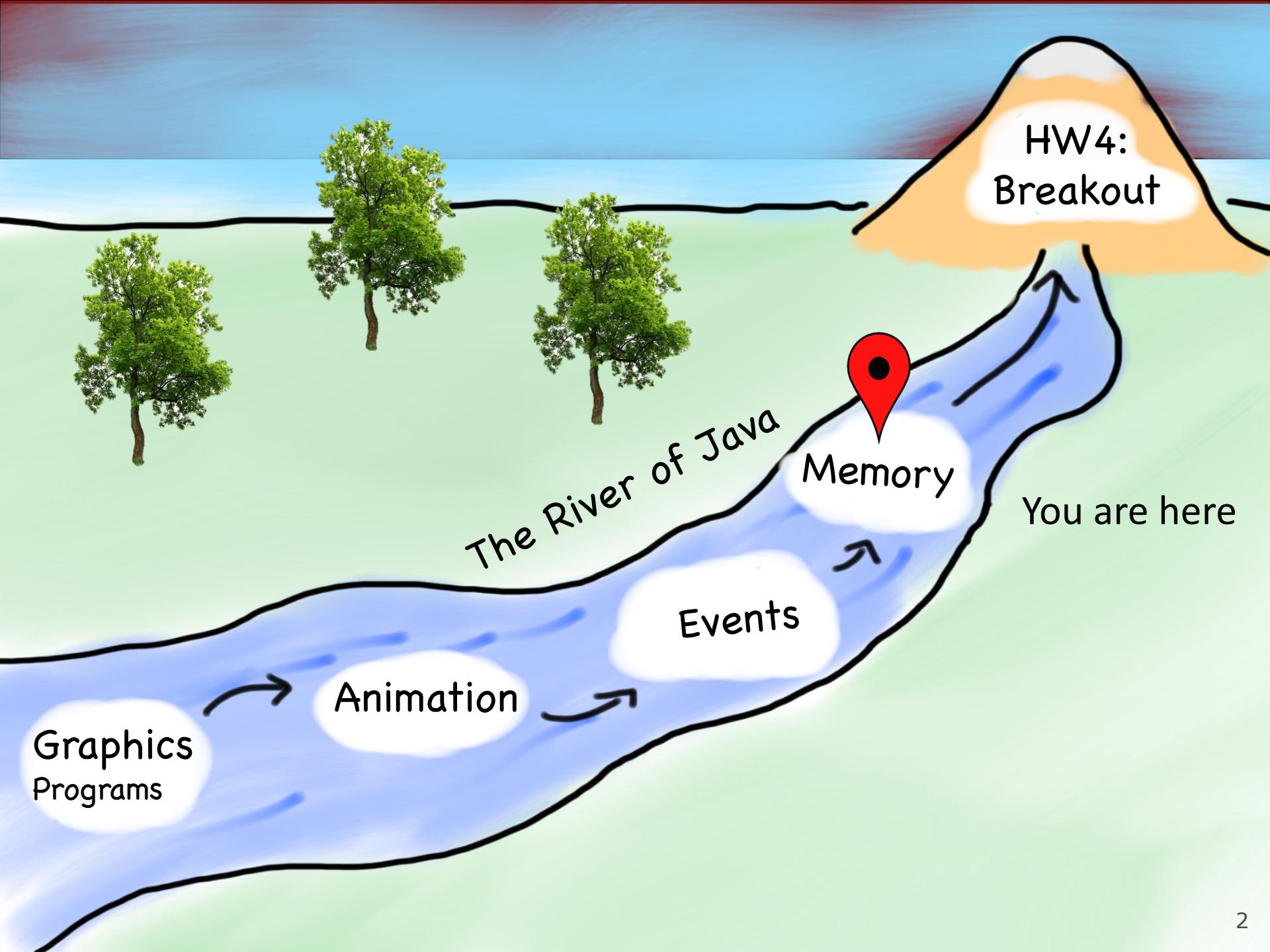


# **CS 106A, Lecture 15**

# **Events and Memory**



# Plan for Today

- Announcements
- Review: Null, Events and Instance Variables
- Extra Practice: Googley Eyes
- A Boolean Aside
- Memory
- Revisiting Whack-A-Mole

# Plan for Today

- Announcements
- Review: Null, Events and Instance Variables
- Extra Practice: Googley Eyes
- A Boolean Aside
- Memory
- Revisiting Whack-A-Mole

# Plan for Today

- Announcements
- Review: Null, Events and Instance Variables
- Extra Practice: Googley Eyes
- A Boolean Aside
- Memory
- Revisiting Whack-A-Mole

# Null

**Null** is a special variable value that objects can have that means “nothing”. Primitives cannot be null.

If a method returns an object, it can return **null** to signify “nothing”. (just say **return null;**)

```
// may be a Gobject, or null if nothing at (x, y)  
GObject maybeAnObject = getElementAt(x, y);
```

Objects have the value **null** before being initialized.

```
Scanner myScanner; // initially null
```

# Null

You can check if something is null using == and !=.

```
// may be a GObject, or null if nothing at (x, y)
GObject maybeAnObject = getElementAt(x, y);
if (maybeAnObject != null) {
    // do something with maybeAnObject
} else {
    // null - nothing at that location
}
```

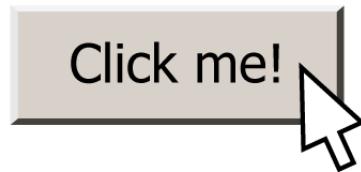
# Null

Calling methods on an object that is **null** will crash your program!

```
// may be a GObject, or null if nothing at (x, y)
GObject maybeAnObject = getElementAt(x, y);
if (maybeAnObject != null) {
    int x = maybeAnObject.getX(); // OK
} else {
    int x = maybeAnObject.getX(); // CRASH!
}
```

# Events

- **event**: Some external stimulus that your program can respond to.



- **event-driven programming**: A coding style (common in graphical programs) where your code is executed in response to user events.

# Events

```
public void run() {  
    // Java runs this when program launches  
}  
  
public void mouseClicked(MouseEvent event) {  
    // Java runs this when mouse is clicked  
}  
  
public void mouseMoved(MouseEvent event) {  
    // Java runs this when mouse is moved  
}
```

# Types of Mouse Events

- There are many different types of mouse events.

- Each takes the form:

```
public void eventMethodName(MouseEvent event) { ... }
```

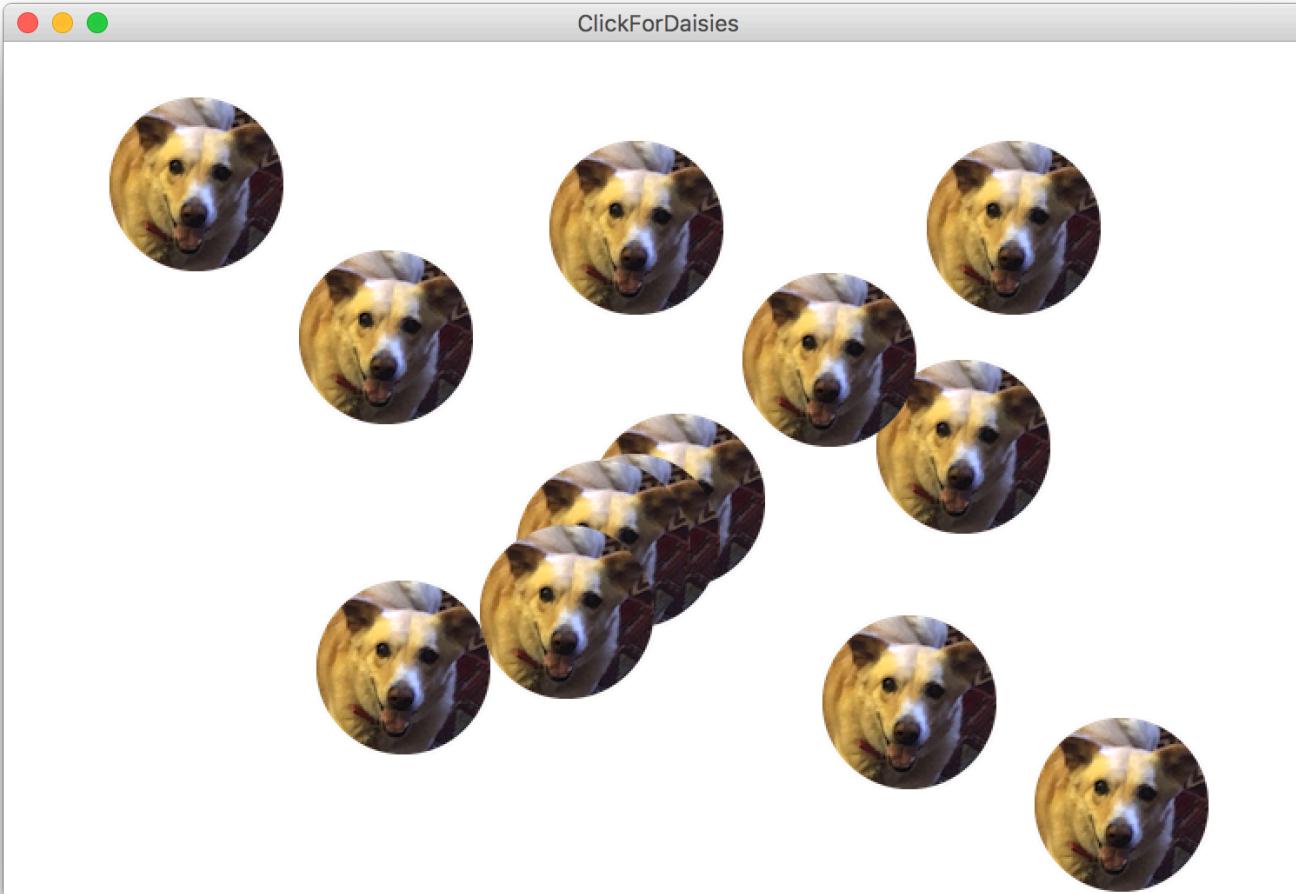
Method	Description
mouseMoved	mouse cursor moves
mouseDragged	mouse cursor moves while button is held down
mousePressed	mouse button is pressed down
mouseReleased	mouse button is lifted up
mouseClicked	mouse button is pressed and then released
mouseEntered	mouse cursor enters your program's window
mouseExited	mouse cursor leaves your program's window

