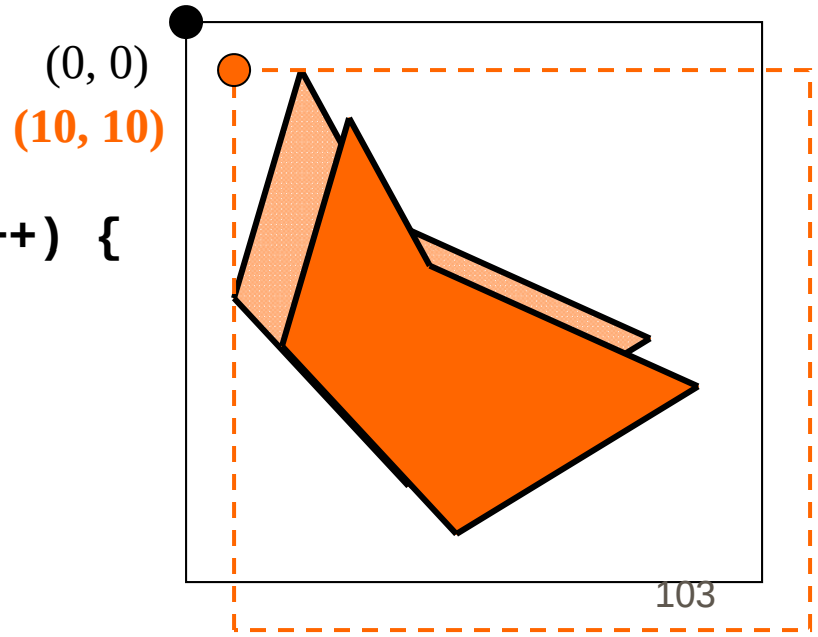
```
translate( 10, 10 );
```



Translation

- Draw same polygon again
- Now located x+10, y+10

```
beginShape();
for(int i = 0; i < xvals.length; i++) {
  vertex(xvals[i], yvals[i]);
}
endShape(CLOSE);
```



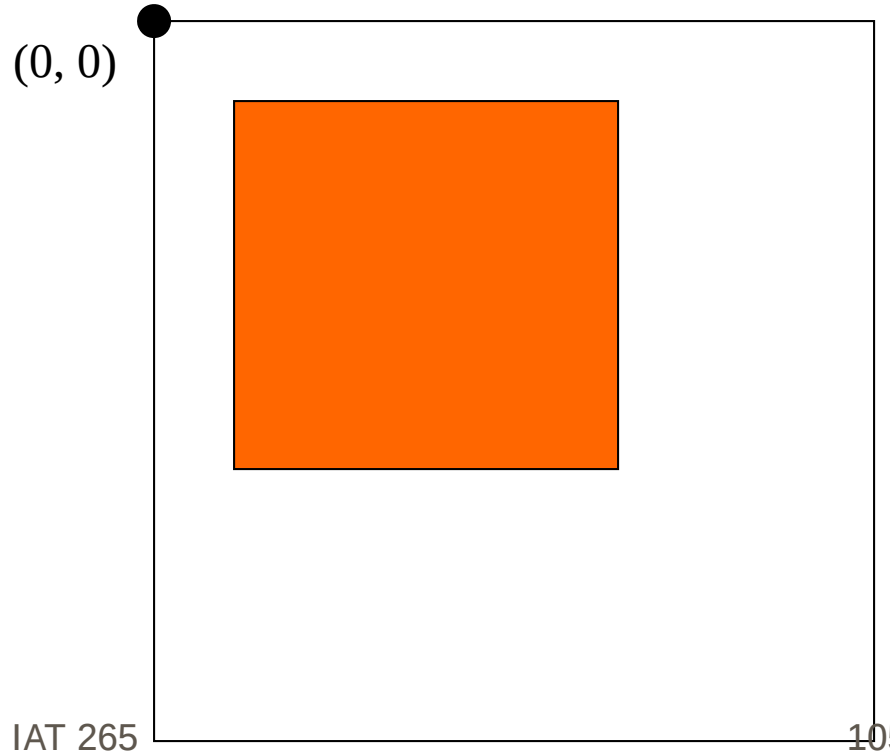
Rotation

- Much like Translation, Rotation moves our drawing space, so that we can draw at different angles.
- Most of the time, you'll want to use Rotation in conjunction with Translation, because `rotate()` rotates the drawing window around the point (0, 0).

Rotation

- Let's look at an example without translation:

```
rect(10, 10, 50, 50);
```

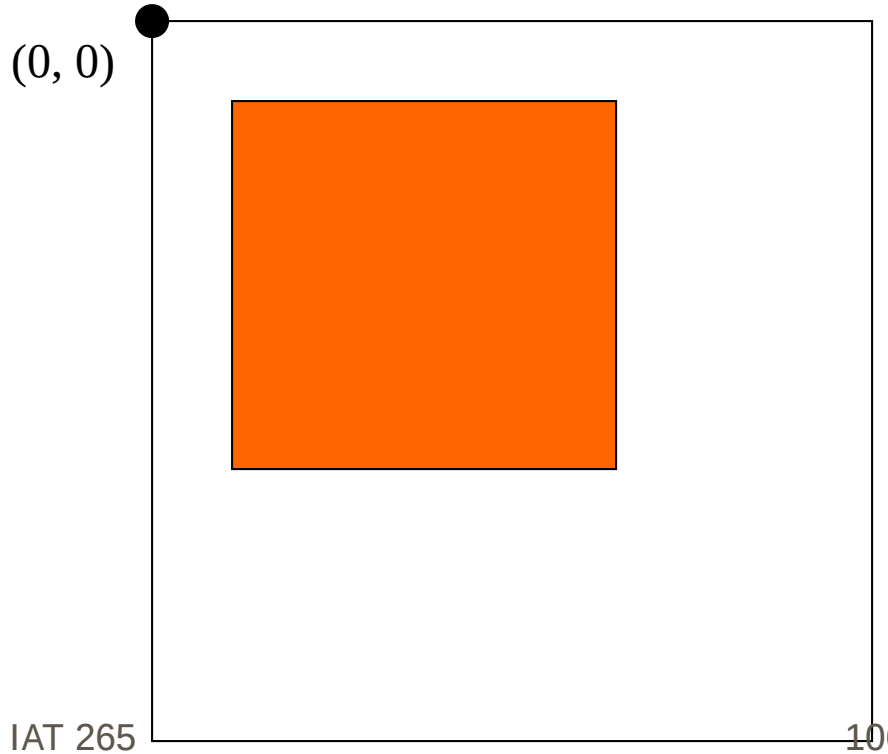


Rotation

- Make a variable with the value for 45 degrees in Radians.

```
float angle = radians(45);
```

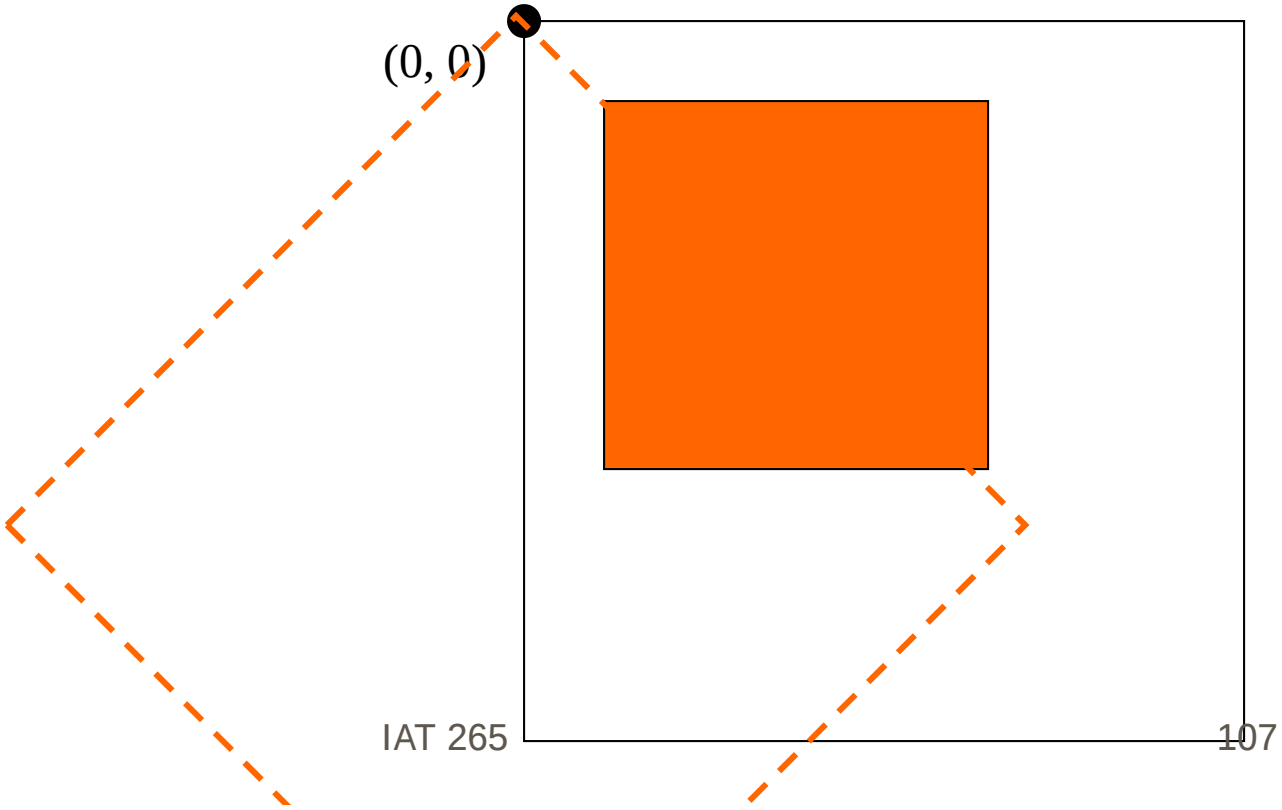
radians() takes an *int* or *float* degree value and returns a *float* radian value.



