Announcements

- SCPD OH only for SCPD students
- Midterm in 1 week...
  - In-Person Review Session on Friday, at 10:30am in Gates B01
  - Midterm will be 7pm - 9pm in Bishop
Plan for Today

● Review: Pass by Reference vs. Value, Eclipse Debugger
● Equality: Primitives and Objects
● Primitives on the Stack
● Objects on the Heap
● Why This Matters!
Pass by Reference

**Objects** are passed by reference.

A few examples:

- `GRect`
- `GImage`
- `Goval`

---

Pass by Value

**Primitives** are passed by value.

A few examples:

- `int`
- `char`
- `double`
- `boolean`
What does this mean?

If something is passed by *reference*, it *can* be altered simply by passing it into a method.

If something is passed by *value*, it *cannot* be altered simply by passing it into a method.
Types of Errors

Syntax Errors:
A programming “typo”. Usually causes a red squiggly line in code.

Execution Errors:
Something that crashes the program after you run it.

Logic Errors:
All of your code runs, but it produces unexpected results.
How Can We Debug Errors?

Using the Eclipse Debugger!
Debugger Commands

- **Suspend.** Stops the program immediately as if it had hit a breakpoint.
- **Terminate.** Exits from the program entirely.
- **Step Into.** Executes one statement of the program and then stops again. If that statement includes a method call, the program will stop at the first line of that method. As noted below, this option is not as useful as **Step Over.**
- **Step Over.** Executes one statement of the program at this level and then stops again. Any method calls in the statement are executed through to completion unless they contain explicit breakpoints.
- **Step Return.** Continues to execute this method until it returns, after which the debugger stops in the caller (the method that called the current method).

*The debugger allows you to step through the execution of a program, one line at a time.*
I can view what my variables are currently equal to by opening the “Variables” View.

How?
- Click Window
- Click Show View
- Click Other...
- Search for Variables
- Click Variables
Setting a Breakpoint

To start the debugger, it helps to set a breakpoint. Double click next to the line number where you want to set a breakpoint. This will create a little blue dot, or `breakpoint`.

Your code will stop and start the debugger when it sees the breakpoint.
Are They Equal?

```java
public void run(){
    int num1 = 12;
    int num2 = 12;

    if(num1 == num2){
        println("These integers are equal!");
    } else {
        println("Actually, these integers are not equal.");
    }
}
```
public void run(){

    GRect rect1 = new GRect(100, 100);
    GRect rect2 = new GRect(100, 100);

    if(rect1 == rect2){
        println("These rectangles are equal!" );
    } else {
        println("Actually, these rectangles are not equal.");
    }
}
public void run(){

    GRect rect3 = new GRect(100, 100);

    if(rect3 == rect3){
        println("This rectangle is equal to itself!");
    } else {
        println("Actually, this rectangle is not equal to itself.");
    }
}
What’s Going On Here?

Can you have two different versions of the same number? Can one number 12 be different from another number 12?
What’s Going On Here?

Can you have **two different versions** of the same number? Can one number 12 be different from another number 12?

Can you have **two different versions** of the same rectangle? Can one rectangle with a width and height of 100 be different from another rectangle with a width and height of 100?
What’s Going On Here?

You can have two different rectangles with the same properties, but you can’t have two different number 12s.

Being the same thing is different from having the same properties.
public void run(){

    GRect rect3 = new GRect(100, 100);

    if(rect3 == rect3){
        println("This rectangle is equal to itself!");
    } else {
        println("Actually, this rectangle is not equal to itself.");
    }
}

This will only evaluate to true if it’s the exact same rectangle! Let’s look into why.
Hey Duke!

Well, you’ll always be in my memory

Heap ofc

Karel! So good to see you. um, will Sarai delete us?

Heap or stack?

Now that’s true friendship <3
public void run(){
    int primitiveInt = 0;
    changeInt(primitiveInt);
}

private void changeInt(int primitiveInt){
    primitiveInt += 10;
}

Remember, in Thursday’s example, when we called a new method, we put a box representing our method on top of the box representing run().

We created a stack.
What Happened: Primitive

```java
public void run(){
    int primitiveInt = 0;
    changeInt(primitiveInt);
}

private void changeInt(int primitiveInt){
    primitiveInt += 10;
}
```

We created a stack.

The computer does the same thing!

It creates a stack of things to keep track of in its temporary memory.
What Happened: Primitive

The computer creates a stack of things to keep track of in its temporary memory.

And when the computer is done with something on the stack?
What Happened: Primitive

public void run(){
    int primitiveInt = 0;
    changeInt(primitiveInt);
    println("primitiveInt: " + primitiveInt, 0, 50);
    ...
}

The old item in memory is removed from the stack!
An Example: Stack

```java
public void run(){
    firstMethodCall()
    ...
}
```

The **stack** stores method calls and local variables in our program.

Right now `run()` is on the top of the stack.
An Example: Stack

The **stack** stores method calls and local variables in our program.

