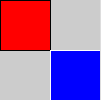
**Using Java Specifics in Processing**

**Carson Foster**

Now that we’ve talked about the necessary tools to draw basic sketches with Processing, you need to know how to transfer the rest of your enormous store of Java knowledge over to Processing. That’s what this lesson is for: we’ll cover topics like classes, inheritance, generics, functions, using Java libraries, and using Processing in Java. However, if you ever wonder about some specific feature of Java that I didn’t cover here, it’s a safe bet to assume that it also works the same way in Processing. Try it in the PDE and see if it works!

Classes, Objects, and More

We all know that classes are like blueprints, and that they are helpful constructs for programming. Never fear, you can use the beloved concept in Processing as well. In fact, they have the exact same syntax as in Java, with only a few minor differences. First, let’s look at an example of a simple class in Processing.



As you can see, this class is formed exactly as a Java class would be. However, there are two specific details that I want to point out about this example. First, in Processing, if you do not declare an access modifier (like private), then a method of property will be public by default. This differs from Java, where leaving off an access modifier corresponds to the “default” access modifier, which is different from the other modifiers. Finally, the access modifiers do not act as you would expect them to:



This prints out "You can’t read this!" twice. This might seem odd, but it makes sense once you consider how Processing works.

Essentially, all of this code in being enclosed by a Java class that extends a class that Processing provides, called the PApplet. This class essentially represents any Processing application. All of the code that you write in the PDE is wrapped by this class, meaning that any classes you write inside of the PDE will be nested, inner classes of the PApplet subclass. Therefore, the private instance variables are private within the enclosing class, not within the inner class. This means that code outside of the inner class, but not outside of the file, can still access private properties and methods.

There is another effect of this: Java is specified such that inner classes cannot have static properties or methods, so none of the classes you write in the PDE can have static variables or methods either. This sounds fairly restricting, but we’ll look at how to use Processing commands in pure Java later in this lesson, which removes a lot of the limitations necessarily imposed by only using the PDE.

Interfaces and Abstract Classes

Interfaces are also present in Processing, and they work the same as in Java. I have noticed two things, however. First, the PDE will not let you use static methods in interfaces, due to an outdated Java version. Despite this, you can still use static methods if you convert to Java 1.8+, or use another IDE with a more recent Java version. And yes, unlike inner classes, inner interfaces can still have static variables and methods. Secondly, I was unable to implement an interface and then use the implementing class in the mode of Processing that we’ve been using so far.

To understand what I mean by this, we first must understand the two ‘modes’ of Processing: active and static. So far, we’ve been using Processing in the static mode, where there is just one set of instructions to do once. Some things can be used in static mode, like classes (see example). However, other things require you to use the other mode, which is active. Active mode refers to when Processing performs a setup set of instructions, and then performs a separate set of instructions every frame. This will allow us to animate later, and also allows us to use more complex programming constructs, like interfaces.



As you can see, the active mode is very different from the static mode. The setup function (we’ll discuss other function specifics later) contains the code that is run once, at the very beginning of execution. The draw function contains the code that is run once every frame. You can still have variables outside of these functions, but these become “global” variables that are accessible to the whole program. On a side note, the draw function is not required to use active mode.

Now, let’s look at an example interface in Processing.



This program simply prints output. As you can see, we use the active mode here. The actual code for interfaces and the implementation is exactly the same as in Java. In addition, abstract classes work the same as in Java, but as they are (once again) inner classes, static members and methods are forbidden.



Note that active mode is not needed here.

Inheritance

There are no changes to how inheritance and polymorphism work. It is the same as in Java. As a good rule of thumb, mechanics in Processing work the same as in Java overall, but might have slightly different syntax. However, inheritance shares the same syntax from Java, and both inheritance and polymorphism work as in Java.