Rotation

- Rotate our drawing canvas 45 degrees around the origin.

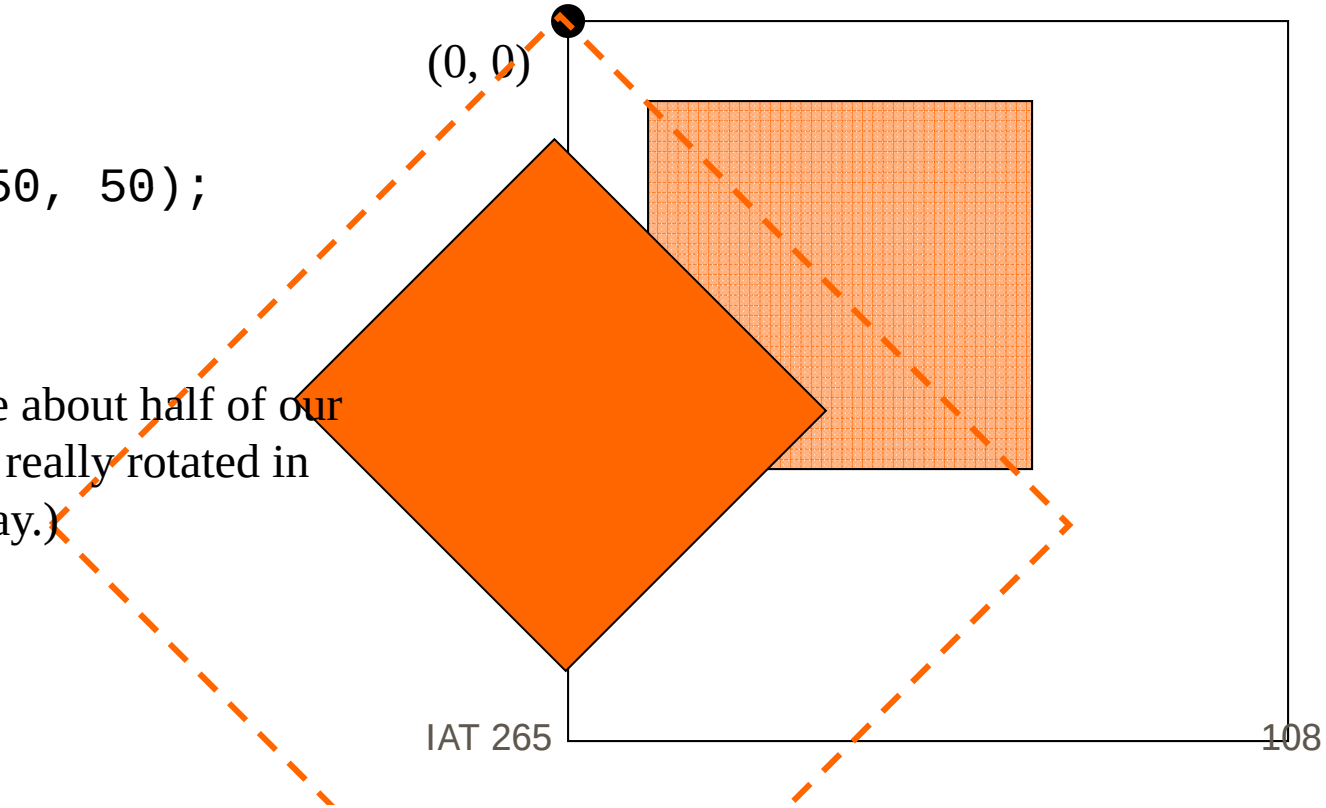
```
rotate(angle);
```



Rotation

- Draw the same square, now on our rotated coordinate system

```
rect(10, 10, 50, 50);
```



(We only get to see about half of our square, and it isn't really rotated in any satisfactory way.)

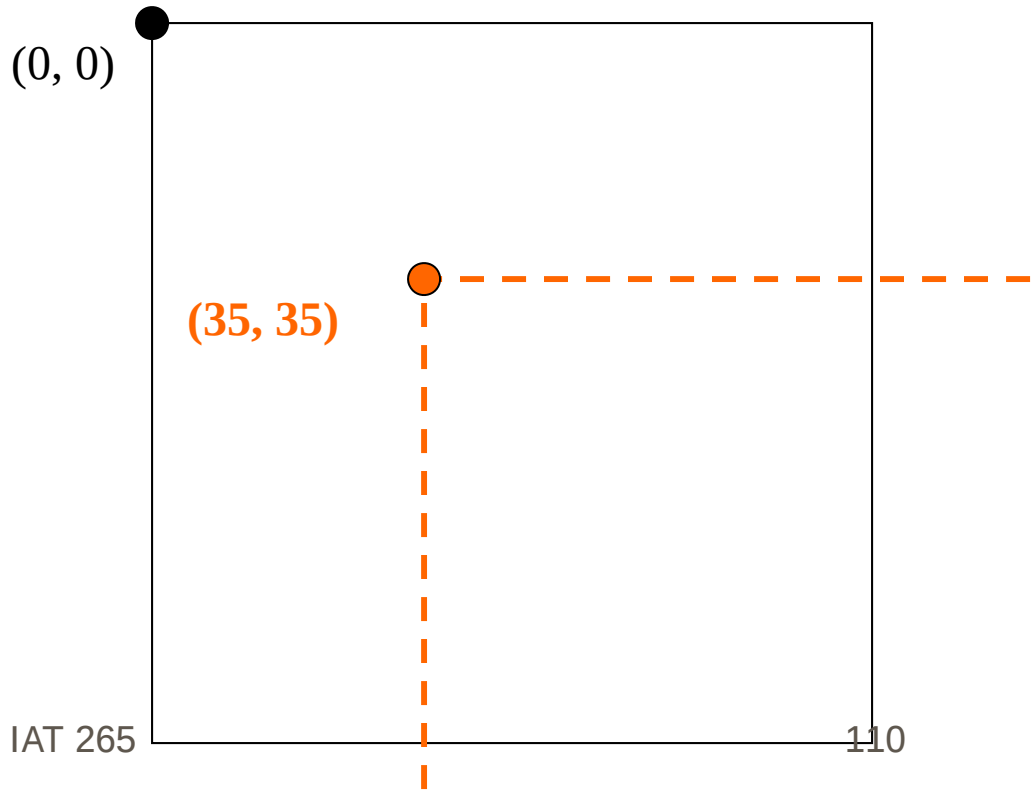
Rotation

- Let's try this from the start, using translation.
- Where should we translate to?
  - The point **around** which we want to rotate. So let's try and rotate around the center of the square.
  - This means moving the origin, and drawing the square around it.

Rotation

- Let's start with setting our rotation point:

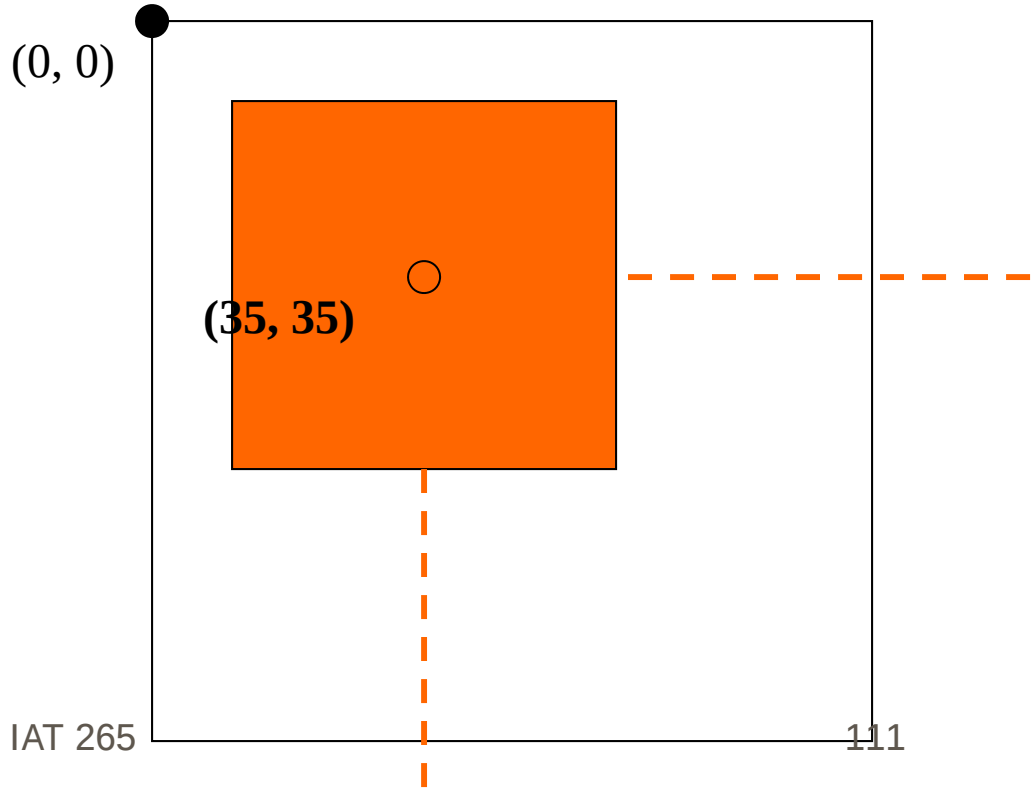
```
translate(35, 35);
```



Rotation

- Now let's draw a square with this point at its center.