**Right now**

*firstMethodCall()* is on the top of the stack.
An Example: Stack

```java
public void run(){
    firstMethodCall()
}
```

```java
public void firstMethodCall(){
    secondMethodCall()
}
```

```java
public void secondMethodCall(){
    thirdMethodMethodCall()
    ...
}
```

Right now
secondMethodCall() is on the top of the stack.

And so it continues.
An Example: Stack

```java
public void run(){
    int num1 = 18;
    int num2 = 13;
    double answer = average(num1, num2);
}
```

Right now `run()` is on the top of the stack.

We will create a new stack frame for it.
As we create local variables, they are added to our stack as well.

Local variables are variables created within our current scope.
public void run(){
    int num1 = 18;
    int num2 = 13;
    double answer = average(num1, num2);
}

As we create **local variables**, they are added to our stack as well.

**Local variables** are variables created within our current scope.
An Example: Stack

```java
public void run(){
    int num1 = 18;
    int num2 = 13;
    double answer = average(num1, num2);
}
```

Look, we see a new method!

Remember, we always evaluate the right side first!
public void run(){
    int num1 = 18;
    int num2 = 13;
    double answer = average(num1, num2);
}

Look, we see a new method!

(Remember, we always evaluate the right side first!)

What do we do now?
An Example: Stack

```java
public void run(){
    int num1 = 18;
    int num2 = 13;
    double average = calculateAverage(5, 10);
}
```

Create a new stack frame!

Every time a new method is called we create a new stack frame and copy the parameter values that were passed in!

```java
private double average(double a, double b){
    double sum = a + b;
    return sum / 2;
}
```

```
run
average
```

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>18.0</td>
</tr>
<tr>
<td>b</td>
<td>13.0</td>
</tr>
</tbody>
</table>
```
public void run(){
    int num1 = 18;
    int num2 = 13;
    double average = calculateAverage(5, 10);
}

double calculateAverage(double a, double b){
    double sum = a + b;
    return sum / 2;
}

As new local variables are created, we created new boxes for them and add them to the stack as well!
An Example: Stack

```java
public void run(){
    int num1 = 18;
    int num2 = 13;
    double average = calculateAverage(5, 10);
}
```

When we `return` this allows a method to pass information back to the **caller**.

The **caller** is the method that called our current method.

```java
private double average(double a, double b){
    double sum = a + b;
    return sum / 2;
}
```

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.0</td>
<td>13.0</td>
<td>31.0</td>
</tr>
</tbody>
</table>
An Example: Stack

public void run (){  
    int num1 = 18;  
    int num2 = 13;  
    double answer = average(num1, num2);  
}

When `average` returned, we removed it from the stack frame!

This removed it from memory as well as all of the local variables it created!
All of the primitive variables we create in memory have a fixed size.

All ints receive the same amount of space, all doubles receive the same amount of space, etc...
Primitives

- **ints** receive 4 bytes.
- **doubles** receive 8 bytes.
- **chars** receive 2 bytes.

Booleans are less precisely defined, but still a primitive!
Objects

What about objects?
Primitives receive a set amount of space on the stack when they are created! They store a small amount of data.
Primitives receive a set amount of space on the stack when they are created! They store a small amount of data.

Objects store LOTS of data!
Primitives receive a set amount of space on the stack when they are created! They store a small amount of data.

Primitives receive a set amount of space on the stack when they are created! They store a small amount of data.

Objects store LOTS of data! Java chooses to store objects on the heap.
The Heap is more permanent memory. Things on the heap don’t disappear as methods are called or returned.

Why do we care that objects are stored on the heap?
public void run(){

    GRect rect3 = new GRect(100, 100);

    if(rect3 == rect3){
        println("This rectangle is equal to itself!");
    } else {
        println("Actually, this rectangle is not equal to itself.");
    }
}

Remember: This will only evaluate to true if it's the exact same rectangle! This has something to do with how objects are stored in memory.
Like before, we'll put `run()` on our stack!