# MouseEvent Objects

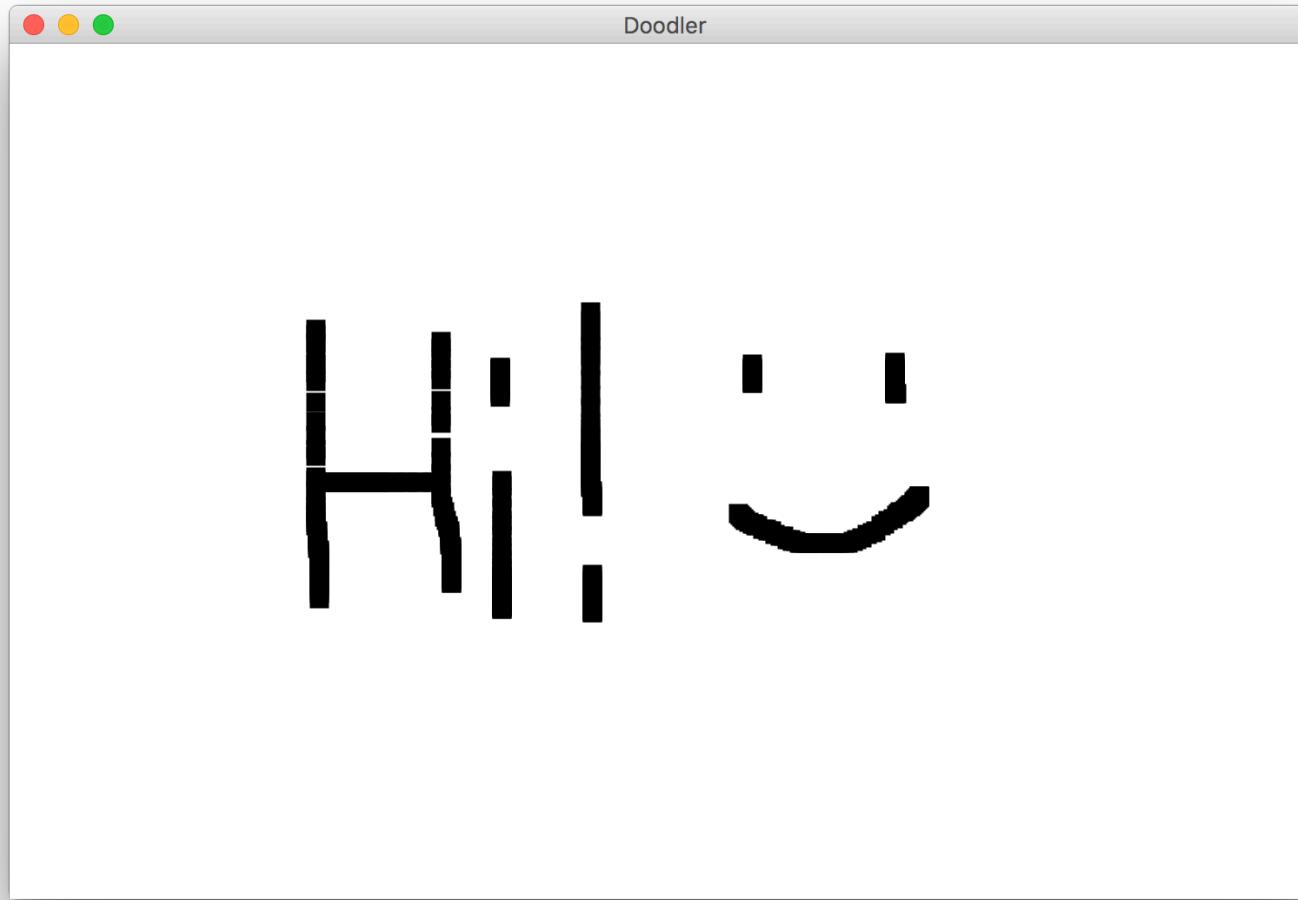
- A MouseEvent contains information about the event that just occurred:

Method	Description
<code>e.getX()</code>	the x-coordinate of mouse cursor in the window
<code>e.getY()</code>	the y-coordinate of mouse cursor in the window

# Example: ClickForDaisies



# Example: Doodler



# Doodler

```
public void mouseDragged(MouseEvent event) {  
    double mouseX = event.getX();  
    double mouseY = event.getY();  
    double rectX = mouseX - SIZE / 2.0;  
    double rectY = mouseY - SIZE / 2.0;  
    GRect rect = new GRect(rectX, rectY, SIZE, SIZE);  
    rect.setFilled(true);  
    add(rect);  
}
```

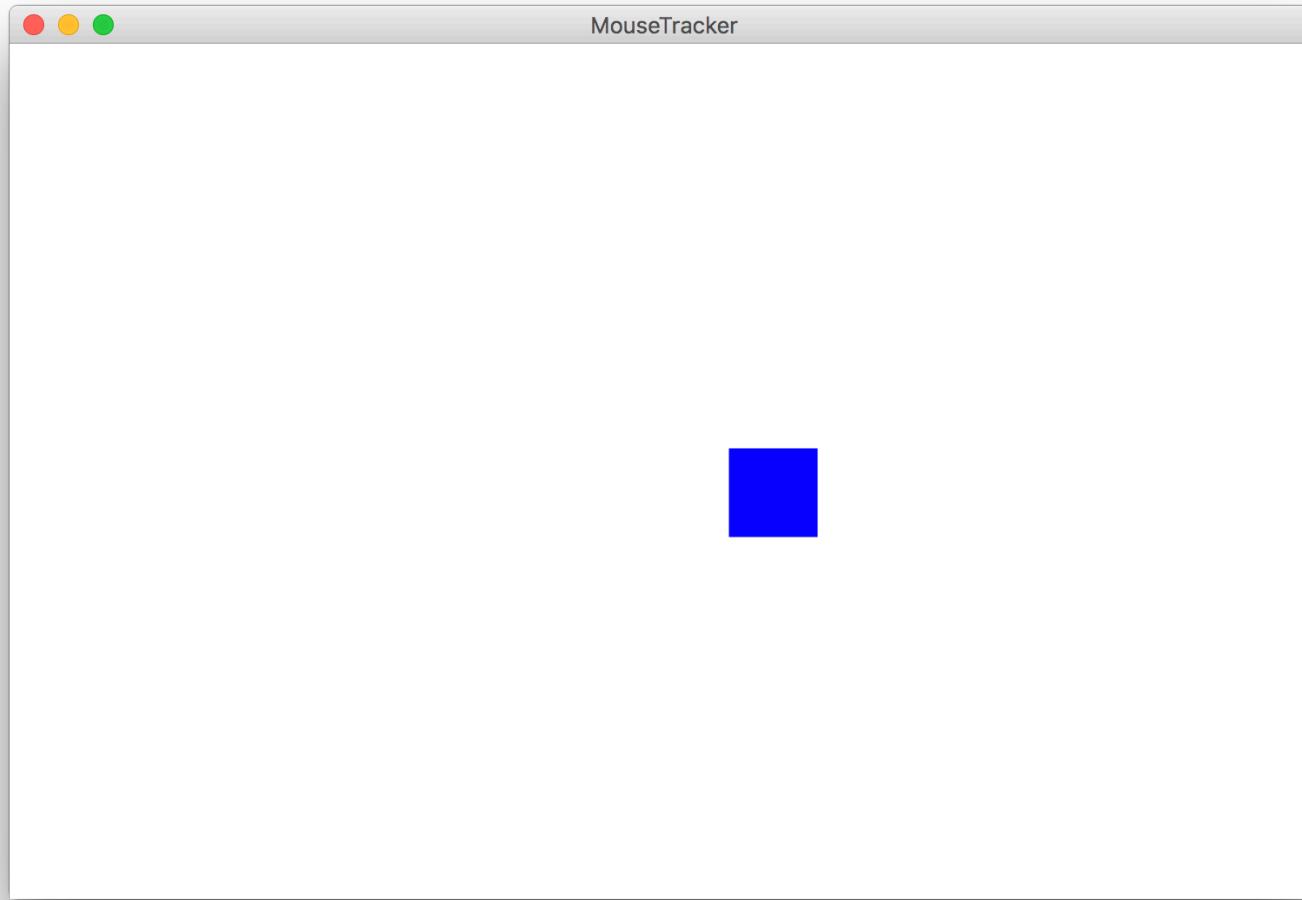
What if we wanted the *same* GRect to track the mouse, instead of making a new one each time?

# Instance Variables

```
private type name; // declared outside of any method
```

- **Instance variable:** A variable that lives outside of any method.
  - The *scope* of an instance variable is throughout an entire file (class).
  - Useful for data that must persist throughout the program, or that cannot be stored as local variables or parameters (event handlers).
  - *It is bad style to overuse instance variables*

# Example: MouseTracker



# Whack-A-Mole

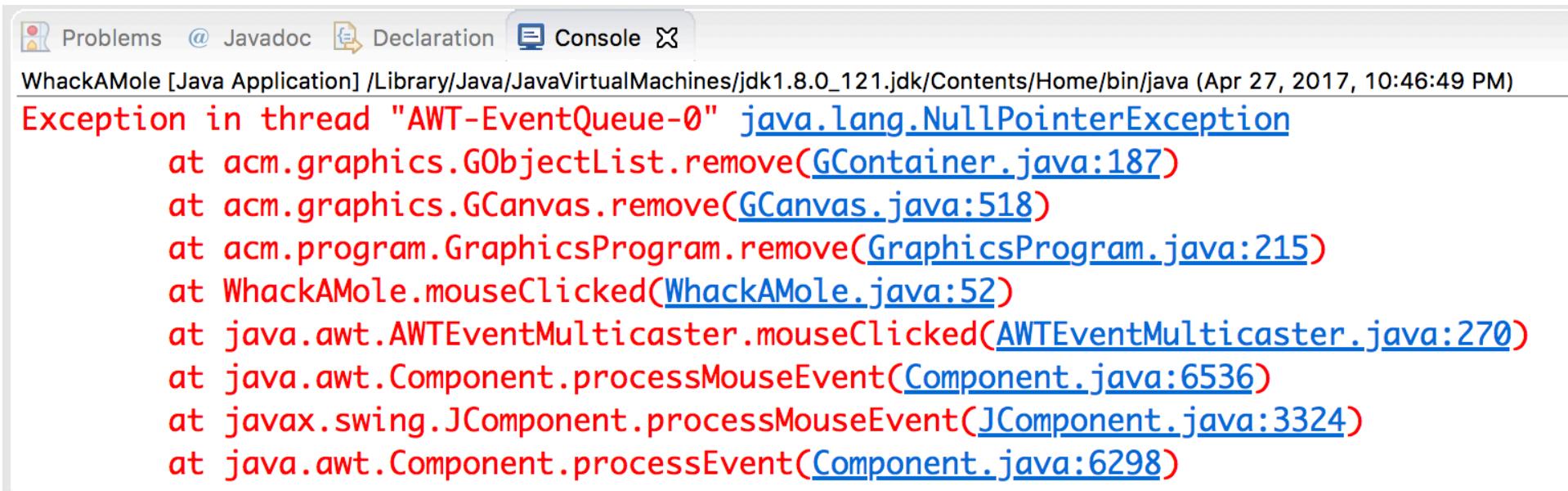
We used instance variables and events to make Whack-A-Mole!

- A mole should appear every second at a random location, and stop once the user has gotten at least 10 points.
- If the user clicks a mole, remove it and increase their score by 1
- There should be a GLabel in the left corner showing their score



# Exception

- If the user clicks an area with no mole, the program crashes.
  - A program crash in Java is called an **exception**.
  - When you get an exception, Eclipse shows red error text.
  - The error text shows the line number where the error occurred.
  - Why did this error happen?
  - How can we avoid this?



The screenshot shows the Eclipse IDE interface with the 'Console' tab selected. The console output displays a stack trace for a `java.lang.NullPointerException` that occurred in the thread "AWT-EventQueue-0". The stack trace lists the following method calls:

```
Exception in thread "AWT-EventQueue-0" java.lang.NullPointerException
    at acm.graphics.GObjectList.remove(GContainer.java:187)
    at acm.graphics.GCanvas.remove(GCanvas.java:518)
    at acm.program.GraphicsProgram.remove(GraphicsProgram.java:215)
    at WhackAMole.mouseClicked(WhackAMole.java:52)
    at java.awt.AWTEventMulticaster.mouseClicked(AWTEventMulticaster.java:270)
    at java.awt.Component.processMouseEvent(Component.java:6536)
    at javax.swing.JComponent.processMouseEvent(JComponent.java:3324)
    at java.awt.Component.processEvent(Component.java:6298)
```

# Whack-A-Null?

```
public void mouseClicked(MouseEvent event) {  
    double mouseX = event.getX();  
    double mouseY = event.getY();  
    GObject mole = getElementAt(mouseX, mouseY);  
  
    remove(mole);  
    score++;  
    scoreLabel.setText("Score: " + score);  
}
```

# Whack-A-Null?

```
public void mouseClicked(MouseEvent event) {  
    double mouseX = event.getX();  
    double mouseY = event.getY();  
    GObject mole = getElementAt(mouseX, mouseY);  
  
    remove(mole);  
    score++;  
    scoreLabel.setText("Score: " + score);  
}
```

Problem: **mole** may be null if the user doesn't click on a mole! Removing null will crash ☹