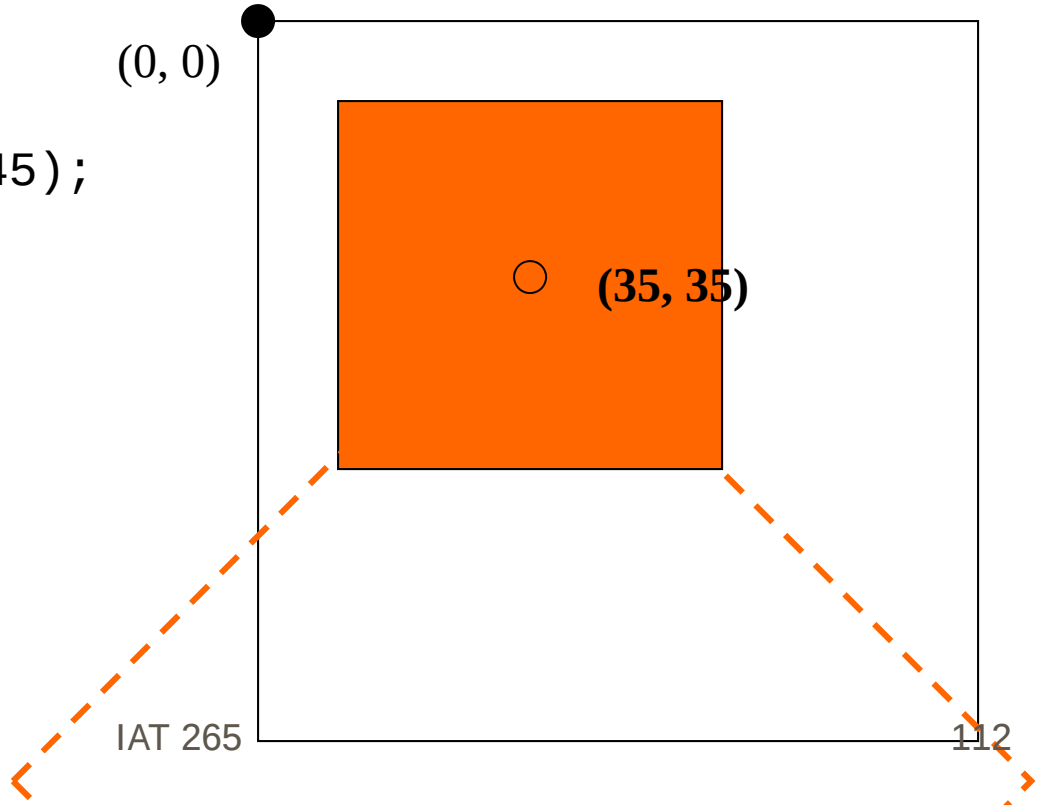
```
rect( -25, -25, 50, 50);
```



Rotation

- Then let's do the same rotate we did last time.

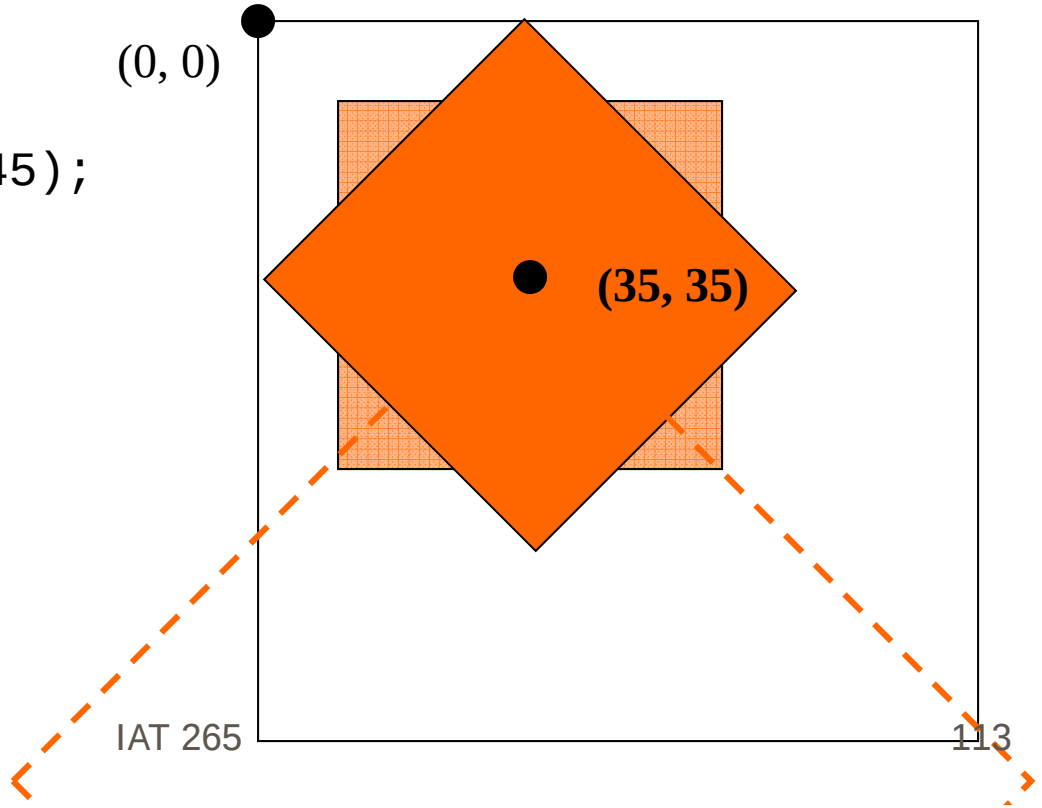
```
float angle = radians(45);
rotate(angle);
```



Rotation

- Now when we draw the same square as before, it will have the same center point.

```
float angle = radians(45);
rotate(angle);
```

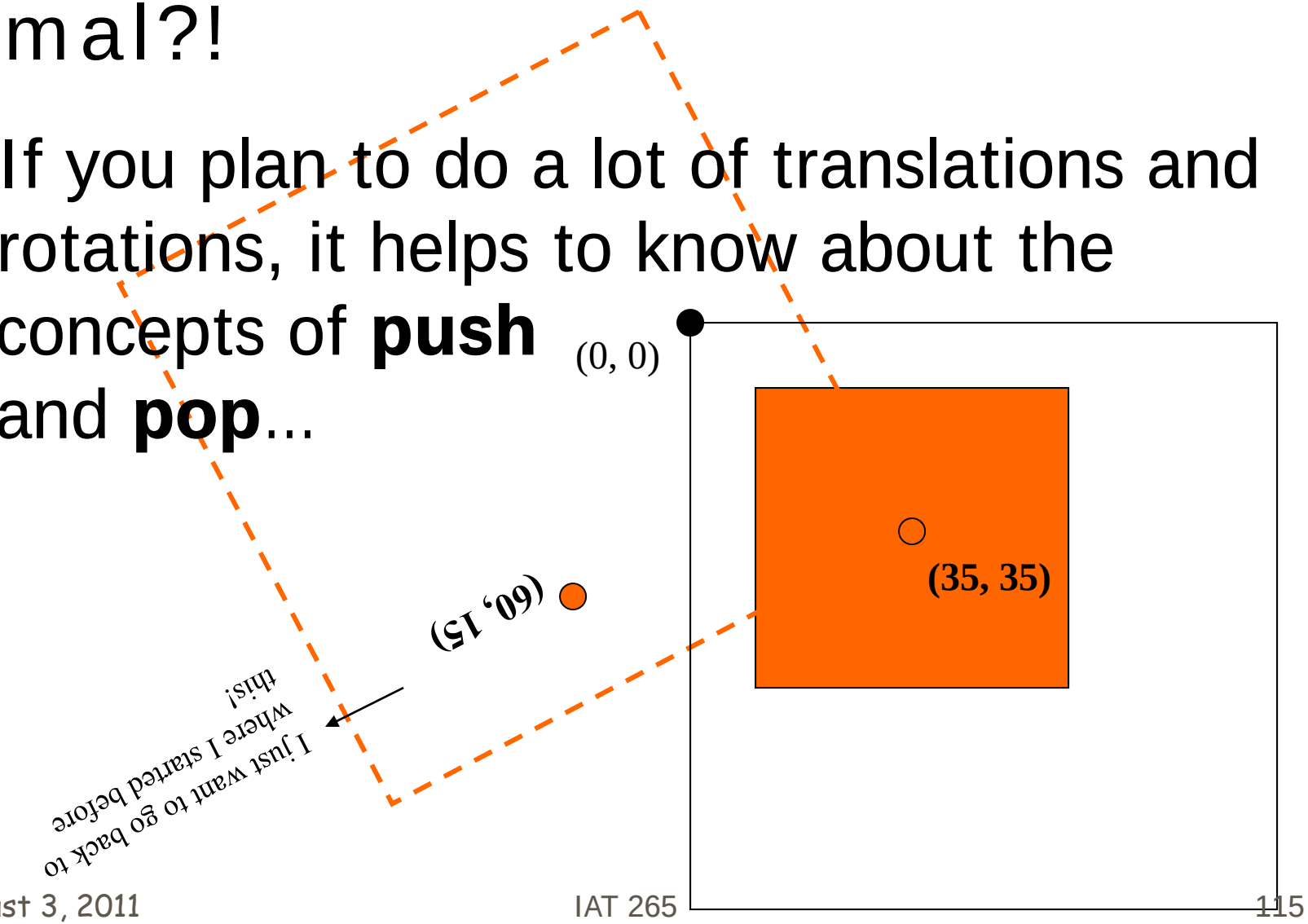


Rotation

- Try applying rotation to your animations using `draw()`. What variable will you want to iterate to make a shape rotate over time?
- Try making a custom polygon rotate instead of a square.