Functions

Functions are similar to methods in Java, and you might have seen me use the two terms interchangeably. However, there is a slight difference. In general, functions are not attached to classes or other constructs, while methods are. In Java, all reusable, procedural blocks of code are attached to classes, meaning that there are no true functions in Java. In Processing, as we have seen with the setup and draw functions, code that is not associated with a class is allowed. (Yes, technically, they are still associated with the wrapping Java PApplet class, but they are not associated with any user-defined classes).

Treat a function as a method without an access modifier, and you’ll be fine. This means that functions have all of the same properties as methods: they can take in parameters, return values, not return values, and have their own scope with local variables. Let’s look at an example:



As you can see, we use the active mode here. This is required by Processing; functions and the static mode do not mix. Like a method, a function is reusable chunk of code that can be used multiple times, as seen here. There’s not really a whole lot of complexity to this topic, so we’ll move on to our next topic.

Generics

Now, we’ll discuss how to use generics in Processing. First, within a class declaration, generics work exactly the same as in Java. However, in the PDE, either due to an old Java version or a specific Processing oddity, you cannot use the convenient diamond operator (when you use <> instead of <class> again) when instantiating a new object with generics. Once more, however, this can be worked around by using Processing libraries in regular Java, which we will cover later in this chapter.



As you can see in this example of generics, the syntax is the same as in Java, but I had to create a new Gimme<B>(arr) instead of a new Gimme<>(arr).

In addition, you can use generics with functions in Processing. This is simply a reflection of using generic methods in Java, so it works the same way. However, I was not familiar with generic methods, so I’ll provide an explicit example of the syntax for those of you who are also unfamiliar with them (but in Processing, of course). The method signature syntax looks like this:



To call a generic function/method, you call it like normal (in Processing). The Java VM can infer the types of the passed arguments using something called, wait for it, type inference. In pure Java, you can explicitly specify the types of the arguments when you call the method by putting <Type> right before the function name, like so: <Integer>func(1, 2).

Using Java Libraries

Now that we know how to appropriately use the many constructs of Java in Processing, it is time to learn how to use the Java libraries. Spoiler: it happens to be the exact same as in Java. The import statements are the same, and using them is also the same. Of course, keep in mind that you can’t use the diamond operator in the PDE, so that is one small change from vanilla Java. Let’s look at a quick example on the next page and then move on:

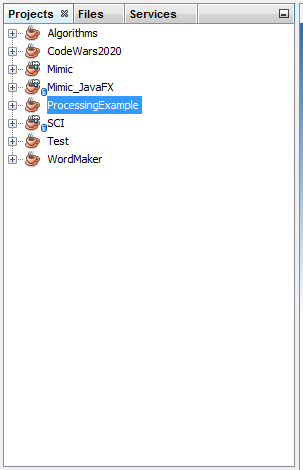


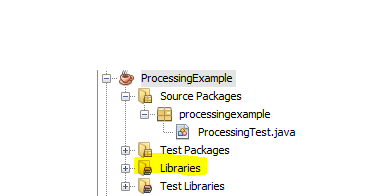
As you can see, using an ArrayList is as easy as ever in Processing.