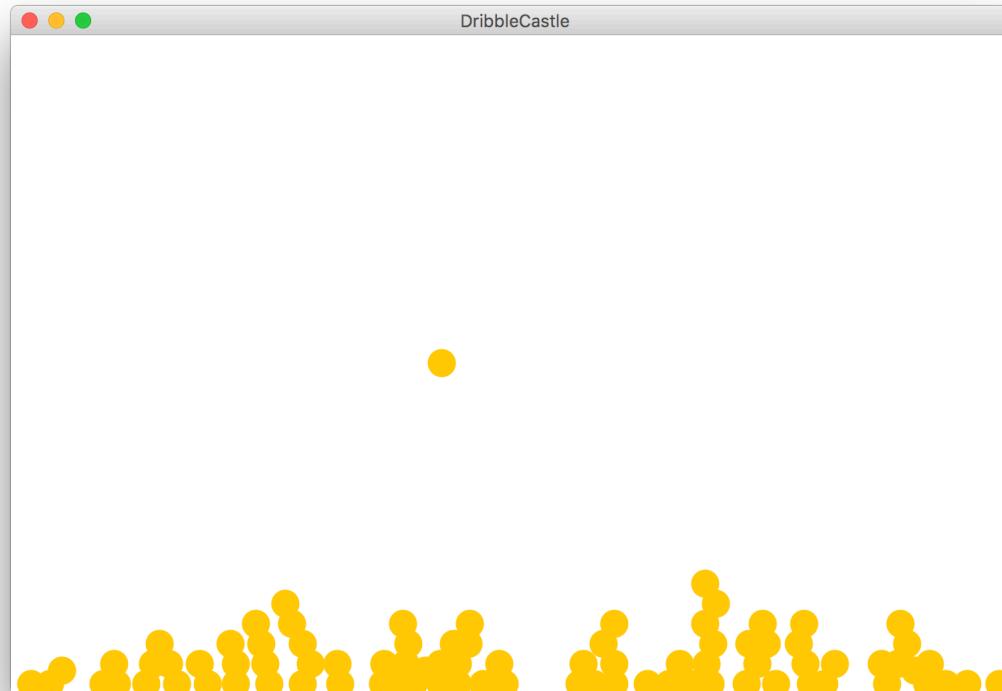
# Whack-A-Mole!

```
public void mouseClicked(MouseEvent event) {  
    double mouseX = event.getX();  
    double mouseY = event.getY();  
    GObject mole = getElementAt(mouseX, mouseY);  
  
    if (mole != null) {  
        remove(mole);  
        score++;  
        scoreLabel.setText("Score: " + score);  
    }  
}
```

# Revisiting DribbleCastle

We wrote a program called **DribbleCastle** that simulates wet sand falling and forming cool shapes.

But we left out one part: how do we stop the animation if one dribble collides with another?



# DribbleCastle

```
private void dropOneDribble() {  
    GOval dribble = makeDribble();  
    add(dribble);  
  
    while (!hasHitBottom(dribble) &&  
          !hasHitSomethingElse(dribble)) {  
        dribble.move(0, Y_VELOCITY);  
        pause(PAUSE_TIME);  
    }  
}
```

# DribbleCastle

```
private void dropOneDribble() {  
    GOval dribble = makeDribble();  
    add(dribble);  
  
    while (!hasHitBottom(dribble) &&  
          !hasHitSomethingElse(dribble)) {  
        dribble.move(0, Y_VELOCITY);  
        pause(PAUSE_TIME);  
    }  
}
```

# DribbleCastle

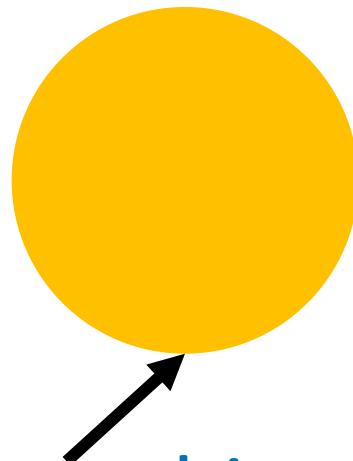
```
private void dropOneDribble() {  
    GOval dribble = makeDribble();  
    add(dribble);  
  
    while (!hasHitBottom(dribble) &&  
          !hasHitSomethingElse(dribble)) {  
        dribble.move(0, Y_VELOCITY);  
        pause(PAUSE_TIME);  
    }  
}
```

# hasHitSomethingElse

```
private boolean hasHitSomethingElse(GOval dribble) {  
    double checkX = dribble.getX()  
        + dribble.getWidth() / 2.0;  
    double checkY = dribble.getY()  
        + dribble.getHeight() + 1;  
    GObject hit = getElementAt(checkX, checkY);  
    return hit != null;  
}
```

# hasHitSomethingElse

```
private boolean hasHitSomethingElse(GOval dribble) {  
    double checkX = dribble.getX()  
        + dribble.getWidth() / 2.0;  
    double checkY = dribble.getY()  
        + dribble.getHeight() + 1;  
    GObject hit = getElementAt(checkX, checkY);  
    return hit != null;  
}
```



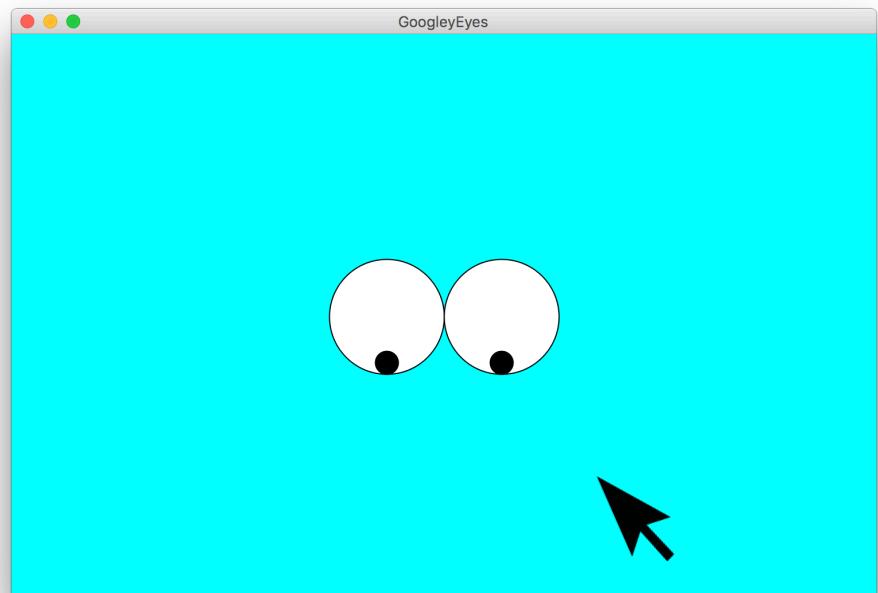
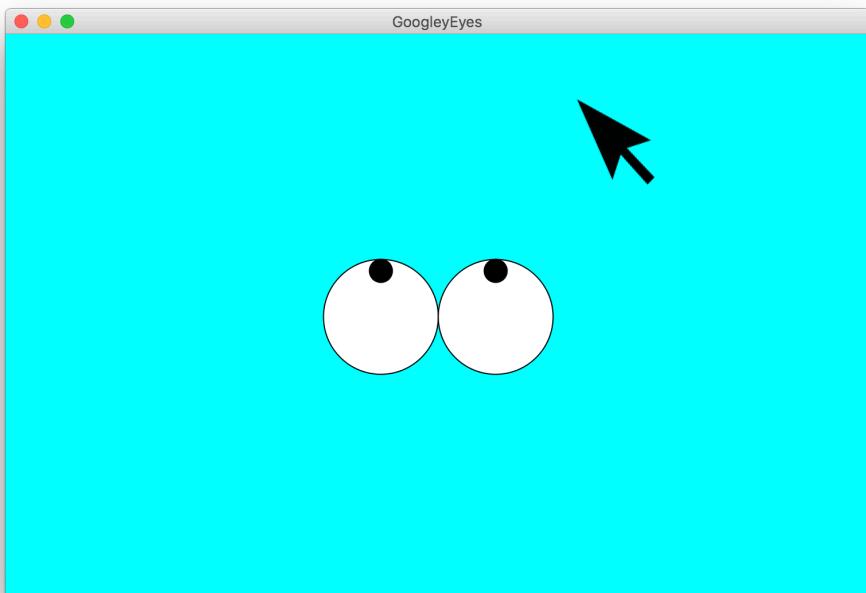
Check if something is here

# Plan for Today

- Announcements
- Review: Null, Events and Instance Variables
- Extra Practice: Googley Eyes
- A Boolean Aside
- Memory
- Revisiting Whack-A-Mole

# Extra Practice: GoogleyEyes

Let's write a program called **GoogleyEyes** that draws eyes whose pupils follow the user's mouse vertically as it moves around.



# Plan for Today

- Announcements
- Review: Null, Events and Instance Variables
- Extra Practice: Googley Eyes
- A Boolean Aside
- Memory
- Revisiting Whack-A-Mole

# A Boolean Aside

There is one helpful property of how Java evaluates boolean expressions, called **short-circuit evaluation**.

When evaluating boolean expressions, Java only evaluates *as much of the expression as it needs to* in order to evaluate it to be true or false.

# A Boolean Aside

```
String str = readLine("? ");

if (str.length() > 0 && str.charAt(0) == 'A') {

    ...
}
```

# A Boolean Aside

```
String str = readLine("? "); // what about ""!  
  
if (str.length() > 0 && str.charAt(0) == 'A') {  
    ...  
}
```

# A Boolean Aside

```
String str = readLine("? "); // what about ""!  
  
if (str.length() > 0 && str.charAt(0) == 'A') {  
    ...  
}
```

# A Boolean Aside

```
String str = readLine("? "); // what about ""!  
  
if (str.length() > 0 && str.charAt(0) == 'A') {  
    ...  
}
```

Java only executes this if  
the first part is **true!** This  
means it never crashes.

# A Boolean Aside

```
GObject obj = getElementAt(x, y);  
  
if (obj == null || obj.getX() == 0) {  
    ...  
}
```

# A Boolean Aside

```
GObject obj = getElementAt(x, y); // what about null!  
  
if (obj == null || obj.getX() == 0) {  
    ...  
}
```

# A Boolean Aside

```
GObject obj = getElementAt(x, y); // what about null!  
  
if (obj == null || obj.getX() == 0) {  
    ...  
}
```

# A Boolean Aside

```
GObject obj = getElementAt(x, y); // what about null!  
  
if (obj == null || obj.getX() == 0) {  
    ...  
}
```

Java only executes this if  
the first part is **false!** This  
means it never crashes.

# Plan for Today

- Announcements
- Review: Null, Events and Instance Variables
- Extra Practice: Googley Eyes
- A Boolean Aside
- **Memory**
- Revisiting Whack-A-Mole

# A Variable love story

By Chris Piech

# A Variable origin ~~love~~ story

Nick Troccoli

By ~~Chris Piech~~

# Chapter 1: Birth

Once upon a time...

# ...a variable x was born!

```
int x;
```

# ...a variable x was born!

```
int x;
```



# x was a primitive variable...

```
int x;
```

Aww...!

It's so  
cuuuute!



# ...and its parents loved it very much.

```
int x;
```

We should  
give it....  
value 27!



# ...and its parents loved it very much.

$$x = 27;$$

We should  
give it....  
value 27!



A few years later, the parents decided to have another variable.

**...and a variable rect was born!**

**GRect rect;**



# rect was an object variable...

```
GRect rect;
```

Who's a  
cute  
GRect???

It's so  
square!



**...and its parents loved it very much.**

**GRect rect;**

We should  
make it.... a big,  
strong GRect!



# ...and its parents loved it very much.

```
GRect rect = new Grect(0, 0, 50, 50);
```

We should  
make it.... a big,  
strong GRect!