Wait! How do I get back to normal?!

- If you plan to do a lot of translations and rotations, it helps to know about the concepts of **push** and **pop**...



Pushing and Popping

- Pushing is a way to say:

  "Remember this orientation!"

  `pushMatrix();`

- Popping is a way to say:

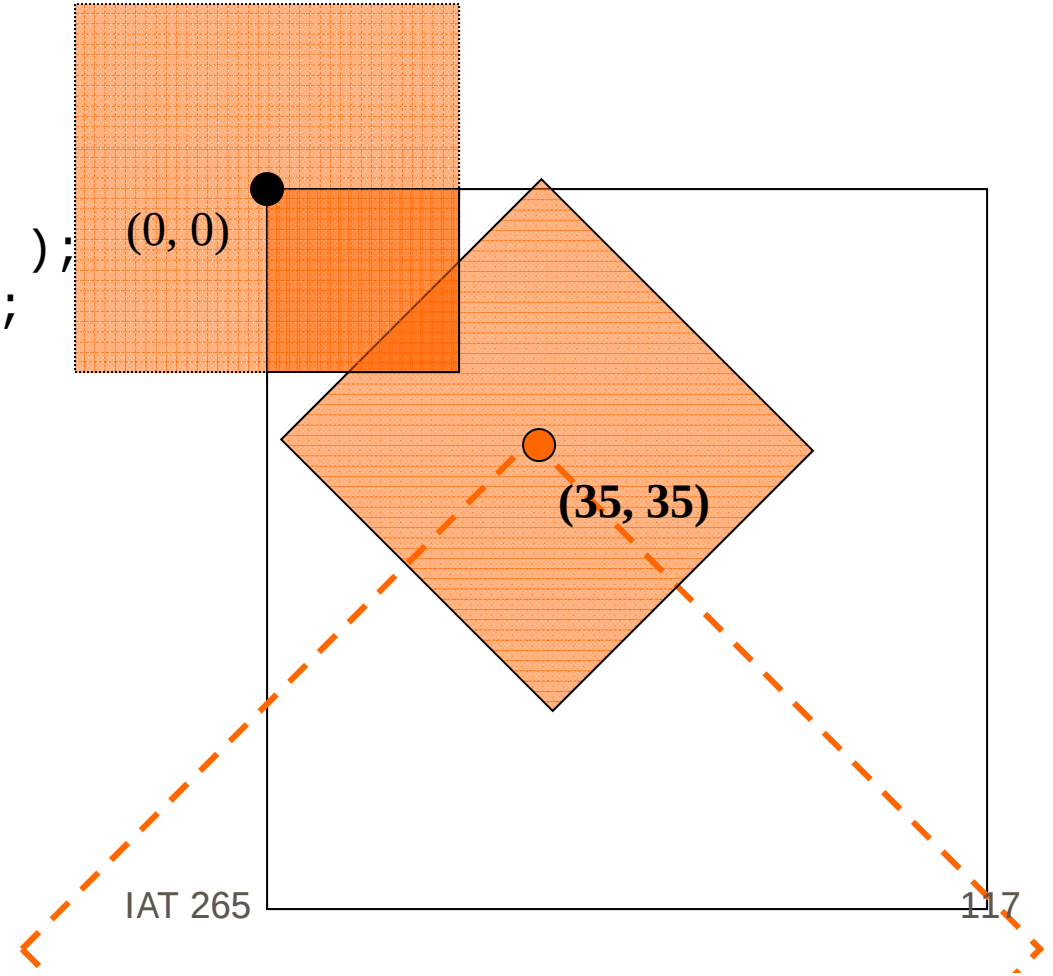
  "Take me back to the way things once were!"

  `popMatrix();`

Push & Pop

- If we want to remember the original orientation...

```
pushMatrix();
 translate(35,35);
 rotate( radians(45) );
 rect(-25,-25,50,50);
popMatrix();
rect(-25,-25,50,50);
```



You can push and pop as many times as you want. It's like you're writing *an address for the way things were* on a card, and putting it on a stack each time you **push**… and **pop** just takes the first card off the top of the stack.

Stack

pushMatrix() and popMatrix() control a **stack**

A **stack** stores chunks of data like a stack of plates in a cafeteria

Last-In, First-out **(LIFO)**

The graphics **matrix stack** lets you pushMatrix(), draw stuff, popMatrix()

Returns drawing to state before pushMatrix

2D Arrays

- Java allows us to make Array of Arrays – otherwise called 2D Array

```
int[][] bob = new int[3][4];
color[][] pixels2d = new color[200][200];
```

- However, Processing doesn't provide us with a 2D array of pixels to use

2D Arrays

- Interestingly, 2D Arrays are just covering up a 1D array

- The 2D array

```
color[][] pixels2d = new color[10][20];
color c2 = pixels2d[3][2];
```

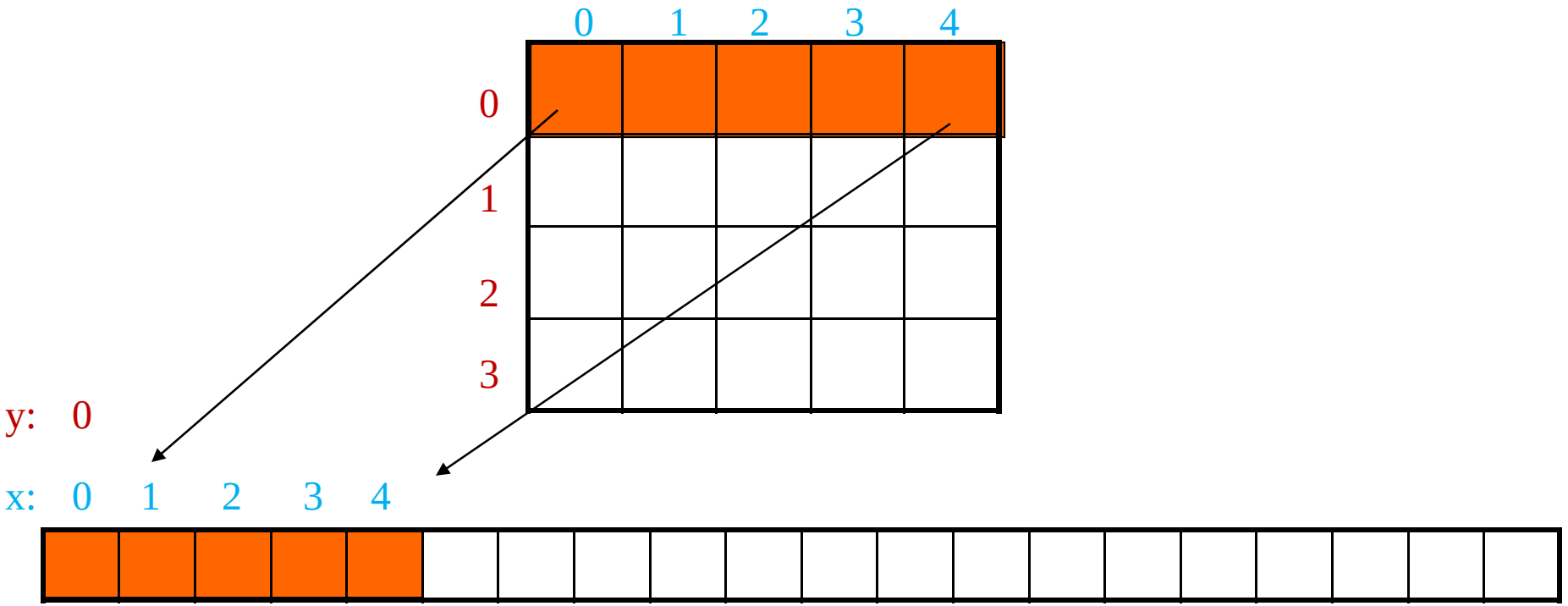
- is equivalent to:

```
color[] pixels1d = new color[200];
color c1 = pixels1d[3 + 2*10];
```

Underneath, these two pieces of code do the same thing. Computer graphics, however, normally uses 1D array to store pixels of an image – pixels[]