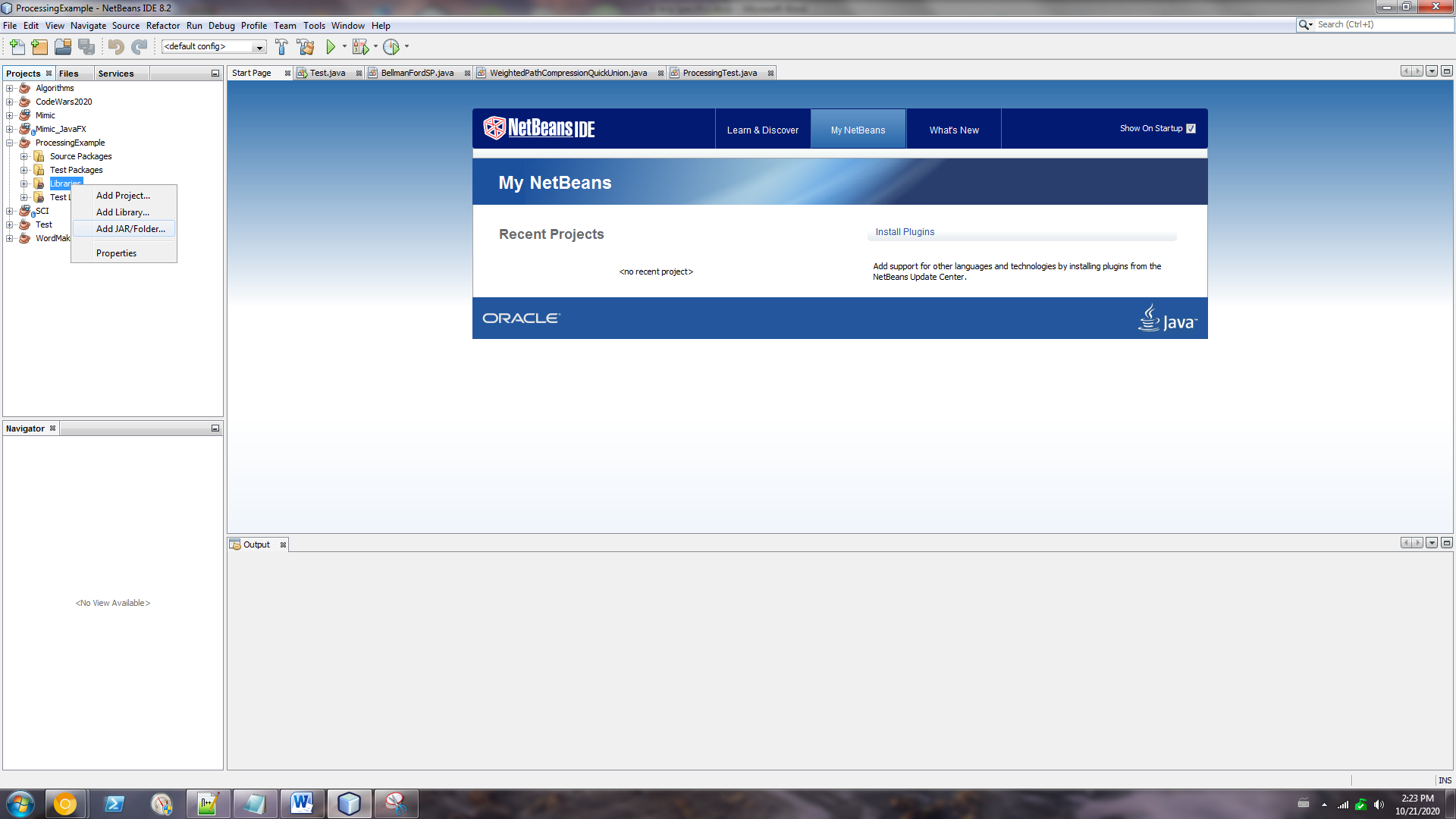
Using Processing in Pure Java

Finally, we come to coveted section: how to use the Processing libraries in Java. This is not recommended by the developers for either coding beginners or complex projects, but here’s how to do it. I’ll provided explicit directions for NetBeans, but you can follow the same general steps for other IDEs to get it working as well.

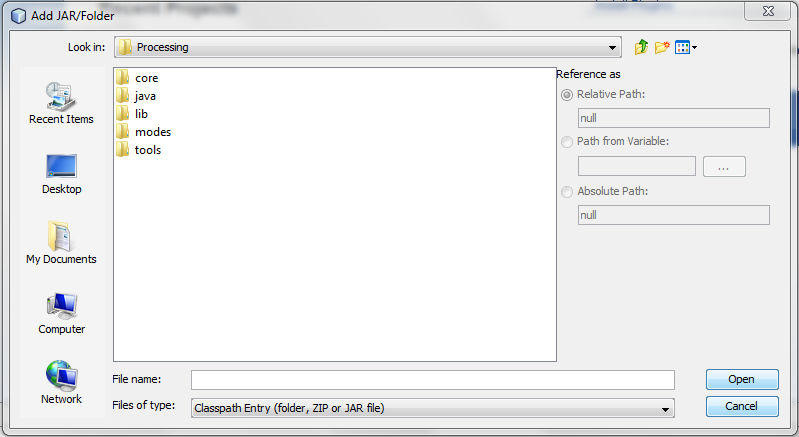
1. Add the Processing core JAR file to your project library set.
2. In NetBeans 8.2, find your project in the Projects Tab.