# ...but rect's box was not big enough for an object!

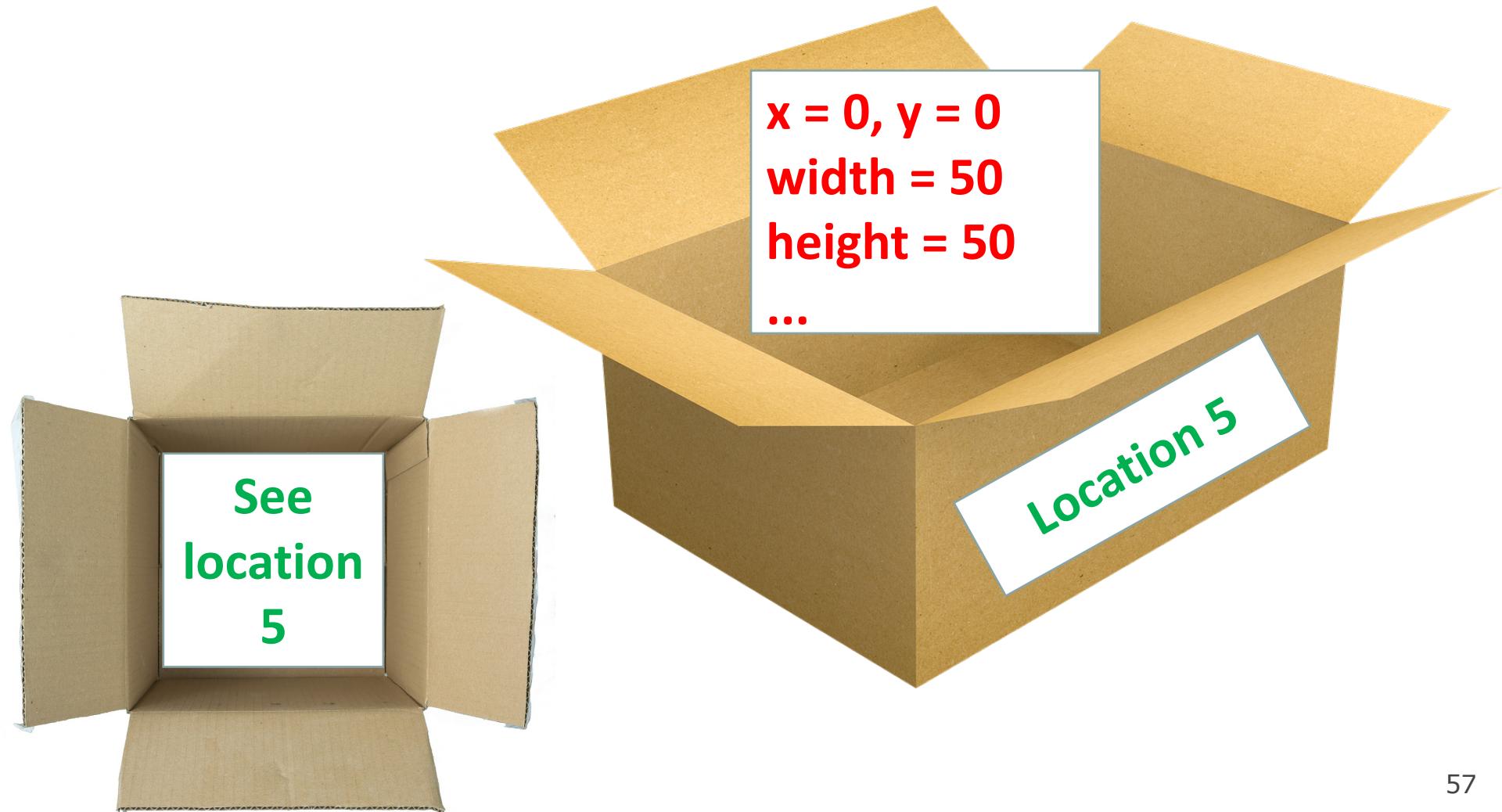
```
GRect rect = new Grect(0, 0, 50, 50);
```

That box isn't  
big enough to  
store  
everything  
about a GRect!



# ...so they stored the information in a bigger box somewhere else.

```
GRect rect = new Grect(0, 0, 50, 50);
```



# Chapter 2: Friends

# ...x makes a new friend!

```
int y = 27;
```



# ...x makes a new friend!

```
int y = 27;
```

Hi! I'm y.



# ...x makes a new friend!

```
int y = 27;
```



# ...x makes a new friend!

```
int y = 27;
```

\*blush\*

X

27

y

27

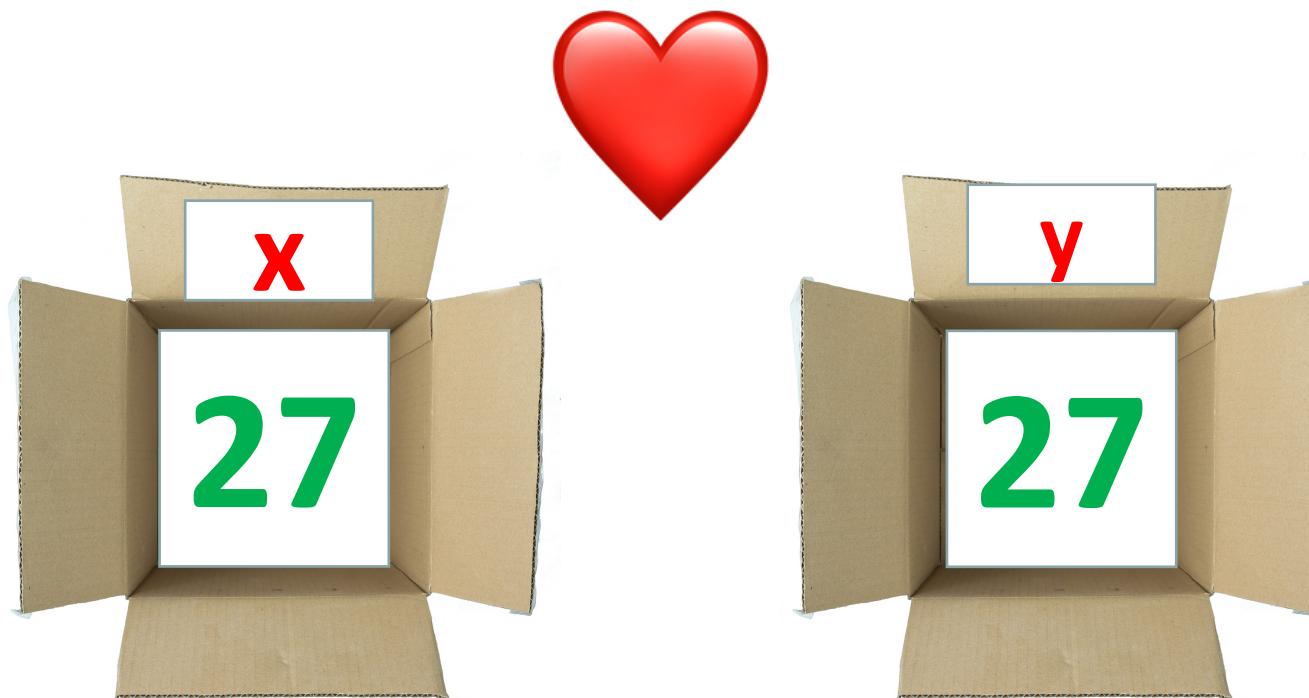
# They can use == to compare values.

```
if (x == y) { // true!  
...  
}
```



# They can use == to compare values.

```
if (x == y) { // true!  
...  
}
```



See “A Variable Love Story” for  
more...

# rect also makes a new friend!

```
GRect r = new Grect(0, 0, 50, 50);
```



rect

See  
location 5



r

See  
location 24

# Rect also makes a new friend!

```
GRect r = new Grect(0, 0, 50, 50);
```



# Rect also makes a new friend!

```
GRect r = new Grect(0, 0, 50, 50);
```



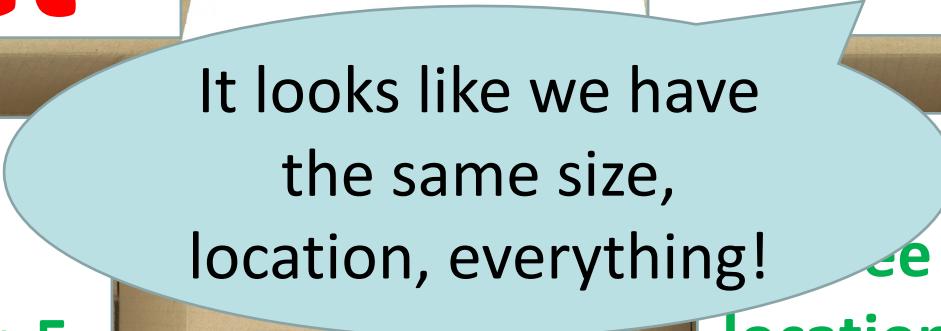
rect

See  
location 5



r

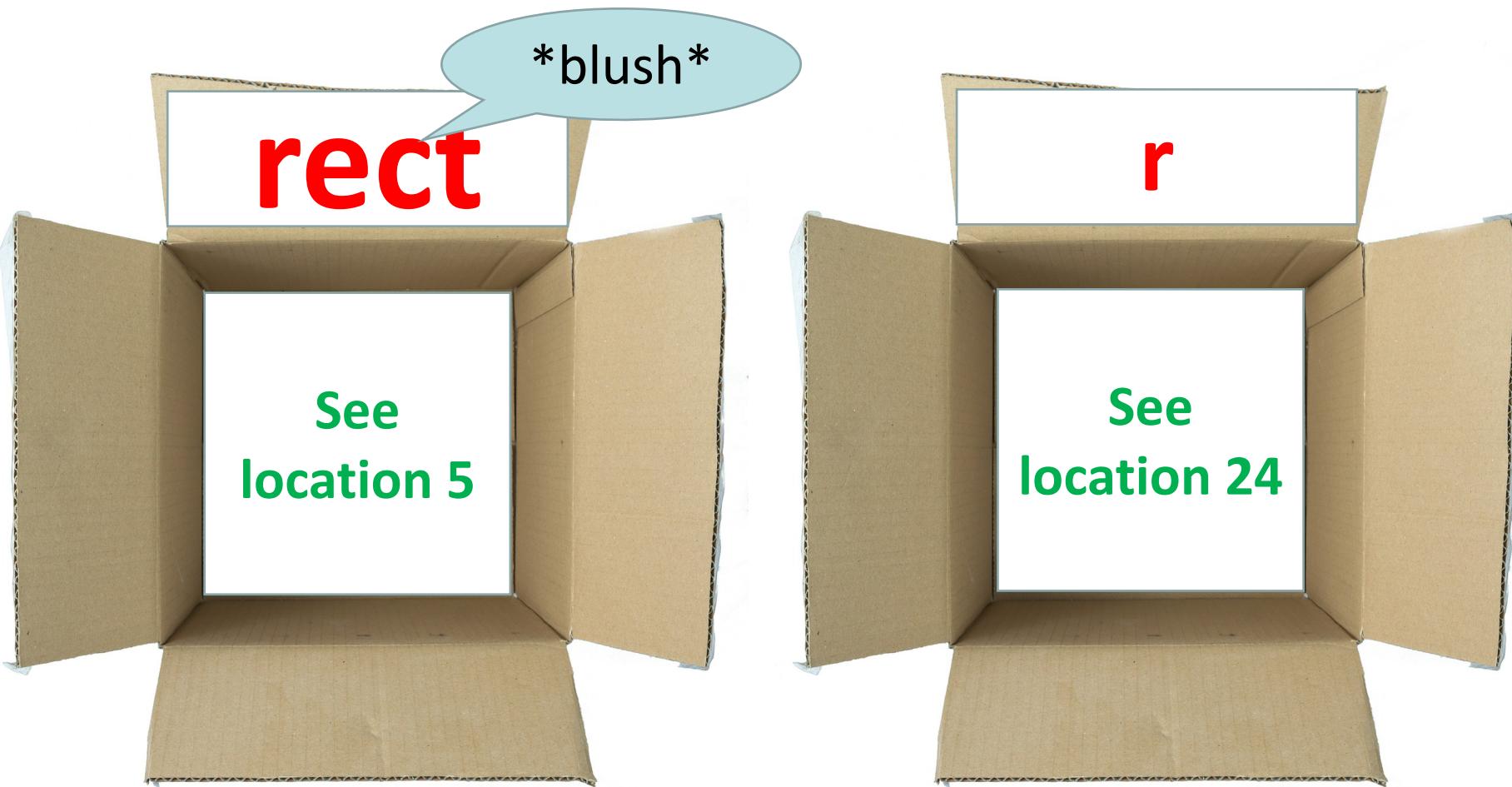
See  
location 24



It looks like we have  
the same size,  
location, everything!