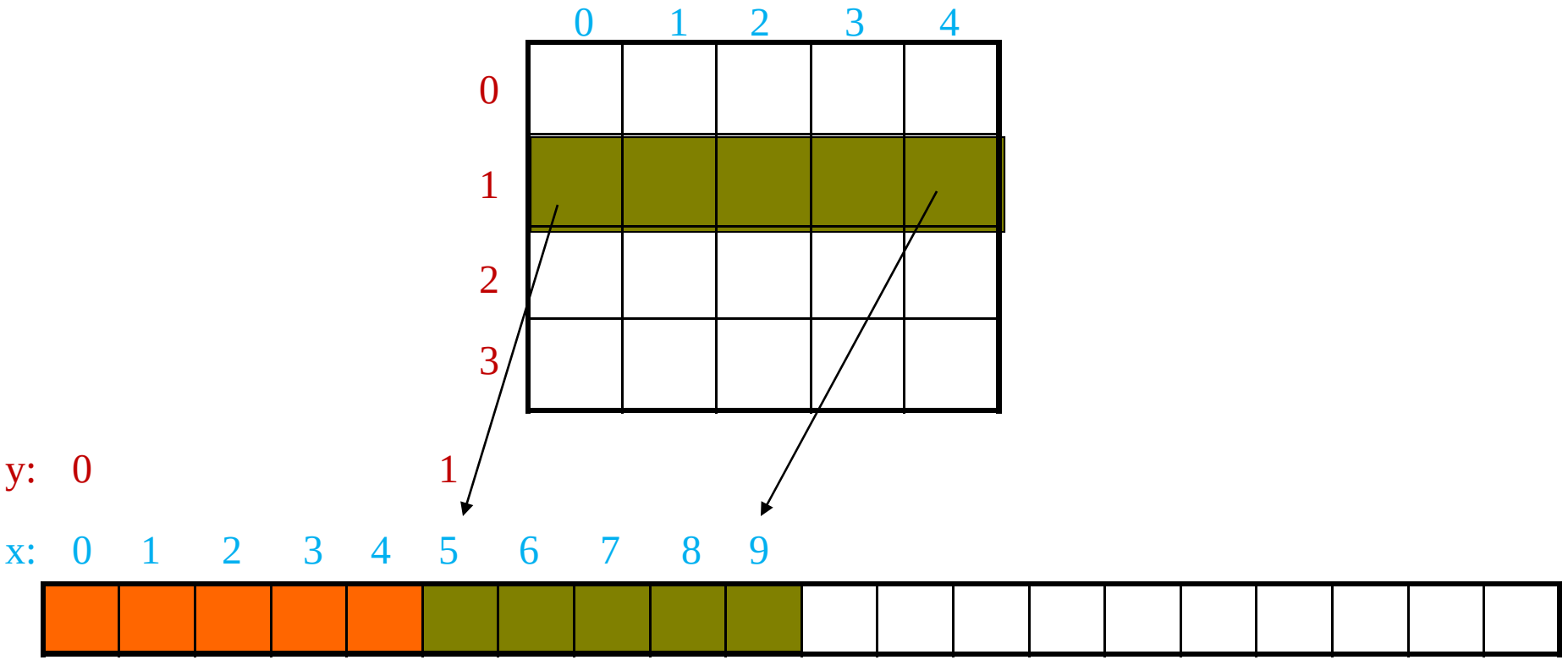
2D matrix converted into 1D array

- How do we map pixels of a 2D image into a 1D `pixels[]` array?



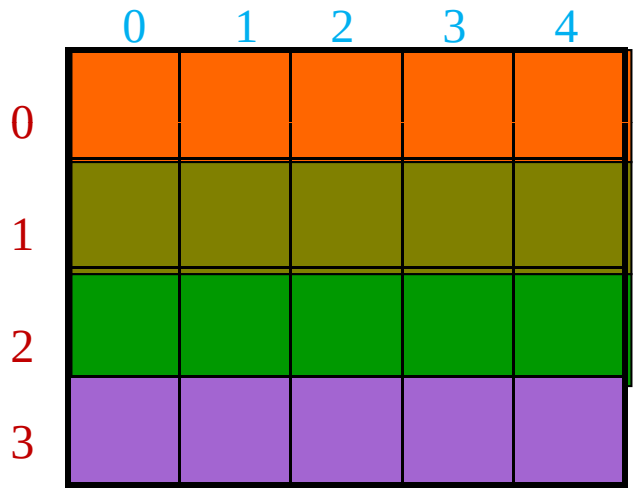
2D matrix converted into 1D array

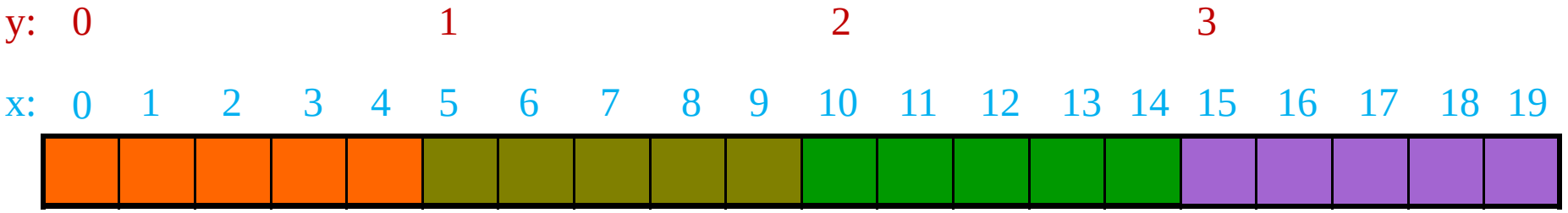
- Its 2nd row goes into 2nd segment of the pixels[] array



2D matrix converted into 1D array

- The same for 3rd and 4th rows...





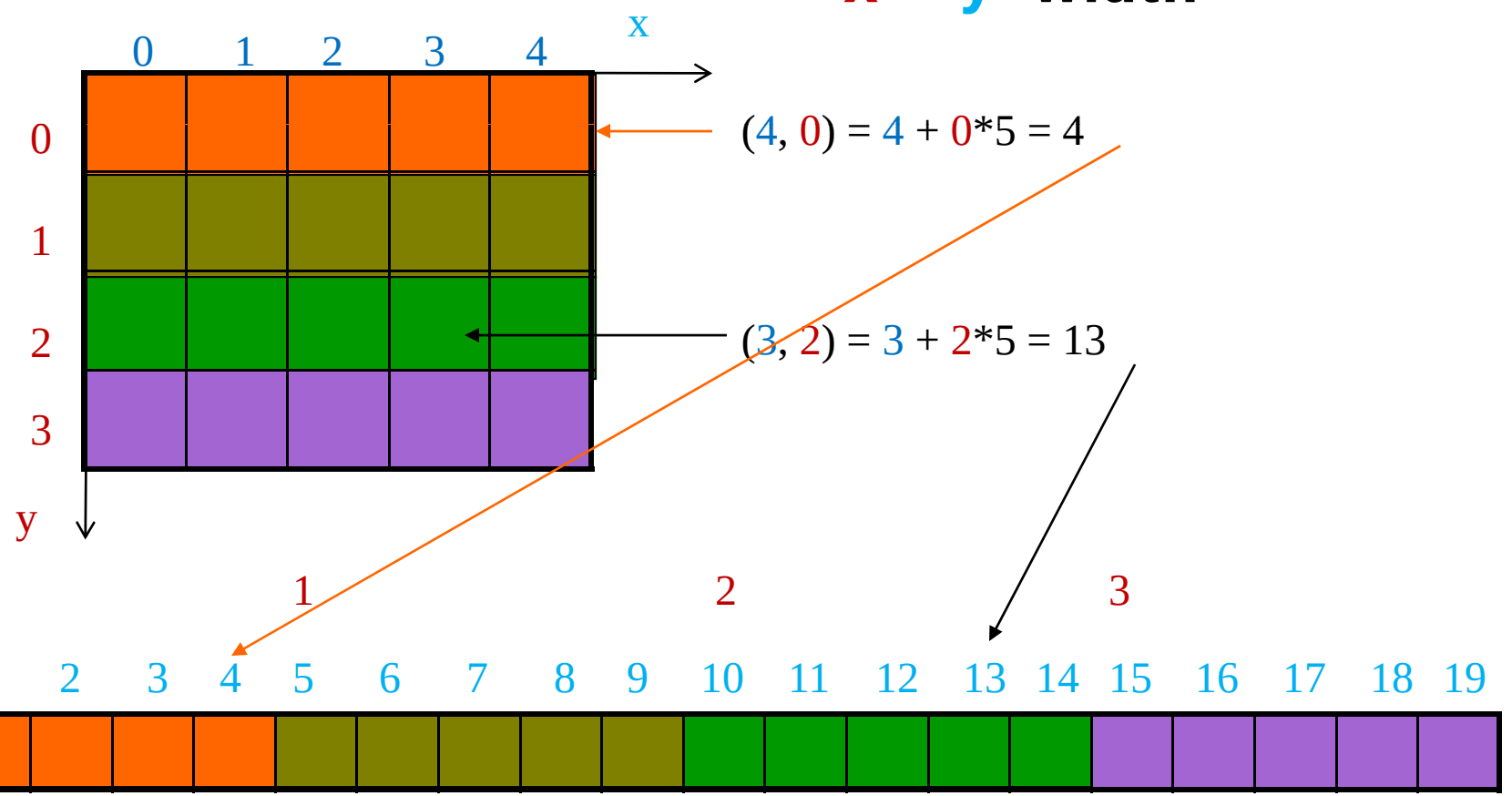
Accessing Pixels

- The **`PImage`** object allows you to access each of its pixels (color values) with the **`pixels[]`** array

- You can get the width and height of the image using the **`width`** and **`height`** **fields** of `PImage`

