This lecture started with a continuation of last class and using overriding methods to solve a problem. We finished by discussing the method call stack.

Overriding Methods, cont.

Recall the GeometricFrame.java class we wrote. It had a feature to place the size on the title. However, it suffered from two problems:

1. We had to remove the size from the title before we changed the title or else it would not display the correct title.

2. We had to call displaySizeOnTitle everytime the window changed size or else it would not display the correct size.

We now know how to fix problem 1.

We need to override the setTitle method! Why? Because we want to change its behavior so that when the setTitle method is called, it detects whether the size is being displayed

and then keeps the display on if it is.

We know it is the setTitle method that needs to be overridden because it is when we call that method that the display does not work properly.

How to override a method? We write another one with the exact same return type, name and parameter signature:

public void setTitle(String title) {

Now, what should we do?

In lecture, there were a lot of suggestions, but a hint you should keep in mind is that we do not want to write code that directly puts the size on or off the title.

We already have a routine to do that, and it works just fine. If we were to duplicate those actions, we would risk breaking the routines we already wrote.

Instead, let's use the routines and recall how we have to currently handle the title.

If the size is displayed:

turn the display off

now change the title

turn the display back on

If the size is not displayed:

set the title as normal

Recall that originalTitle stored the original title if the display was on and it stored null if the display was off.

Also recall the use of

super

Super refers to the parent class. It is used with parentheses to call the parent class constructor. It is used with a dot to access a field or method of the parent class.

Now, we know what we need to do:

public void setTitle(String title) {

if (originalTitle != null) {

displaySizeOnTitle(false);

super.setTitle(title);

displaySizeOnTitle(true);

}

else

super.setTitle(title);

}

WARNING: We just changed setTitle! As a result, everywhere that uses setTitle will now use our new setTitle! (Even code that is in JFrame will use our setTitle method. Recall how the polymorphism rules

about how Java determines which method is called.

- For the most part, we want all the places that call setTitle to call our method so that the size is properly displayed.

EXCEPT: inside turnDisplayOnTitle, we do not want to call our new setTitle. We really want to call JFrame's setTitle.

So, we have to change each

this.setTitle(...)

to

super.setTitle(...)

Do we also need to override getTitle?

That depends, do you want the code calling getTitle to retrieve what was sent using setTitle or do you want getTitle to retrieve the title with the size coordinates added?

The choice is up to you. For fun, try overriding getTitle to always return the title that was set using setTitle.

One final note:

We could also try overriding the setSize method so that the size is automatically updated on the title. While at first glance this seems like a good solution, but it turns out that the setSize method

is not the only way that the size of the window can change. We would like to make it so that -any- way the window can have its size changed causes the title display to change. We will see how to do

that later in the course.

The Java Memory Model, part 2

The memory is divided into a heap and a stack.

The heap is used to store classes and instances.

The stack is used to store information used by method calls.

Everytime a method is called, a "stack frame" is placed on the top of the stack. When the method exits, that frame is taken off the top of the stack.

The top of the stack always contains the stack frame of the currently executing method.

The stack frame contains:

1. Method parameters (including the "hidden" parameter this for non-static methods) and any local variables declared inside the method

2. Bookkeeping needed to return from a function (where in the code do we need to start executing, and what should we do with the value returned by the method?)

Compound statements (blocks of code):

When a compound statement containing a variable declaration is encountered, a "mini-frame" is placed on the stack that contains any variable declared inside the

compound statement. This "mini-frame" is removed once the compound statement is done executing.

Example 1: We did this example that is similar to the lab.

public void displaySizeOnTitle(boolean turnDisplayOn) {

....

super.setTitle("...");

....

}

Suppose we run f.displaySizeOnTitle(true)

On the stack will go a stack frame for displaySizeOnTitle. (I will call this Frame 1).

The stack frame contains space for 2 parameters: this (because the method is not static) and turnDisplayOn.

this gets assigned the address that is stored in f.

turnDisplayOn gets assigned true.

Now the method runs.

Inside the method, super.setTitle(...) is called.

On the stack will go a stack frame for setTitle. (I will call this Frame 2).

Frame 2 sits on top of Frame 1. The stack frame contains space for two parameters, this and the parameter title.

this will store the same address as is in this on Frame 1. (super refers to this, but it accesses the methods of the parent class.)

The parameter title stores the address of the String of the title we want to give the window. Note that the String is stored in the heap because it is an object.

Now the method setTitle runs. It will use the variables in the top frame of the stack.

When it is done executing, its frame, Frame 2, is removed from the stack.

The top of the stack is now Frame 1. When setTitle is done, execution returns to displaySizeOnTitle

When displaySizeOnTitle continues its execution, its frame, Frame 1, is at the top of the stack. displaySizeOnTitle can access its variables from that frame.