# Rect also makes a new friend!

```
GRect r = new Grect(0, 0, 50, 50);
```



But when they use == to compare  
values...

# ...something goes wrong.

```
if (rect == r) { // false!  
    ...  
}
```



# ...something goes wrong.

```
if (rect == r) { // false!
```

...

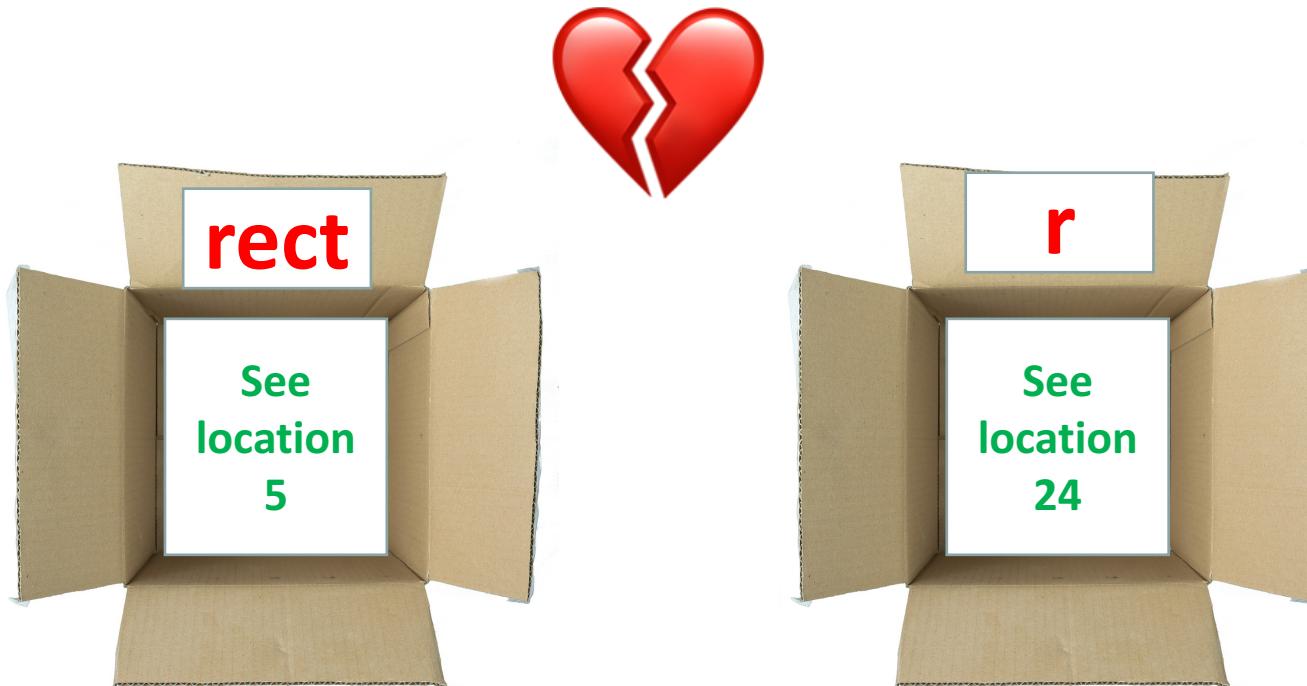
}

...but...but I thought  
we had so much in  
common!



# ...something goes wrong.

```
if (rect == r) { // false!  
...  
}
```



# ...something goes wrong.

```
if (rect == r) { // false!  
    ...  
}
```

You see, `==` compares  
what is in *each variable's  
box*.



# ...something goes wrong.

```
if (rect == r) { // false!  
    ...  
}
```

Primitives store their  
*actual value* in their box.



# ...something goes wrong.

```
if (rect == r) { // false!  
    ...  
}
```

But objects store the  
*location* where all their  
information lives.



# ...something goes wrong.

```
if (rect == r) { // false!
```

```
• • •  
}
```

This means == on objects compares their *locations*, which is not what we want here!



# Chapter 3: Twins

# One day, they wanted a twin.

Wow, what an awesome int! Let's make another one.



# One day, they wanted a twin.

Wow, what an awesome int! Let's make another one.

```
int x2 = x;
```



# ...so x2 was born!

Wow, what an awesome int! Let's make another one.

```
int x2 = x;
```



# ...so x2 was born!

But let's increment  
this one by 2.

```
int x2 = x;  
x2 += 2;
```



# ...so x2 was born!

```
int x2 = x;  
x2 += 2;
```



# ...so x2 was born!

```
int x2 = x;  
x2 += 2;
```



# Then, they wanted a twin for rect, too.

Wow, what an  
awesome GRect!  
Let's make another  
one.



# ...so rect2 was born!

Wow, what an awesome GRect!  
Let's make another one.

```
GRect rect2 = rect;
```



# ...so rect2 was born!

Wow, what an awesome GRect!  
Let's make another one.

```
GRect rect2 = rect;
```



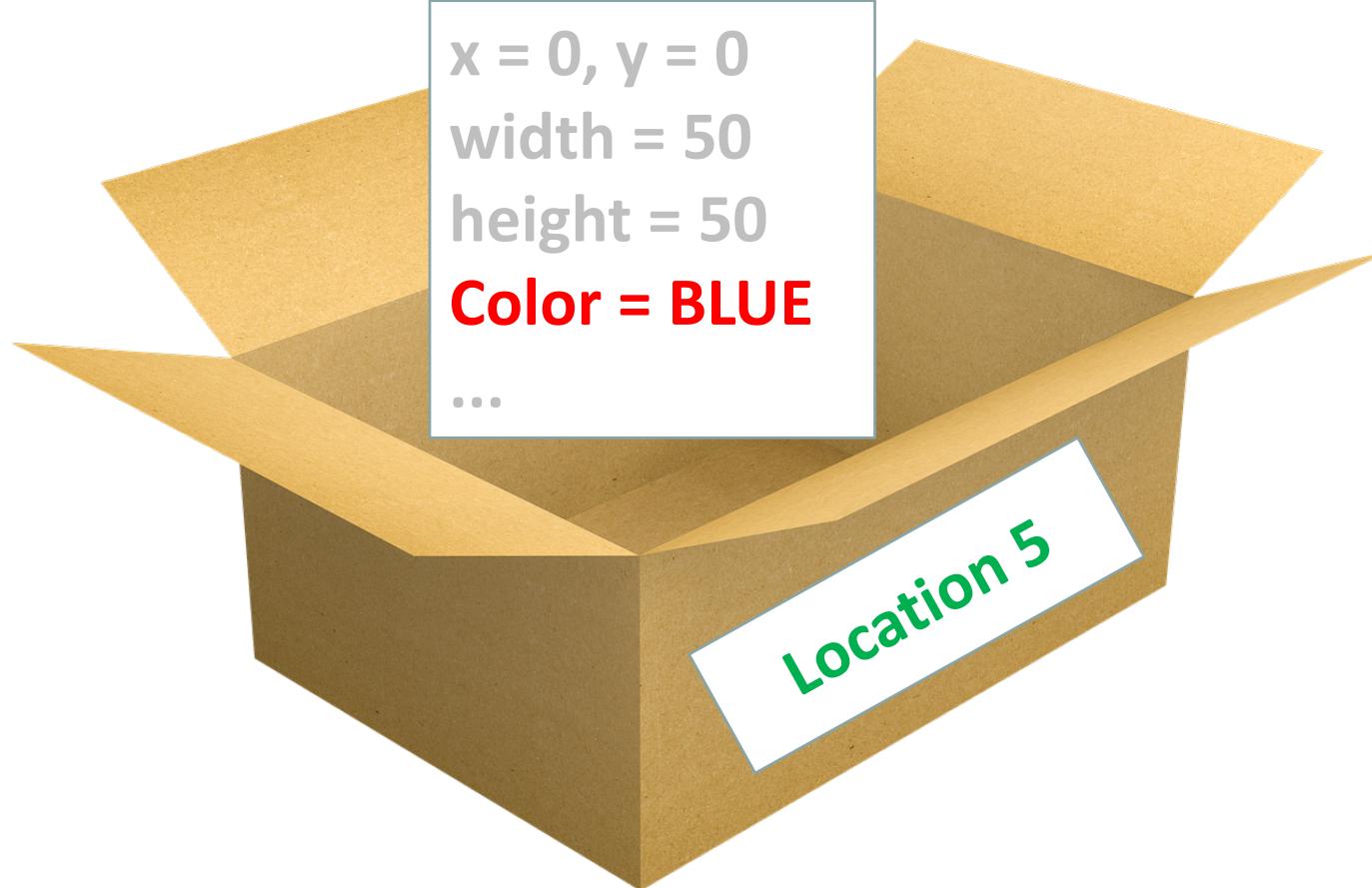
# ...so rect2 was born!

But let's make this  
one BLUE.

```
GRect rect2 = rect;  
rect2.setColor(Color.BLUE);
```



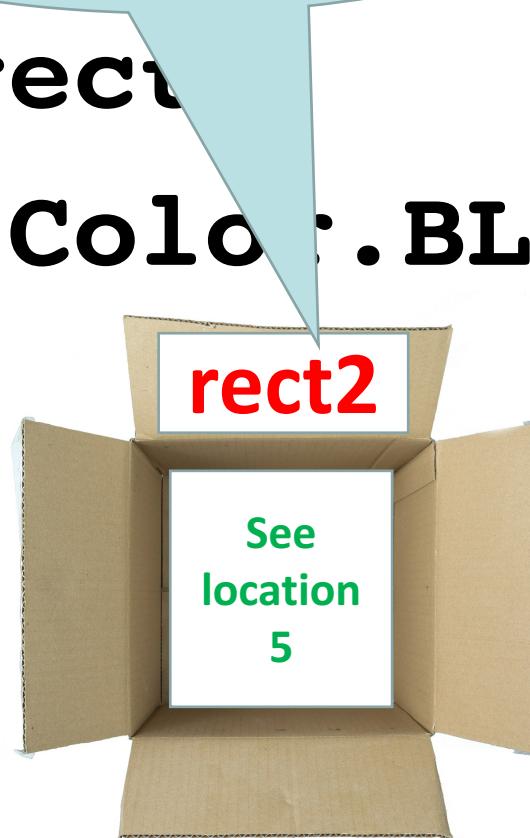
**...so they went to location 5 and changed the color to blue.**



# But something went wrong...

Cool, I'm blue!

```
GRect rect2 = rect  
rect2.setColor(Color.BLUE);
```



# But something went wrong...

```
GRect rect = new GRect(100, 100, 200, 200);  
rect.setFill(Color.BLUE);  
rect.setOutline(Color.BLACK);  
rect.setLineWidth(2);  
  
GRect rect2 = new GRect(100, 100, 200, 200);  
rect2.setFill(Color.BLUE);  
rect2.setOutline(Color.BLACK);  
rect2.setLineWidth(2);
```



Hey, why am  
I blue TOO?