Accessing Pixels

- Calculate array Index:
  - **x + y*width**



Accessing Pixels

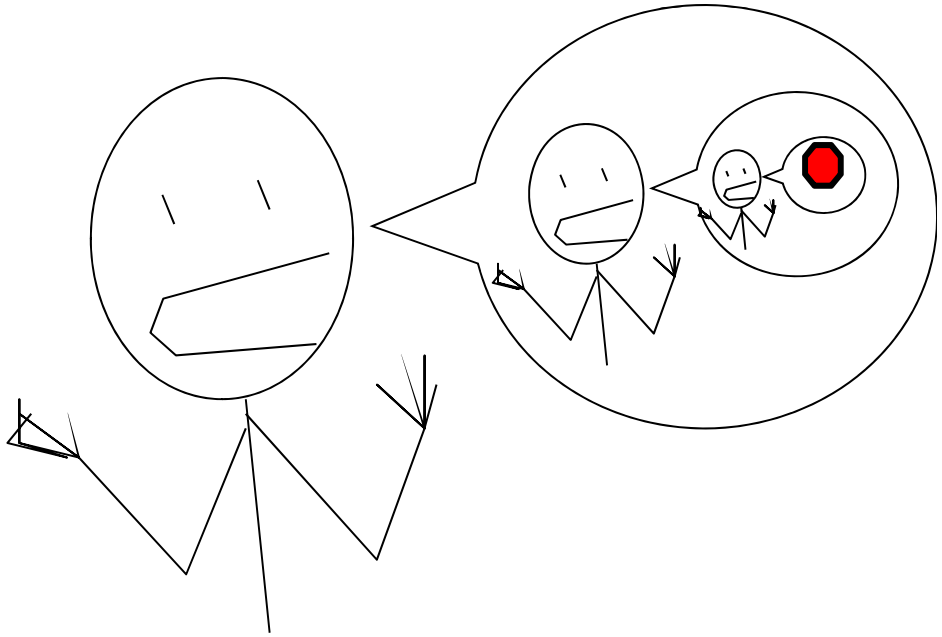
- Now we know the array index, how to get the color from a pixel?

```
PImage img = loadImage("face.jpg");
image(img, 50, 50);
//get color at (3, 2)
color c1 = img.pixels[15 + 10*img.width];
// set our lines' color
stroke(c1);
line(50, 150, 80, 180);
line(80, 180, 110, 140);
```

Recursion

- Recursion basically means a method calling itself

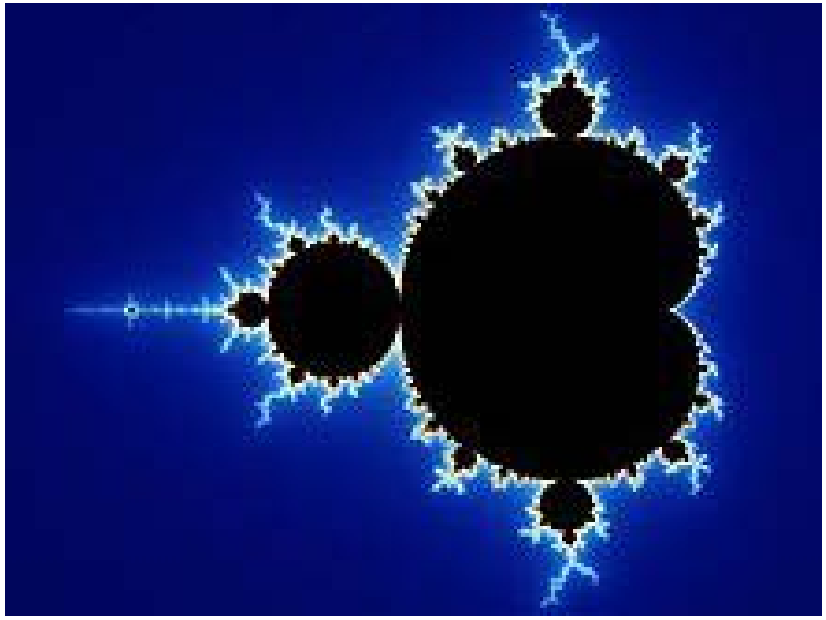
```
int factorial(int n)
{
  if( n > 1 )
  {
    return(  n* factorial( n-1 )  );
  }
  else
    return( 1 );
}
```



Base Case

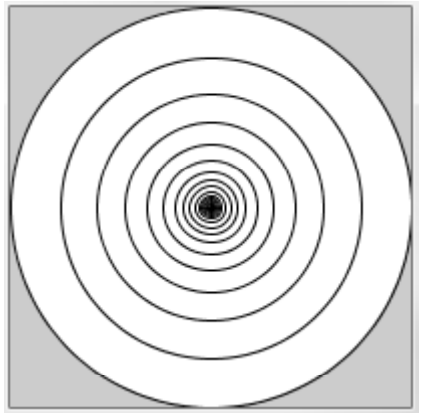
- Must have at least one **Base Case**
  - A case or condition that returns without further recursion
  - Stops the recursive chain
  - Eg factorial(int n)
    - Returned 1 when n = 1
    - In every other call, n decreases by 1

Recursion is a good instrument for generating fractals – parts are a reduced version of the whole

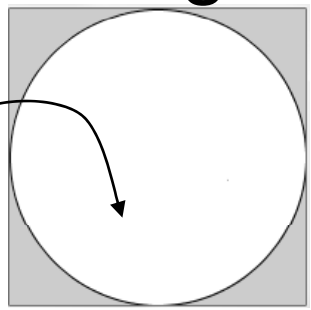


A simple example

```
void drawCircle (float x, float y, float d) { //d - diameter
    ellipse(x, y, d, d );

    if (d  > 2) {
          d *= 0.75; //shrink d by 25% each recursion
          drawCircle (x, y, d );
    }
}

void setup(){
  size(200, 200);
  drawCircle (width/2, height/2, width); //drawCircle(100, 100, 200);
}
```



Let's step through what happens



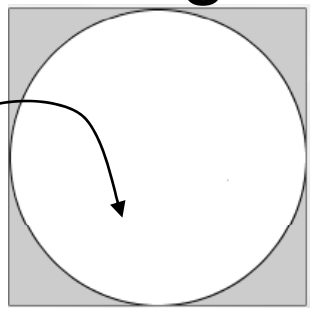
```
drawCircle(100, 100, 200);
```

```
void drawCircle (float x, float
y, float d) (d=200)
{
   ellipse(x, y, d, d);

   if (d > 10) {
      d *= 0. 5;
      drawCircle(x, y, d);
   }
}
```

```
void drawCircle (float x, float y,
float d) (d=100)
{
   ellipse(x, y, d, d);

   if (d > 10) {
      d *= 0. 5;
      drawCircle(x, y, d);
   }
}
```

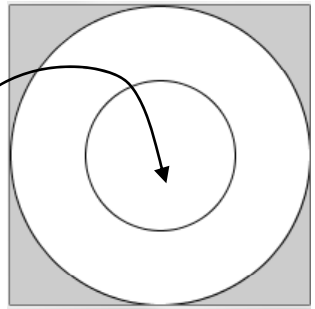
Let's step through what happens

```
drawCircle(100, 100, 200);
```



```
void drawCircle (float x, float
y, float d) (d=200)
{
   ellipse(x, y, d, d);

   if (d > 10) {
      d *= 0. 5;
      drawCircle(x, y, d);
   }
}
```

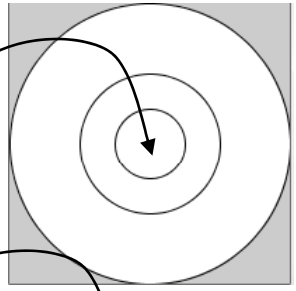
```
void drawCircle (float x, float
y, float d) (d=100)
{
   ellipse(x, y, d, d);

   if (d > 10) {
      d *= 0. 5;
      drawCircle(x, y, d);
   }
}
```



```
void drawCircle (float x, float
y, float d) (d=50)
{
   ellipse(x, y, d, d);

   if (d > 10) {
      d *= 0. 5;
      drawCircle(x, y, d);
   }
}
```

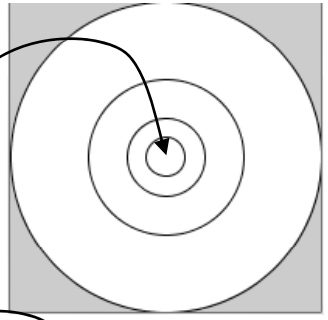
Let's step through what happens

```
drawCircle(100, 100, 200);
```



```
void drawCircle (float x, float
y, float d) (d=50)
{
   ellipse(x, y, d, d);

   if (d > 10) {
      d *= 0. 5;
      drawCircle(x, y, d);
   }
}
```

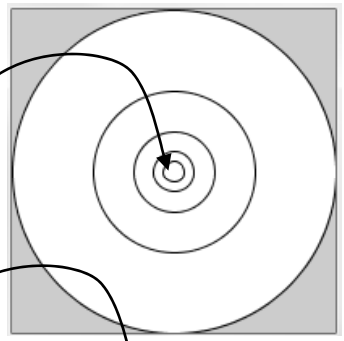
```
void drawCircle (float x, float
y, float d) (d=25)
{
   ellipse(x, y, d, d);

   if (d > 10) {
      d *= 0. 5;
      drawCircle(x, y, d);
   }
}
```



```
void drawCircle (float x, float
y, float d) (d=12.5)
{
   ellipse(x, y, d, d);

   if (d > 10) {
      d *= 0. 5;
      drawCircle(x, y, d);
   }
}
```