Once displaySizeOnTitle completes, its frame is removed and execution returns to the method that called displaySizeOnTitle.

TAKE AWAY RULE:

A local variable exists from the moment it is declared (that is when it is placed into its frame, method frame or mini-frame, until the end of the compound

statement it is in (that is when the frame it is in gets removed from the stack).

Example 2:

Computing the factorial of an integer. 5! = 5 x 4 x 3 x 2 x 1

We will use a technique called recursion. In recursion, we identify a base case (the smallest case for which the method needs to work), and we handle the base case explicitly.

For other cases, we reduce the problem to bring it closer to the base case, and call the method on the reduced problem.

What is the smallest case? 0! = 1 (by mathematical definition)

If we assume that we can correctly compute factorial for smaller values, how do we compute n! ? We note that n! = n x (n-1)!

public static int factorial(int n) {

if (n == 0)

return 1;

else

return n \* factorial(n - 1);

}

Now, we trace the behavior of calling factorial(6).

The stack places a new method frame for factorial on the stack.

1) bookkeeping info

2) the parameter n storing the value 6 (note there is no hidden parameter this because factorial is a static method)

In the execution of factorial(6), we get to the line "return n \* factorial(n-1)". This is a method call, so we must place a frame on the stack.

A new frame for factorial(5) is placed on the stack on top of the frame for factorial(6). The frame contains:

1) bookkeeping info

2) the parameter variable n storing the value 5

NOTE: The parameter variable n in the frame of factorial(5) is a DIFFERENT variable from the parameter variable n of the frame for factorial(6).

They both have the same name, but the memory location (location on the stack) is different, and they each are storing a different value.

In the execution of factorial(5), we get to the line "return n \* factorial(n-1)". This is a method call, so we must place a frame on the stack.

A new frame for factorial(4) is placed on the stack on top of the frame for factorial(5). The frame contains:

1) bookkeeping info

2) the parameter variable n storing the value 4

And the process continues. Finally, we get to the method call factorial(0). At this point we have 5 frames on the stack. At the top is the frame

for factorial(0), under it is the frame for factorial(1), etc, down to the bottom which is the frame for factorial(4). When factorial(0) returns, the

value 1 is sent to the line "return n \* factorial(n-1)" and the frame for factorial(0) is removed. Now the top frame of the stack is the frame for

factorial(1). So, when the line "return n \* factorial(n-1)" is executed, the return value of 1 is used, and the value of the parameter variable n is 1.

At this time, we multiply 1\*1 and the method returns 1. When the factorial method returns, the top frame is removed from the stack. The frame underneath is now the top frame.

That frame has a variable n storing the value 2. So, when the line "return n \* factorial(n-1)" is executed, n stores 2, the value returned was 1, and the

multiplication is 2\*1. Thus, the value of 2 is returned.

When the factorial method returns, the top frame is removed from the stack. The frame underneath is now the top frame.

That frame has a variable n storing the value 3. So, when the line "return n \* factorial(n-1)" is executed, n stores 3, the value returned was 2, and the

multiplication is 3\*2. Thus, the value of 6 is returned.

When the factorial method returns, the top frame is removed from the stack. The frame underneath is now the top frame.

That frame has a variable n storing the value 4. So, when the line "return n \* factorial(n-1)" is executed, n stores 4, the value returned was 6, and the

multiplication is 4\*6. Thus, the value of 24 is returned.

When the factorial method returns, the top frame is removed from the stack. The frame underneath is now the top frame.

That frame has a variable n storing the value 5. So, when the line "return n \* factorial(n-1)" is executed, n stores 5, the value returned was 24, and the

multiplication is 5\*24. Thus, the value of 120 is returned.

When the factorial method returns, the top frame is removed from the stack. The frame underneath is now the top frame.

That frame has a variable n storing the value 6. So, when the line "return n \* factorial(n-1)" is executed, n stores 6, the value returned was 120, and the

multiplication is 6\*120. Thus, the value of 720 is returned.

Finally, our call to factorial(6) returns the value 720.

Note that recursion is another way to create loops, but we did not explicitly use a Java loop like "for" or "while". Instead we used if statements and method calls.

Note that Java does not handle recursion very well in that recursion has the potential to use up all the stack space. Other languages do a better job, but for Java (and C),

you need to consider how deep the recursion will be (how many active frames will be on the stack at a time), and only use recursion for situations where the stack will not

be used up. Except for the memory problem, recursion is a very useful technique. Recursion mimics our problem solving creativity better than loops do. A common technique is to first

develop an algorithm for a complex problem using recursion. We then analyze the solution and if we are lucky that not too many stack frames will be active at a time,

we are done. Otherwise, if the recursion is going to be too deep, we rewrite the algorithm replacing the recursion with loops.