# But something went wrong...

You see, when you set one variable equal to another, the value *in its box* is copied over.

```
GRect rect2 = rect;  
rect2.setColor(Color.BLUE);
```



# But something went wrong...

For primitive variables, this just means we copy their value.

```
GRect rect2 = rect;  
rect2.setColor(Color.BLUE);
```



# But something went wrong...

But for objects, we copy its  
*location* instead.

```
GRect rect2 = rect;  
rect2.setColor(Color.BLUE);
```



# But something went wrong...

Therefore, both rect and rect2 think their information is at location 5.

```
GRect rect2 = rect;  
rect2.setColor(Color.BLUE);
```



# But something went wrong...

When you change information for an object,  
you go to its location first, and then update  
the information.

```
GRect rect2 = rect;  
rect2.setColor(Color.BLUE);
```



# But something went wrong...

Since rect and rect2 reference the same location, when you change information for one of them, it changes for both!

```
GRect rect2 = rect;  
rect2.setColor(Color.BLUE);
```



# Chapter 4: Leaving the Nest

# **X grew up, and went to college.**

**college(x);**

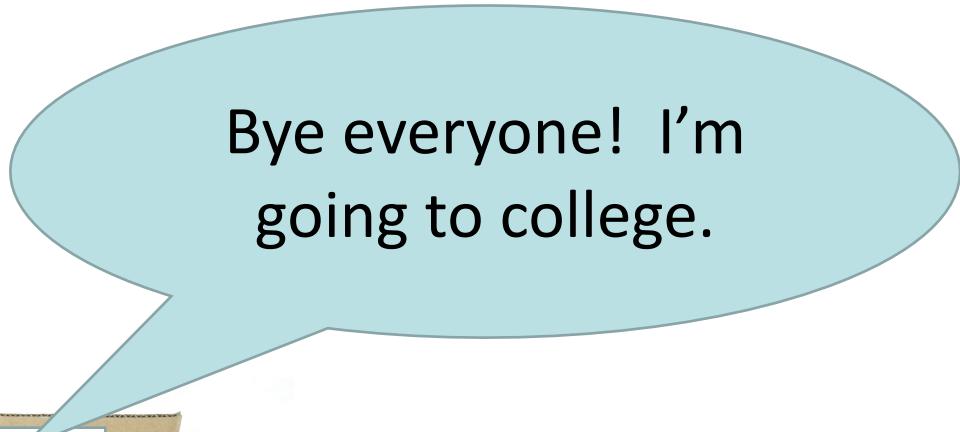


# **X grew up, and went to college.**

**college(x);**



We'll miss you,  
honey! Don't forget  
to call! Don't stay up  
too late! Watch out  
for rogue bicyclists!



Bye everyone! I'm  
going to college.

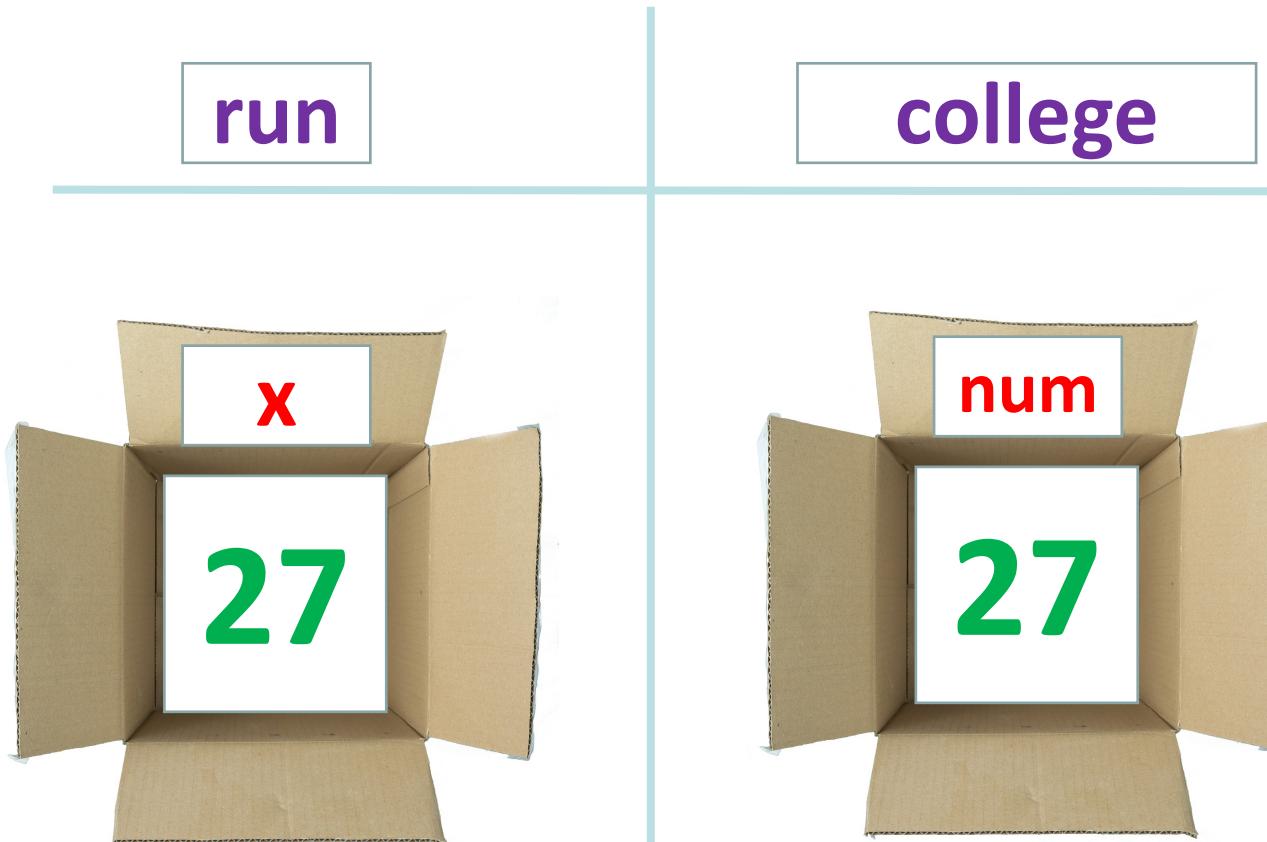


# X grew up, and went to college.

```
private void college(int num) {
```

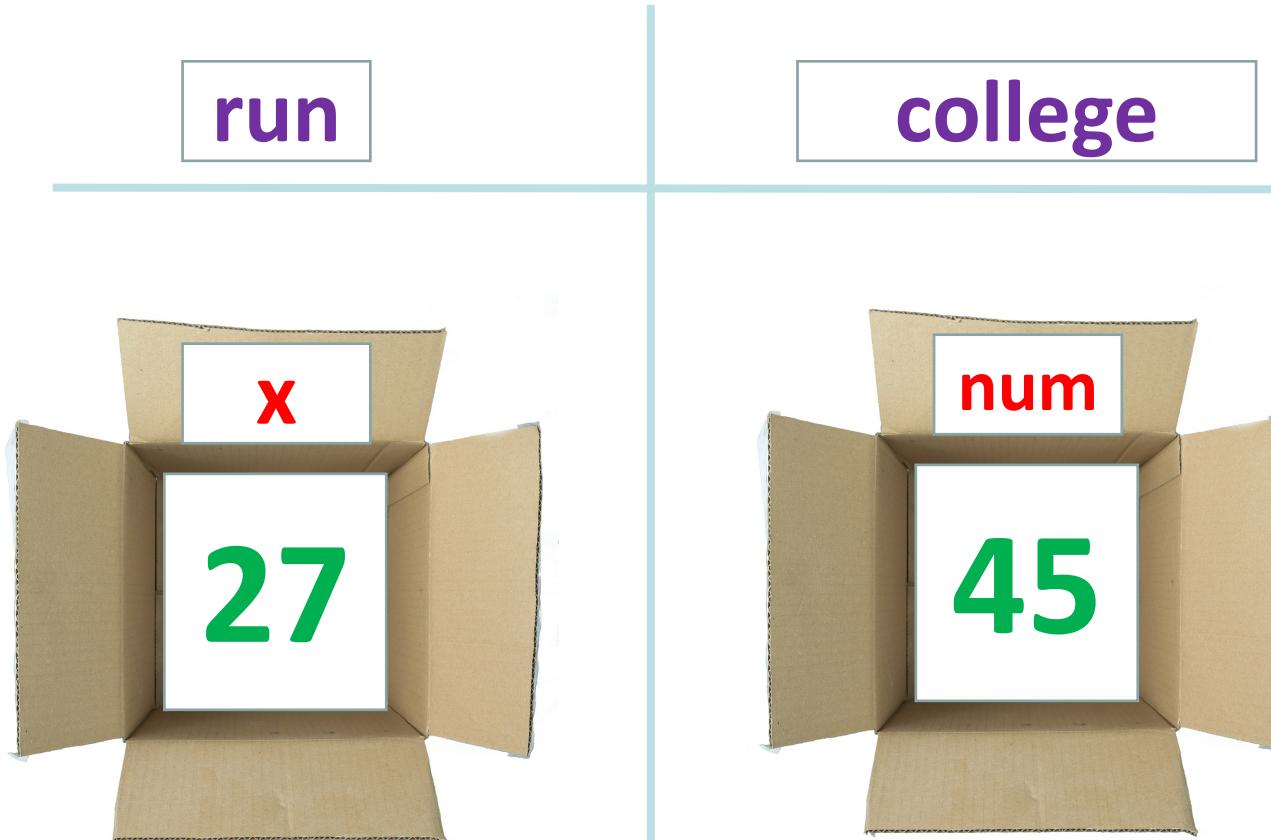
...

}



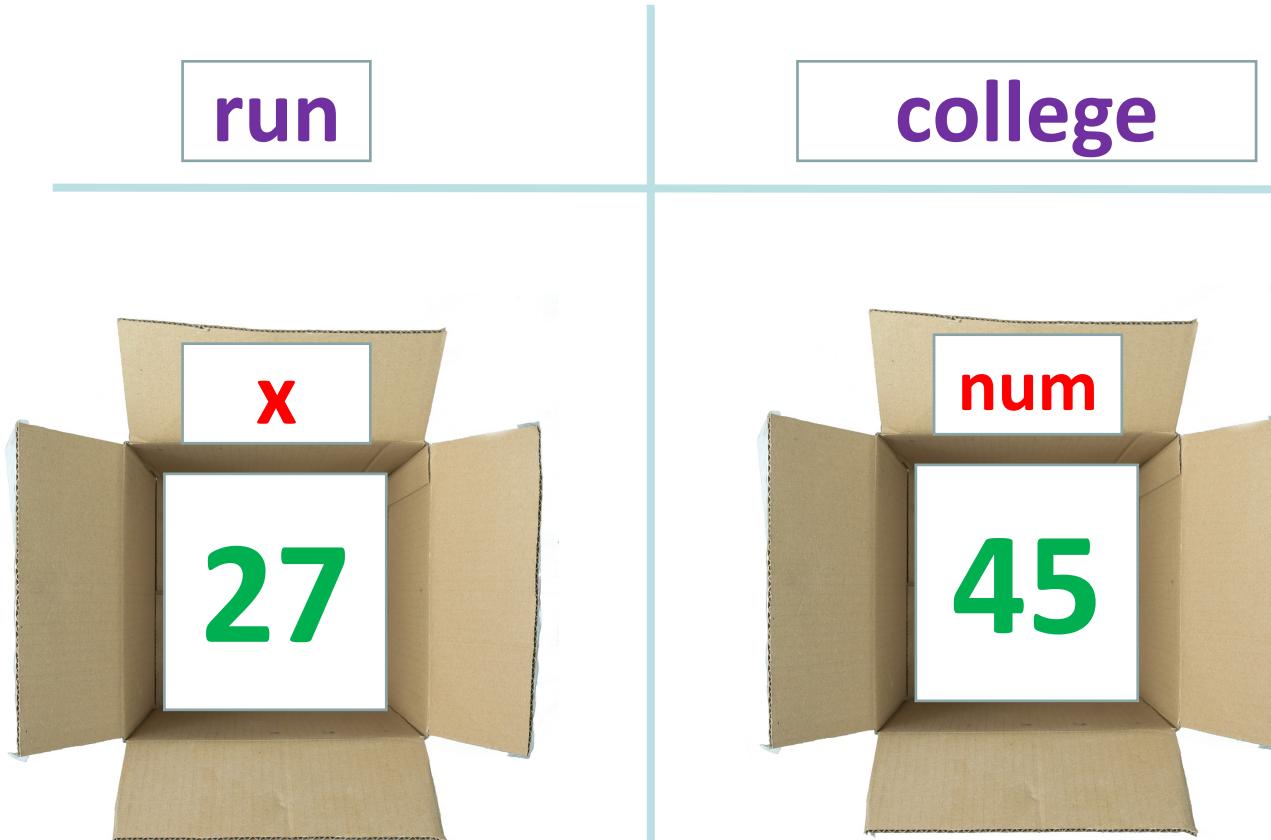
# x had an amazing college experience...

```
private void college(int num) {  
    num = 45;  
}
```



# ...but ultimately returned home the same.

```
private void college(int num) {  
    num = 45;  
}
```



**...but ultimately returned home the same.**

**college(x);**

**println(x);**



# **...but ultimately returned home the same.**

```
college(x);
```

```
println(x);
```

Ohhhh honey! You'll  
always be our little  
27.