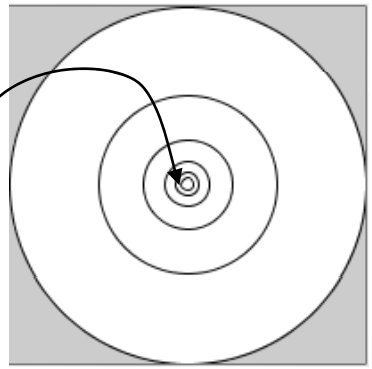
Let's step through what happens

```
drawCircle(100, 100, 200);
```



```
void drawCircle (float x, float
y, float d) (d=12.5)
{
   ellipse(x, y, d, d);

   if (d > 10) {
      d *= 0. 5;
      drawCircle(x, y, d);
   }
}
```

```
void drawCircle (float x, float
y, float d) (d=6.25)
{
   ellipse(x, y, d, d);

   if (d > 10) {
      d *= 0. 5;
      drawCircle(x, y, d);
   }
}
```



d = 6.25 < 10 → base case hit, so stop recursive call, start to return

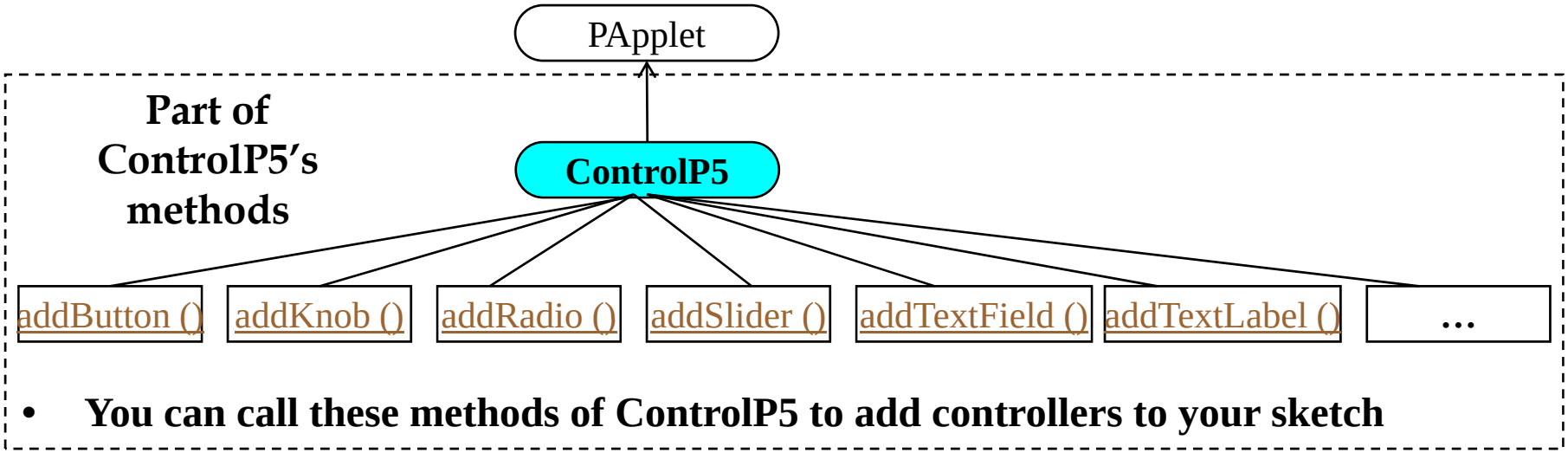
Challenge of GUI Programming

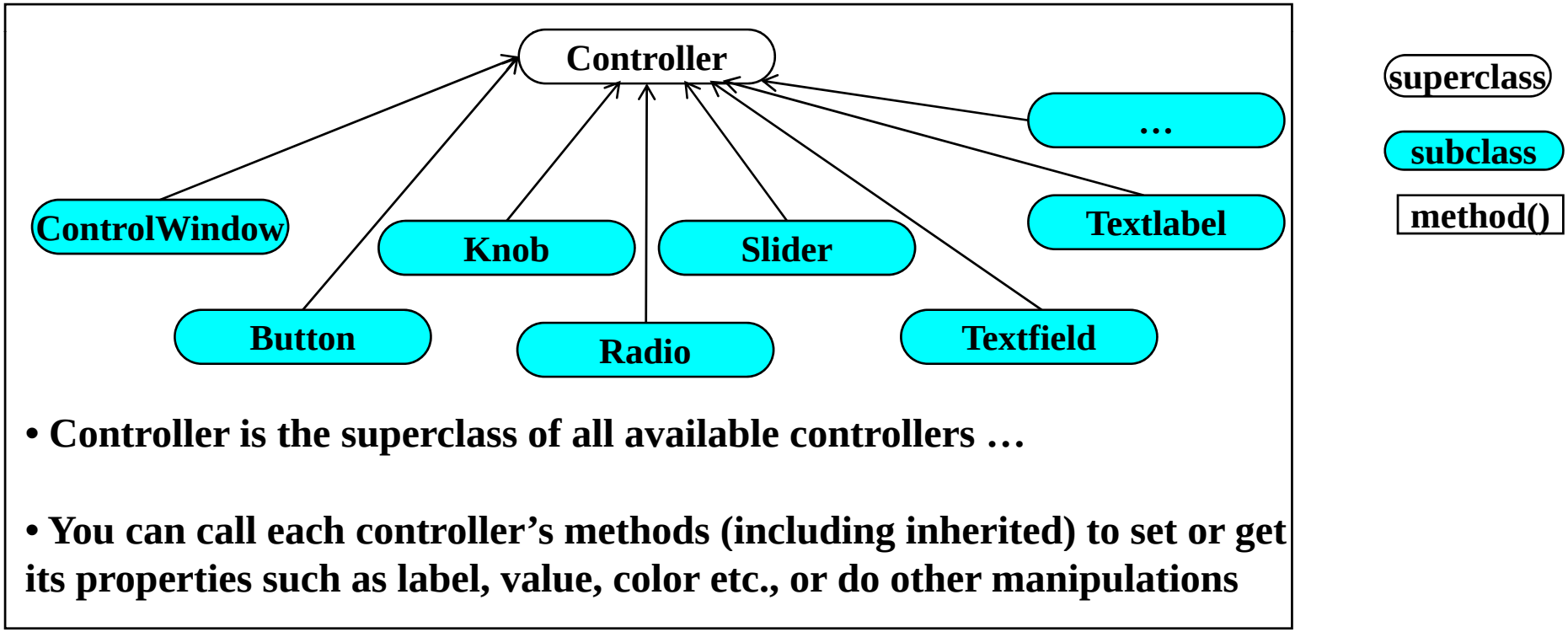
- The Challenge lies mainly in the need to dynamically change user interface (based on events) at runtime

- It can be tackled by a program design pattern named: **event-driven programming**

Event-Driven Programming: Architecture

- Program waits for *events* to occur and then responds, which is divided down to three sections:
  - ***Event firing*** – objects/interactions generate events
  - ***Event detection*** – listeners check for events
    - Normally taken care of by programming frameworks
  - ***Event handling*** – functions respond to events
    - Programmers need to define these functions (aka event-handlers), typically they are ***callbacks***


PApplet
Part of ControlP5's methods
ControlP5
addButton ()
addKnob ()
addRadio ()
addSlider ()
addTextField ()
addTextLabel ()
…
• You can call these methods of ControlP5 to add controllers to your sketch

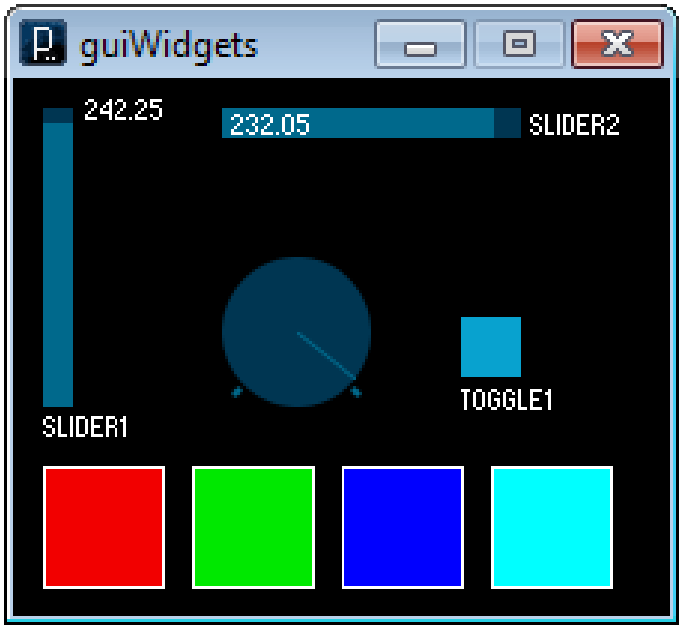

Controller
ControlWindow
Knob
Slider
…
Textlabel
Button
Radio
Textfield
superclass
subclass
method()
• Controller is the superclass of all available controllers …
• You can call each controller's methods (including inherited) to set or get its properties such as label, value, color etc., or do other manipulations

Handle Events from Multiple Controllers

- Modify the colors in the color array with different controllers, and display them in the rectangles

This will require you to:

1) **add those controllers to the window**
2) **handle events fired from these controllers**
3) **respond by changing the color squares as per which controller is changed**



Implementation

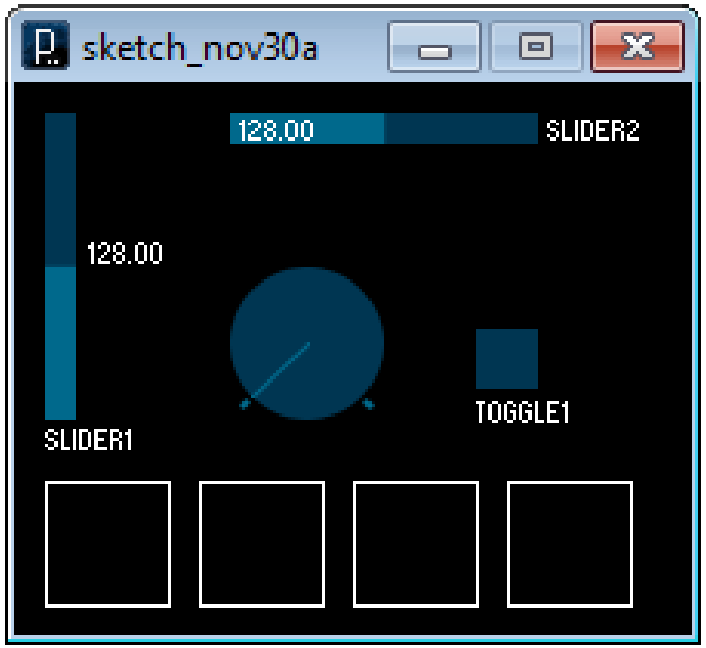
2. Add the controllers

Can you tell which is horizontal or vertical?

```
void setup() {
  …    //whatever was done so far
 //Add Sliders. Parameters: name, minimum, maximum,
 //default value (float), x, y, width, height
  controlP5.addSlider("slider1",0,255,128,10,10,10,100);
  controlP5.addSlider("slider2",0,255,128,70,10,100,10);

  //Add a Knob. Parameters: name, minimum, maximum,
  //default value (float), x, y, diameter
  controlP5.addKnob("knob1",0,360,0,70,60,50);

  //Add a toggle which have two states: true or false
  //parameters: name, default value (boolean), x, y,
  //width, height
  controlP5.addToggle("toggle1",false,150,80,20,20);
}
```

Run the sketch and this is what you got so far...

