The Java API

Java comes with thousands of pre-defined classes.

The listing for these classes is in the Java API (Application Programming Interface)

The API lists -everything- about a class, and much of the information is unneeded for most applications.

The important things to look for:

- At the top, the package the class is in is listed. Use this to determine what to import.

- Next comes the class declaration, as it appears in the Java file creating the class.

- That is followed by a description of the class, sometimes long, sometimes short, and can often be skimmed.

- Next comes a list of the classes inside this class, then the fields, and then the methods.

- Look for the method needed. It might be in the class, or it could be inherited from a previous class.

Clicking on the method name will send you to the description of the method.

The key thing to note is the return type and the parameter signature (how many inputs and what types). You need these to use the method.

Remember: if the methods use classes you do not know or understand, you probably do not need to use the method. Stick to methods whose input and output types are things you know.

(For now, that means limit yourself to methods with primitive types, JFrame, or String. We will slowly get used to more types.)

Polymorphism

In the API, we see that :

JFrame extends Frame

Frame extends Window

Window extends Container

Container extends Component

Component extends Object

Object is at the top of every class hierarchy so Object extends nothing.

If we create an instance of JFrame with

new JFrame()

the object created is type JFrame, but it is also type Frame, and type Window, and type Container, and type Component and type Object.

It is all these types. This is called "polymorphism" for "many types". (Note that primitive values can only be one type.)

There are two important types with every object:

The true type (run-time type) is the what the object "really is" as created by the new operator. So new JFrame() creates an object whose true type is JFrame.

The current type (compile-time type) is how the object is being used at a specific operation in the code. You can always tell the current type by looking at

the type of a variable, typecast, return type, etc.

The true type of an object never changes while the current type can change from line to line.

Java is strongly-typed so the Java compiler will verify that you always use the current type appropriately. You can never access a method or a field of an object

if that method or field does not exist in the current type.

Here is an example:

JFrame j = new JFrame(); <- creates an object whose true type is JFrame and stores it in a variable of type JFrame.

j.setSize(300, 500); <- legal because setSize is a method available to the current type JFrame. (The current type of the object is JFrame because that is the type of the variable j.)

Window w = j; <- legal because the object whose address is stored in j is a Window as well as a JFrame. (No typecast is needed because Window is "wider" than JFrame. When A extends B, B is wider than A.)

w.setSize(300, 600); <- also legal because setSize is a method available to the current type Window. (The current type of the object is Window because that is the type of the variable w.)

Object o = w; <- legal because the object whose address is stored in w is an Object as well as a Window (and as well as a JFrame).

o.setSize(400, 500); <- ILLEGAL! Even though the object's true type is JFrame, the current type is Object (the type of the variable o), and setSize is not a method available to Object.

w = o; <- ILLEGAL! Window is a narrower type than Object becaues Window extends Object

w = (Window)o; <- Legal in this case because the object stored in o has true type JFrame. JFrame extends Window so this object can have its current type be Window.

Note that the current type of an object can be its true type or any class that the true type extends. It can not be a class that extends the true type.

(Also note that this means that Object can be the current type of all objects.)

An Interesting Java Class, more on method design, and the keyword "this"

In lecture, we built GeometricFrame, a class that adds new features to JFrame.

First, we create the class structure:

public class GeometricFrame extends JFrame {

(Remember to import something!)

Now, all the new features of that we want to add to GeometricFrame go in the class body: between the { and }

Everything we place inside the class will be indented. This is the professional Java style (and general coding style) used to make clear what is inside the class and what is not.

We will add three methods:

1) transpose: flips the height and with width of the window.

A method is: access-modifier return-type name(input variables) {

body

}

What modifier? public: we want to be able to use this method anywhere.

What return type? void: there is nothing we need to return. (Hint: a method should just do what its name says it will do. Any additional actions will only confuse the programmer.)

What name? transpose: a great choice!

What input values? No input. We want to just type j.transpose() to flip the window, similar to j.getHeight() to get the height.

public void transpose() {

Now, how do we get the height and width of the window? We use the getHeight() and getWidth() methods. But whose height and width?

When we type g.transpose() we want g's width and height, but g is not declared inside the method body or the class.

The Java keyword we need is "this".

this is a special variable that exists inside instance methods.

this stores the address to the object that the method is acting on.

this acts a hidden parameter to the method. We do not see it as input to the method, but it is. When we call g.transpose(), Java will take the value stored in g (an address for an object) and place it into this.

In class, we were very clever and wrote the method without variables. You might think we need variables to remember the height and width before switching them:

public void transpose() {

int height;

int width;

height = this.getHeight();

width = this.getWidth();

this.setSize(height, width);

}

We could have combined the variable declarations and their assignment.

public void transpose() {

int height = this.getHeight();

int width = this.getWidth();

this.setSize(height, width);

}

However, as we saw in class, we do not need the variables. You only need to use a variable if you are going to use the saved value more than once. Here, we only use each variable once. So, there is no need to remember the value:

public void transpose() {

this.setSize(this.getHeight(), this.getWidth());

}

When can you combine methods calls? Anytime! All that matters in Java is the type. setSize expects two int values as input. Any two expressions that give a value of type int can be placed inside the parentheses.