```java
public void run(){
    GRect rect = new GRect(100, 100);
    rect.setFilled(true);
    rect.setColor(Color.BLUE);
    add(rect, 100, 100);
}
```
Objects

public void run(){
    GRect rect = new GRect(100, 100);
    rect.setFilled(true);
    rect.setColor(Color.BLUE);
    add(rect, 100, 100);
}

We’ll create a GRect.
Remember: we evaluate the right-hand side first!
public void run(){
    GRect rect = new GRect(100, 100);
    rect.setFilled(true);
    rect.setColor(Color.BLUE);
    add(rect, 100, 100);
}

Wait a second! Why did it create a GRect outside of run()?
public void run(){
    GRect rect = new GRect(100, 100);
    rect.setFill(true);
    rect.setColor(Color.BLUE);
    add(rect, 100, 100);
}

Wait a second! Why did it create a GRect outside of run()?

It created our GRect on the heap!
public void run(){
    GRect rect = new GRect(100, 100);
    rect.setFilled(true);
    rect.setColor(Color.BLUE);
    add(rect, 100, 100);
}

rect becomes equal to the **location in memory** where our GRect is.
public void run(){

    GRect rect1 = new GRect(100, 100); // rect1 = location 4
    GRect rect2 = new GRect(100, 100); // rect2 = location 5

    // is the memory address of rect1 equal to the memory address of rect2?
    if(rect1 == rect2){
        println("These rectangles are equal!");
    } else {
        println("Actually, these rectangles are not equal.");
    }
}
public void run(){

    GRect rect1 = new GRect(100, 100); // rect1 = location 4
    GRect rect2 = new GRect(100, 100); // rect2 = location 5

    // is the memory address of rect1 equal to the memory address of rect2?
    if(rect1 == rect2){
        println("These rectangles are equal!");
    } else {
        // no! They’re at different locations in memory!
        println("Actually, these rectangles are not equal.");
    }
}

We cannot create two different GRects in the same location in memory.
public void run(){

    GRect rect4 = new GRect(100, 100); // rect4 = location 10
    GRect rect5 = rect4;

    // is the memory address of rect4 equal to the memory address of rect5?
    if(rect4 == rect5){
        println("These rectangles are equal!");
    } else {
        println("Actually, these rectangles are not equal.");
    }
}

public void run(){
    GRect rect4 = new GRect(100, 100); // rect4 = location 10
    GRect rect5 = rect4;

    // is the memory address of rect4 equal to the memory address of rect4?
    if(rect4 == rect5){
        // They are equal! They both point to the same place in memory!
        println("These rectangles are equal!");
    } else {
        println("Actually, these rectangles are not equal.");
    }
}
Can you have two different versions of the same number? Can one number 12 be different from another number 12?

No. Two primitive values will always be equal if they’re the same. Their values are stored on the stack.
Can you have **two different versions** of the same rectangle? Can one rectangle with a width and height of 100 be different from another rectangle with a width and height of 100?

**Yes.** Two rectangles are only the same if the **value of their memory addresses** are the same. Each rectangle is stored on the heap while the value of their memory address can be stored on the stack.
If something is passed by *reference*, it *can* be altered simply by passing it into a method. This is because we are passing in a *reference to its location in memory*, not a copy of the object.

If something is passed by *value*, it *cannot* be altered simply by passing it into a method. This is because we are a passing in a *copy of its value*. 
public void run(){
    int size = 250;
    GRect rect1 = new GRect(size, size);
    GRect rect2 = rect1;
    changeSize(rect2, size);
    add(rect1, 100, 100);
}

private void changeSize(GRect rect, double size){
    size += 250;
    rect.setSize(size, size);
}
public void run(){
    int size = 250;
    GRect rect1 = new GRect(size, size);
    GRect rect2 = rect1;
    changeSize(rect2, size);
    add(rect1, 100, 100);
}

private void changeSize(GRect rect, double size){
    size += 250;
    rect.setSize(size, size);
}
public void run(){
    int size = 250;
    GRect rect1 = new GRect(size, size);
    GRect rect2 = rect1;
    changeSize(rect2, size);
    add(rect1, 100, 100);
}

private void changeSize(GRect rect, double size){
    size += 250;
    rect.setSize(size, size);
}
public void run(){
    private void changeSize(GRect rect, double size){
        size += 250;
        rect.setSize(size, size);
    }
}
public void run(){

private void changeSize(GRect rect, double size){
    size += 250;
    rect.setSize(size, size);
}
}
What Happens Here?

```java
public void run() {

    private void changeSize(GRect rect, double size) {
        size += 250;
        rect.setSize(size, size);
    }

    rect.setSize(size, size);

    size += 250;
    rect.setSize(size, size);
}
```
public void run(){
    int size = 250;
    GRect rect1 = new GRect(size, size);
    GRect rect2 = rect1;
    changeSize(rect2, size);
    add(rect1, 100, 100);
}

private void changeSize(GRect rect, double size){
    size += 250;
    rect.setSize(size, size);
}
What Happens Here?

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>no method return</td>
<td>ChangingSize (id=29)</td>
</tr>
<tr>
<td>this</td>
<td>GRect (id=68)</td>
</tr>
<tr>
<td>size</td>
<td>250</td>
</tr>
<tr>
<td>rect1</td>
<td>GRect (id=68)</td>
</tr>
<tr>
<td>rect2</td>
<td>GRect (id=68)</td>
</tr>
</tbody>
</table>

GRect[bounds=(0.0, 0.0, 500.0, 500.0), filled=false, fillColor=BLACK]
What happens if there's too many things on the stack?
Stack Overflow!

Oh dear...
Plan for Today

- Review: Pass by Reference vs. Value, Eclipse Debugger
- Equality: Primitives and Objects
- Primitives on the Stack
- Objects on the Heap
- Why This Matters!