- The controllers are all in place, and visually functional, however it brings no change to squares' color. Why?



Handle Events from Multiple Controllers

- That's because we haven't created any code to handle events fired from those controllers when you manipulate them

- This is what we need to do:
  - add the event handler: ***controlEvent(theEvent)***
  - Inside it, use multiple ***if-statements***, one for each controller, to detect which has been changed
  - Then respond by changing the corresponding square's color accordingly

Implementation

3. Add the event handler: ***controlEvent(theEvent)***

```
 …     //whatever is done so far
void controlEvent(ControlEvent theEvent) {
   if(theEvent.controller().name()=="slider1")
      colors[0] = color(theEvent.controller().value(),0,0);

   if(theEvent.controller().name()=="slider2")
      colors[1] = color(0,theEvent.controller().value(),0);

   if(theEvent.controller().name()=="knob1")
      colors[2] = color(0,0,theEvent.controller().value());

   if(theEvent.controller().name()=="toggle1") {
      if(theEvent.controller().value()==1)
         colors[3] = color(0,255,255); //set color to cyan
      else
         colors[3] = color(0,0,0);     //otherwise black
   }
}
```

Debugging

How do I know my program is broken?

- Compiler Errors
  - Errors the compiler can catch – easy to fix
- Runtime Exceptions
  - Errors the runtime environment can catch – more difficult to fix
- Your program just doesn't do the right thing
  - The trickiest one to fix!!

Compiler Errors

- Syntax errors:
  - Maybe missed a bracket/brace/semicolon or other necessary syntax element – easy to fix
- Logical errors – could be tricky for newbie
  - Method call inconsistent with method's signature
  - Access a variable beyond its scope
  - No return statement in a method with a return type
  - Put a non-boolean method in an *if*-statement

Logical Errors

Method call inconsistent with method's signature

- Signature: *Bug(float x, float y, float chgX, float chgY, float sz)*
- Wrong:
  - bugs.add(new Bug());
  - AvatarBug(float x, float y, float chgX, float chgY, float sz) {
    super();
    }
- Right:
  - bugs.add(new Bug(random(width), random(height), random(-1,1),random(-1,1),random(12,36)));
  - AvatarBug(float x, float y, float chgX, float chgY, float sz) {
    super(x, y, chgX, chgY, sz);

Logical Errors (1)

Access a variable beyond its scope

- The "**scope**" of a variable refers to the variable's visibility within a program
  - **Global variables**: accessible from anywhere in the program
  - **Fields** (class member variables): accessible from anywhere in the class that they are declared
  - **Local variables**: declared within a method and accessible only within the method

- A common error: access local variables outside the method that they are declared

Logical Errors (2)

An Example:

- Declare a variable (e.g. *avtBug*) within one method:

```
void setup() {
   AvatarBug avtBug = respawn();
}
```

- then try to access it in another method:

```
void playGame() {
   if (rightKey) avtBug.moveRight();
}
```

- To fix:
  - Declare *avtBug* as a global variable (i.e. declare it outside any method and class)

Logical Errors (3)

No return statement in a method with a return type

- The following method returns boolean but no return statement as the last statement inside it

```
boolean detectCollision(Bug otherBug) {
    if ( abs(bugX-otherBug.bugX)<(bSize+otherBug.bSize)  &&...) {
      return true;
    }
   //??????????????
  }
```

- To fix:

```
boolean detectCollision(Bug otherBug) {
    if ( abs(bugX-otherBug.bugX)<(bSize+otherBug.bSize) && ...) {
      return true;
    }
    return false;
```

Logical Errors (4)

- Or a better way to avoid this issue:

```
boolean detectCollision(Bug otherBug) {
   boolean hit = false;
    if ( abs(bugX-otherBug.bugX)<(bSize+otherBug.bSize) && ...) {
      hit = true;
    }
   return hit;
}
```

Logical Errors (5)

Put a non-boolean method in an *if*-statement

- Wrong:

```
void detectBound() {
  if (bugX+bSize > width ) {
    bugX = width-bSize;
  }
  if (bugX-bSize < 0 ) {
    bugX = bSize;
 }}
```

```
void draw() {
  …
 if( bugi.detectBound() ){
     changeX *= -1;
  }
  …
 }
```

- To fix: either keep the signature, change the way to use it

```
void detectBound() {
  if (bugX+bSize > width ) {
    bugX = width-bSize;
    changeX *= -1;
  }
  if (bugX-bSize < 0 ) {
    bugX = bSize;
    changeX *= -1;
 }}
```

```
void draw() {
   …
   bugi.detectBound();
   …
 }
```

Logical Errors (6)

- or keep the way of using it, change the signature and the logic inside:

```
boolean detectBound() {
   boolean hit = false;
   if (bugX+bSize > width ) {
    bugX = width-bSize;
    hit = true;
   }
   if (bugX-bSize < 0 ) {
     bugX = bSize;
    hit = true;
  }
  return hit;
}
```

```
void draw() {
 …
 if( bugi.detectBound() ){
     changeX *= -1;
  }

  …
 }
```

Runtime Exceptions

Common Runtime Exceptions:

- NullPointerException and ArrayIndexOutOfBoundsException

- Exceptions caused by semantic errors
  - uninitialized variable, bad loop logic, ...

NullPointerException

- Normally because you're calling an object's method when you failed to instantiate it
  - So the variable for the object becomes a pointer pointing to NULL

```
AvatarBug avtBug;
//failed to instantiate it like:
//avtBug = new AvatarBug(width/2, height/2, 4,4,20);
//but try to call its method like:
avtBug.drawBug(); → NullPointerException!!
```

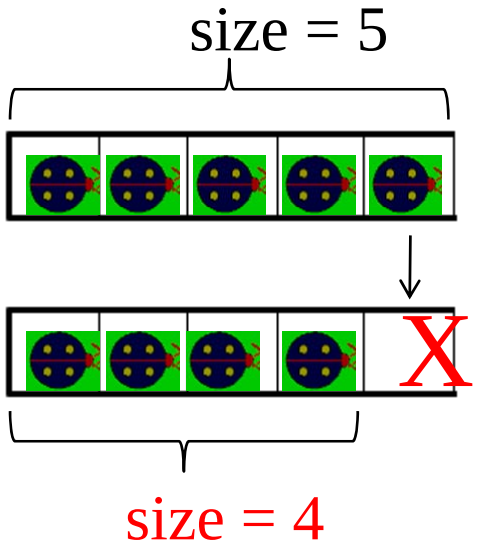
- To fix: Always instantiate after declaring an object!!

ArrayIndexOutOfBoundsException

- Normally because you try to access an array index that goes beyond the max length of an array
  - It becomes tricky esp. when you try to use an constant control value to access a resizable array like ArrayList

```
int count = 5;
for(int i=0; i< count; i++) {
    Bug bugi = (Bug) bugs.get(i);
    ...
```

- To fix: use *bugs.size() as controller*



Data Structures

- **Data structure** is a particular way of storing and organizing data in computer so that it can be used efficiently
  - Data structures are generally based on the ability of a computer to fetch and store data at any place in its memory, specified by an address

- Common data structures include: array, linked list, B-tree, hash-table, heap, ...

Operations on Data Structure

- With a collection of data, we often want to do many things
  - Store and organize data
  - Go through the list of data and do **X** per item
  - Add new data
  - Delete unwanted data
  - Search for data of some value
    - “Give me Smith’s phone number”

What's Important

- Runtime is important
- If you're only going to do it once or twice
  - Use any old algorithm that will work
- If you're going to run it millions of times
  - Think about the algorithm!

Runtime of Sorting

- Bubble sort: since it used nested loops:
  - For each iteration of outer loop, inner loop run n times
  - The total runtime: n*n → O (n^2)
  - This is very slow!!

- Questions for review: What sorting algorithm is the fastest known so far? How fast is it?

Data Structures for runtime reduction

- Some are built to enable fast searching
- Some enable fast insertion/deletion

What	Tree	HashTable
Add	O(lgN+)	O(1)
Delete	O(lgN+)	O(1)
Search	O(lgN)	O(1)

- Questions for review:

  Why add/delete takes longer time than search for tree?

  Why HashTable takes constant time for the operations?

  What strategies are normally employed for collision resolution when using HashTable?