I'm baaaaack! And  
I'm still 27.



**rect grew up too, and also went to college.**

**college2(rect);**

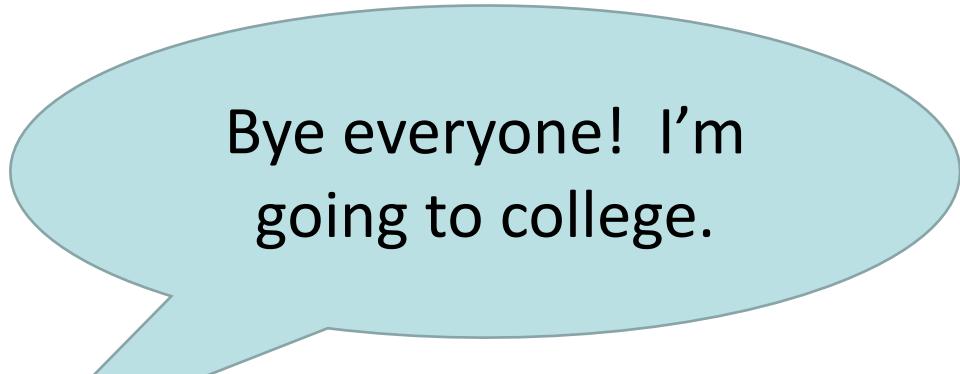


**rect grew up too, and also went to college.**

**college2( rect );**



Not you too, rect! We are empty nesters now...



Bye everyone! I'm going to college.

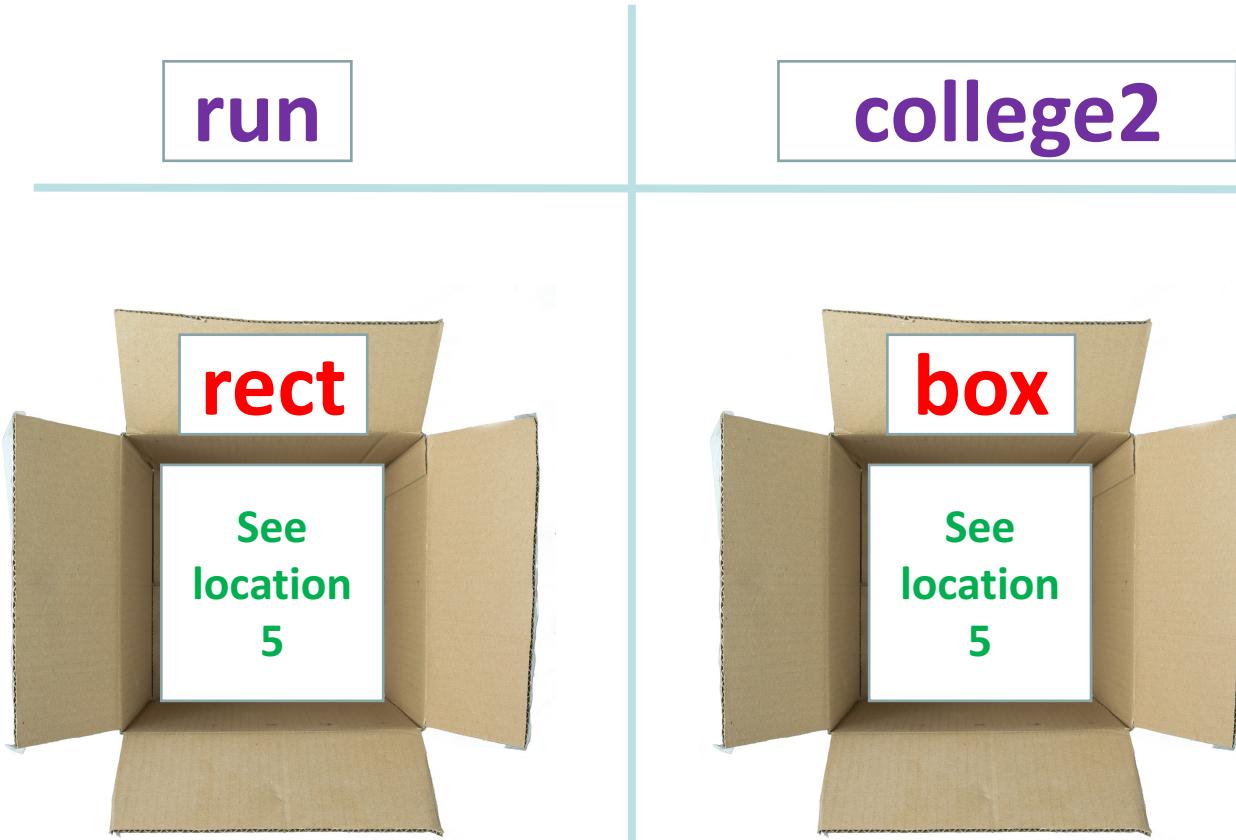


# rect grew up too, and also went to college.

```
private void college2(GRect box) {
```

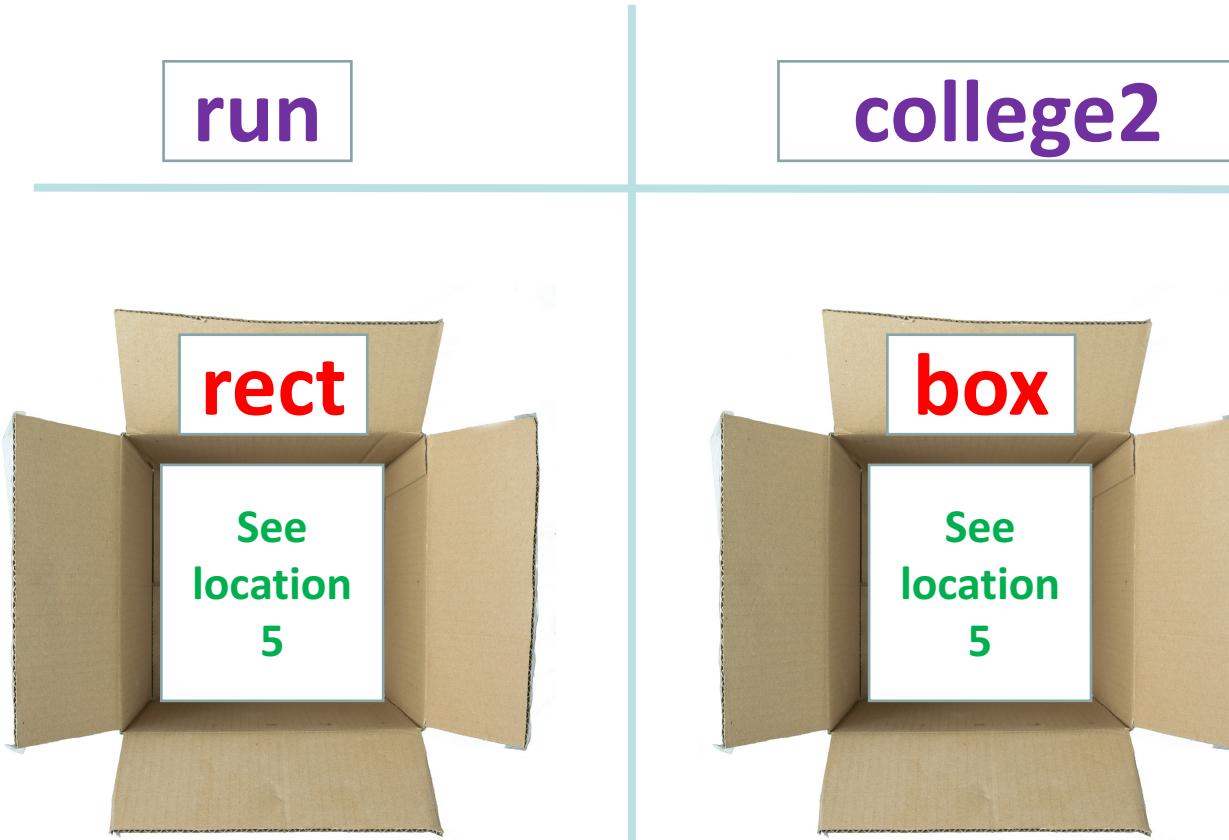
...

}

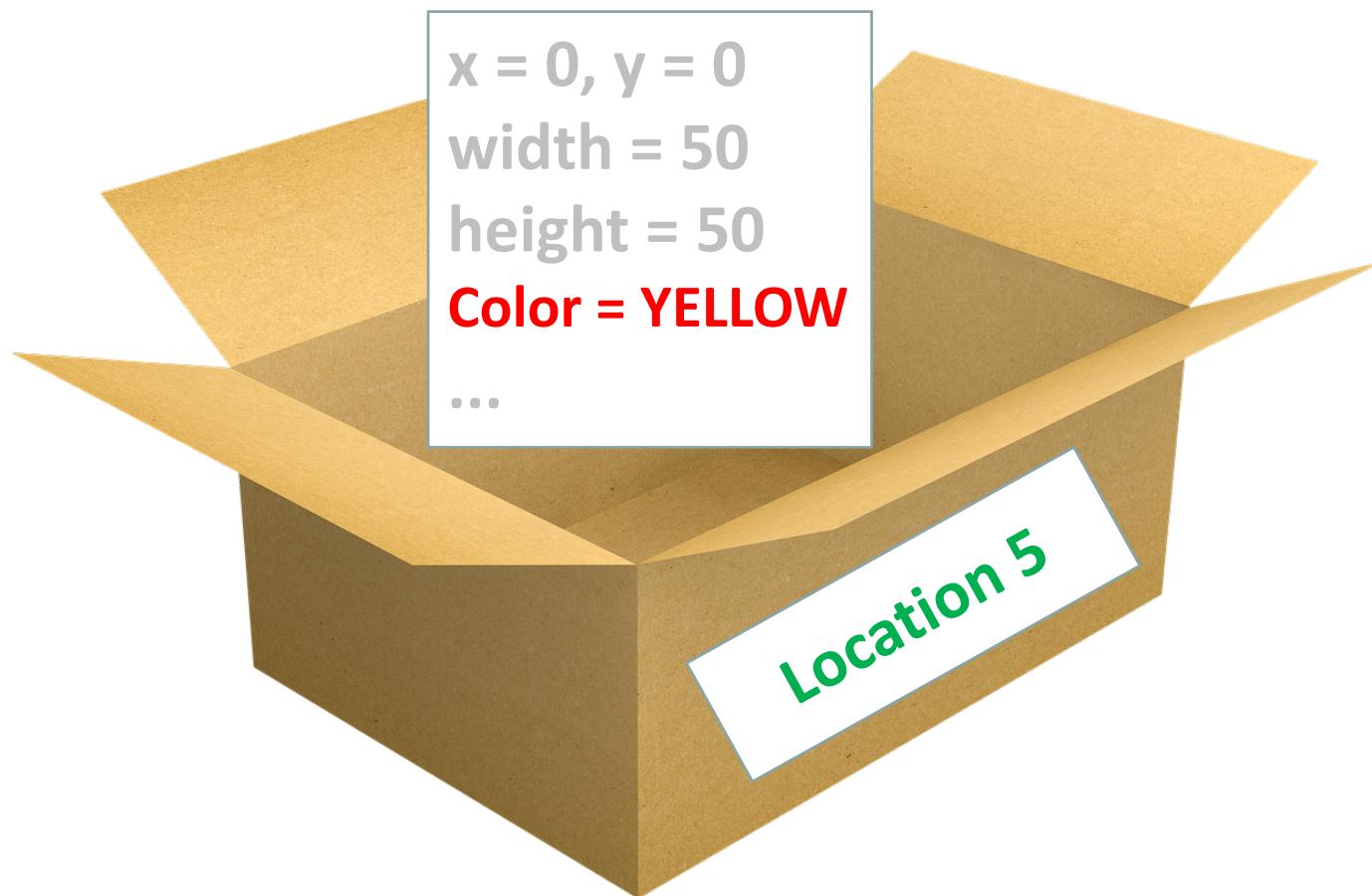


# rect also had an amazing college experience...

```
private void college2(GRect box) {  
    box.setColor(Color.YELLOW);  
}  
}
```

