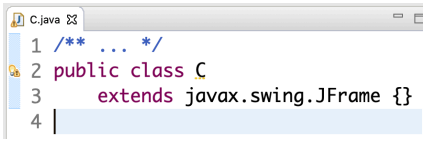
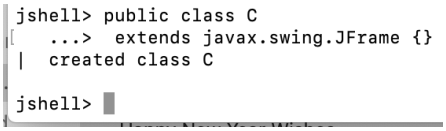
Playing with small classes in JShell

At times, we want to play around with small classes to learn about some feature of Java. We might be able to do this directly in Eclipse, but the need to have a static procedure main in order to execute anything makes it awkward to play around. We show here how to use Eclipse to write the classes and get them syntactically correct but then use JShell to execute things.

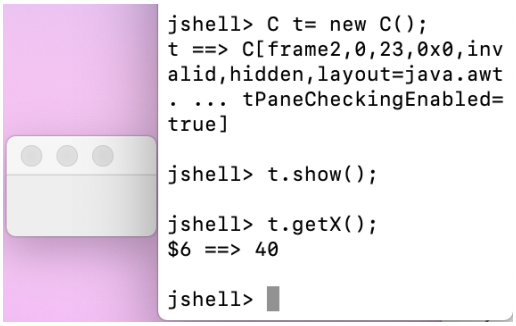
Suppose we write class `C` in Eclipse, as shown to the right, and we want to see that an object of class `C` really contains the methods of class `JFrame`.



To do this, simply copy the whole class and paste it into JShell and hit the return key. The image to the right shows what JShell look like. It created class `C`.



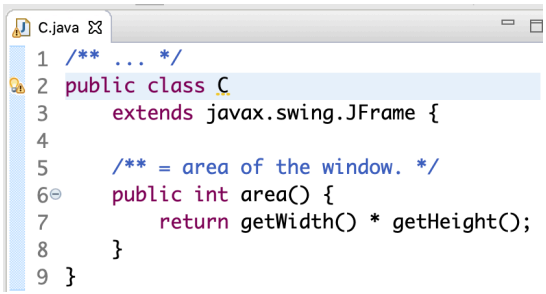
We can now create an object of class `C` and call its method `show()`. Several things to note here.
(1) You see the value of `t`; it is a description of the object caused by calling its method `toString()`. If you don't know about this yet, just gloss over it.
(2) To the left of the command-line window, you see the window associated with the object. We made it bigger and dragged it to this place so you could see it.
(3) You also see a call of function `t.getX()`, which gives the x-coordinate, 40, of the left side of the window.



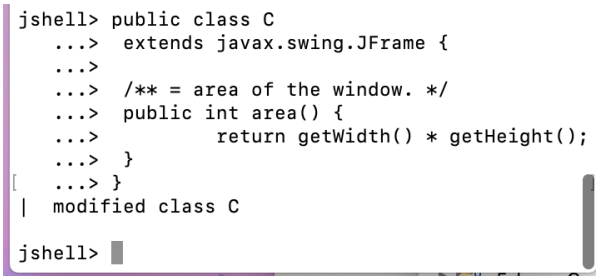
Modifying the class

Now let's modify the class in Eclipse to put in function `area`. You see the image of the class in Eclipse to the right.

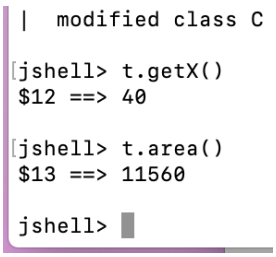
Then, we copy the class declaration and paste into JShell. This is shown in the image to the right below.



The important point to note here is that class `C` has been *modified*. The old version has been replaced by the newly pasted one.



Variable `t` still contains a pointer to the object created using the old version of class `C`, and evaluation of `t.getX()` still produces 40. Interestingly enough, however, the object has been changed to include function `area`, so that `t.area()` is evaluated and produces the size 11560.



(See next page for an example of exploring Java using JShell.)

Exploring the compile-time reference rule

This example illustrates how JShell can be used to explore Java rules and features. To the right are class `C` and subclass `S` in Eclipse, with method `m` declared in `S`. Only one class is declared public; this is a requirement when several classes are declared in one file. To the right of the classes in Eclipse, we see them pasted then into JShell.

C.java

```
1 public class C {}
2
3 class S extends C {
4     int m() {
5         return 5;
6     }
7 }
```

```
jshell> public class C {}
|  created class C

jshell> class S extends C {
   ...>  int m() { return 5;}
   ...> }
|  created class S

jshell>
```

Now let's create a new `S` object and assign it to variable `vS`. Then copy that variable into a new variable `vC` of type `C`. Thus, variable `vS` has perspective `S` on the object while variable `vC` has perspective `C`.

```
jshell> S vS= new S();
vS ==> S@907ef125

jshell> C vC= vS;
vC ==> S@907ef125

jshell>
```

We see that a call `vS.m()` calls method `m()` properly and produces the value 5. But the call `vC.m()` is syntactically illegal. By the compile-time reference rule, for the call to be legal, method `m()` must be declared in `C` or a superclass, and it is not.

```
jshell> vS.m()
$5 ==> 5

jshell> vC.m()
|  Error:
|  cannot find symbol
|    symbol:   method m()
|  vC.m()
|  ^--^
```

However. `vC` can be cast down to class `S` and then the call it legal

```
jshell> ((S)vC).m()
$6 ==> 5
```

It will take some time to get used to testing different aspects of Java using JShell like this. But stick with it and your will soon become fluent with it.