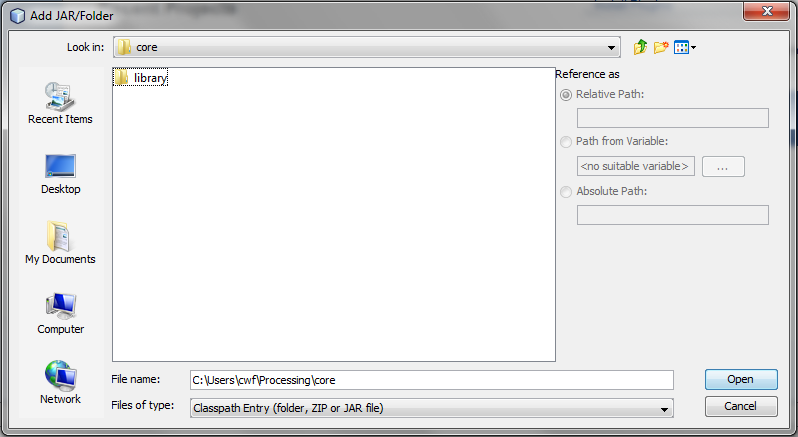


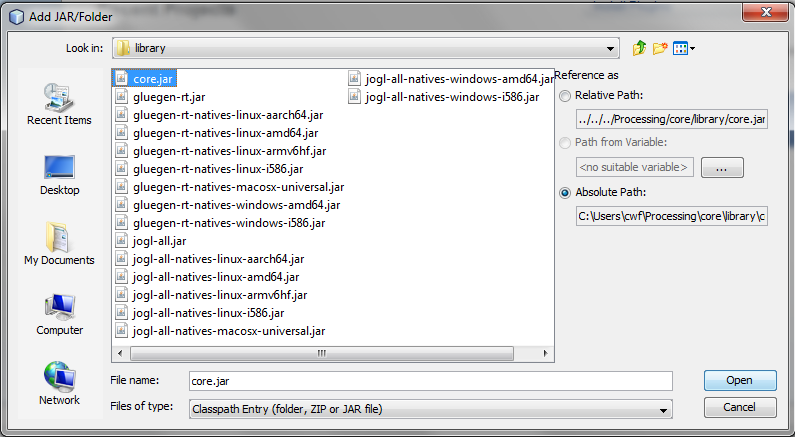
1. Then, find the Libraries section in your project. 
2. Right click on it and select “Add JAR/Folder”.



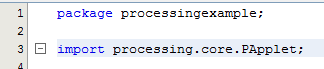
1. Find the location where you installed Processing, and navigate to “Processing\core\library”, where “Processing” is the location where Processing is. From that folder, select “core.java” to add.







1. Create the class outline
2. Start with your package statement and import the necessary Processing core.



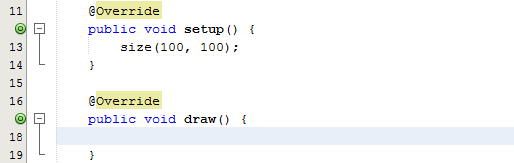
1. Define your class and make sure it extends PApplet.



1. Write the main method. This must be in the exact following format. Of course, replace my package and class names with yours.



1. Use active mode, so override the **setup** method and, optionally, the **draw** method.



1. Use Processing functions, transformations, shapes, etc. in Java, with more freedom! Anything you did in Processing can be pretty much exactly transferred back over in Java, keeping in mind that you should make your access modifiers explicit in Java, and that you must convert static mode to active mode by simply wrapping your statements in the **setup** method.