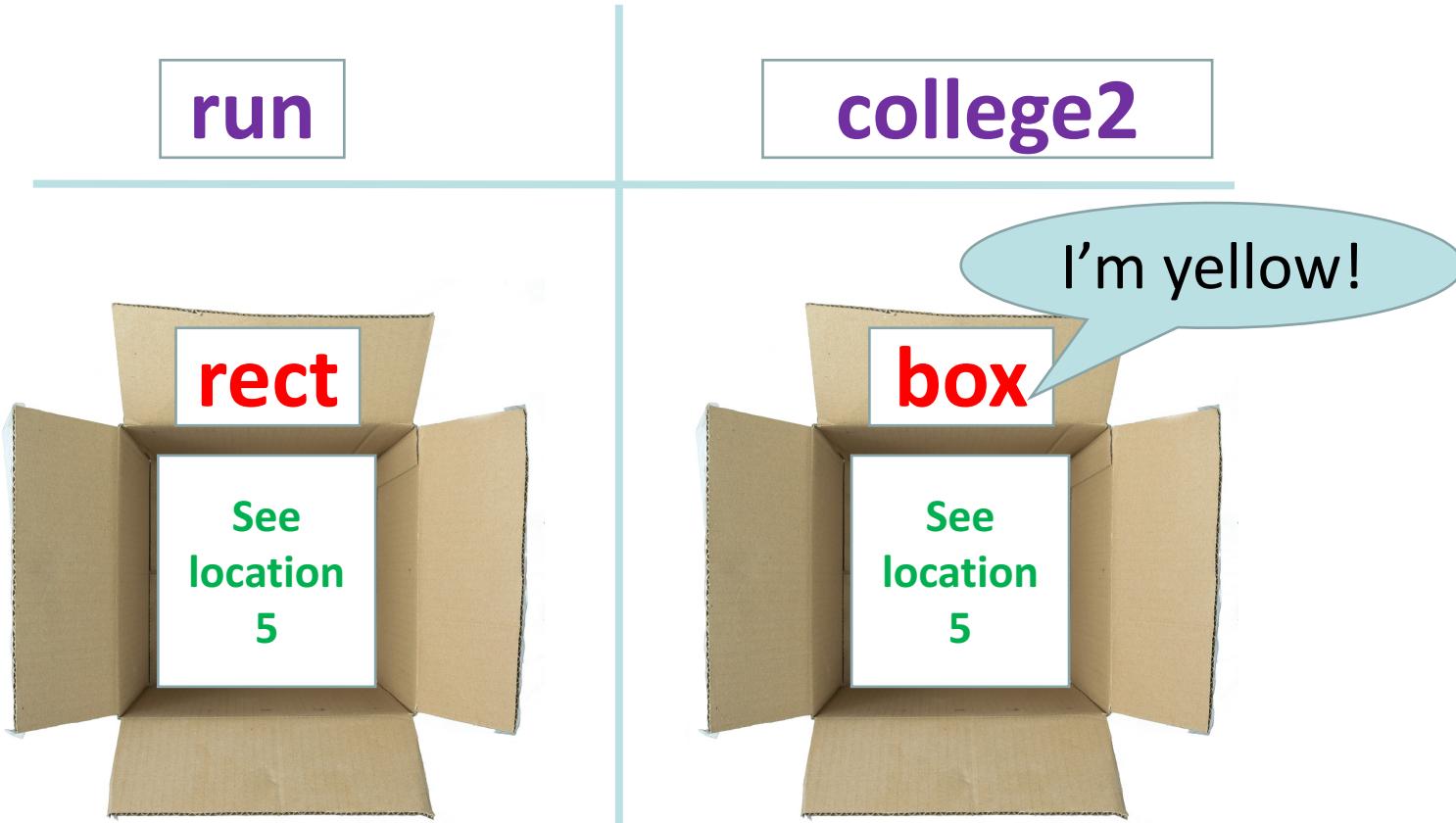


# rect also had an amazing college experience...



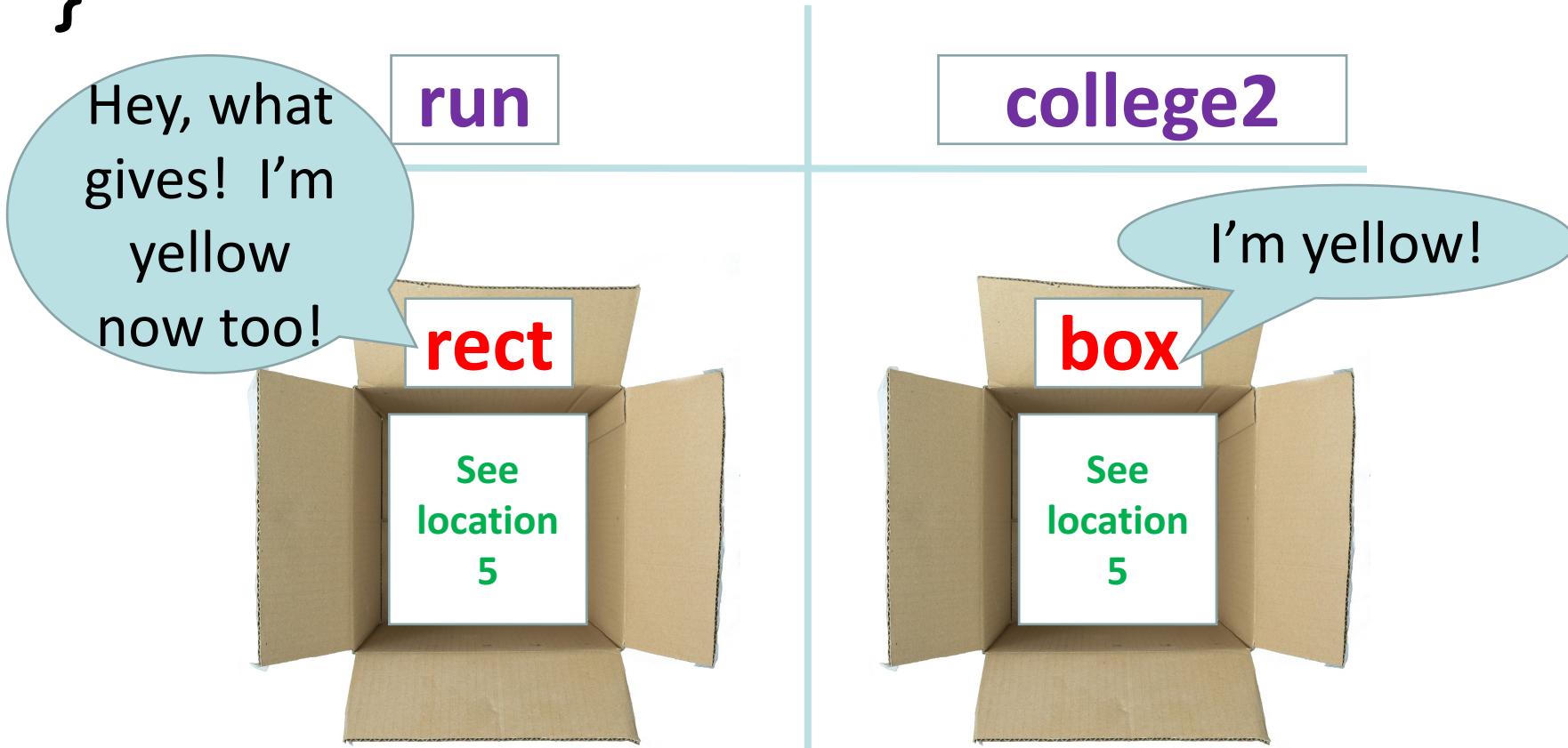
# ...but it returned home different.

```
private void college2(GRect box) {  
    box.setColor(Color.YELLOW);  
}
```



# ...but it returned home different.

```
private void college2(GRect box) {  
    box.setColor(Color.YELLOW);  
}
```



# ...but it returned home different.

```
college(rect);
```

```
add(rect);
```



# ...but it returned home different.

**college(rect);**

**add(rect);**

Oh my gosh....what  
*happened* to you??!

I'm baaaaack! But  
now I'm yellow...



# ...but it returned home different.

```
college(rect);
```

```
add(rect);
```

You see, when a variable is passed as a parameter, the value *inside its box* is copied and given to the method.



# ...but it returned home different.

```
college(rect);
```

```
add(rect);
```

For primitives, we make a copy of their *actual value*. Therefore, changes to a parameter do not affect the original.



# ...but it returned home different.

```
college(rect);
```

```
add(rect);
```

However, for objects we make a copy of their *location*. So changes to the parameter *do affect the original!*



The End

# Primitives vs. Objects

- **Primitives** store their *actual value* in their variable box. You can compare values with == and !=, and the original does not change when passed as a parameter and changed.
- **Objects** store their *location* in their variable box. You can't compare properties of an object via ==, and the original *does* change when passed as a parameter and changed.
- **Primitives** are *passed by value*, **Objects** are *passed by reference*

# Practice – Chapter 2: Friends

# Practice

```
public void run() {  
    int x = 5;  
    int y = 5;  
    println(x == y);  
}
```

Does this print out **true** or  
**false**?

# Practice

```
public void run() {  
    GRect first = new GRect(20, 30);  
    GRect second = new GRect(20, 30);  
    println(first == second);  
}
```

Does this print out **true** or  
**false**?

# Practice – Chapter 3: Twins

# Practice

```
public void run() {  
    int x = 5;  
    int y = x;  
    y += 2;  
    println(y);  
    println(x);  
}
```

What is printed out?

# Practice

```
public void run() {  
    GOval oval = new GOval(0, 0, 50, 50);  
    GOval oval2 = oval;  
    oval2.setColor(Color.BLUE);  
    add(oval2);  
    add(oval);  
}
```

What color are oval and  
oval2?

Practice – Chapter 4: Leaving the Nest

# Practice

```
public void run() {  
    int x = 2;  
    addFive(x);  
    println(x);  
}  
  
private void addFive(int num) {  
    num += 5;  
}
```

What is printed out?

# Practice

```
public void run() {  
    GRect rect = new GRect(0, 0, 50, 50);  
    makeBlue(rect);  
    add(rect);  
}  
  
private void makeBlue(GRect box) {  
    box.setColor(Color.BLUE);  
}
```

What color is rect?

# Primitives vs. Objects

**Primitives** store their actual values. **Objects** store their *locations*.

This affects how your code behaves!

# getElementAt

**getElementAt** returns a *reference* to an object.

```
GRect rect = new GRect(0, 0, 50, 50);  
add(rect);
```

...

```
GObject obj = getElementAt(25, 25);  
println(obj == rect);
```

This prints **true!**

# Plan for Today

- Announcements
- Review: Null, Events and Instance Variables
- Extra Practice: Googley Eyes
- A Boolean Aside
- Memory
- Revisiting Whack-A-Mole

# Revisiting Whack-A-Mole

Let's revisit our Whack-A-Mole program to add some new functionality using what we just learned.



# Recap

- Announcements
- Review: Null, Events and Instance Variables
- Extra Practice: Googley Eyes
- A Boolean Aside
- Memory
- Revisiting Whack-A-Mole

**Good luck on the midterm!**