Return to Module Overview Page (https://onlinelearning.berkeley.edu/courses/665040/pages/module-ten)

10.9: The Object Serialization API

Wouldn't it be great if you could somehow save a running Java program into a file, turn your computer off, come back the next day, turn it back on, load in that file, and be back in the same application, with every window and object exactly in place? This is a terrific way to save user preferences, and an even better way to collaborate on a project with a coworker, especially if you could send her that magic file via e-mail. You can do exactly that, and it's very easy to do, thanks to Java's Object Serialization API.

Simply put, the Object Serialization API takes an arbitrary collection of objects and writes them all onto a stream. You can send the stream to a file, to the network, or elsewhere. Later, the same set of objects, with all their member variables intact, can be recreated from the data on that stream.

To use object serialization yourself, you simply need to create an ObjectOutputStream and call writeObject( ), passing some object as an argument. writeObject( ) will save all objects "reachable" from the object you pass in: all its member variables, all their member variables, and so on. writeObject( ) is clever about only saving once each object it encounters, so it won't make duplicate copies if your objects reference each other multiple times. Here's an example:

```
// This is the top-level object you want
//to save
MyApplication myapp;

try
   {
    FileOutputStream f = new FileOutputStream(filename);
    ObjectOutputStream s  =  new ObjectOutputStream(f);

    s.writeObject(myapp);
    s.close();
    f.close();
   }
catch (IOException ioe)
   {
    System.err.println("Error serializing application:"+ioe);
   }
```

The file "filename" now contains a representation of the state of all the objects reference by myapp. You can execute this code to recreate the MyApplication object later:

```
try
   {
    FileInputStream in = new FileInputStream(filename);
    ObjectInputStream s = new ObjectInputStream(in);
    myapp = (MyApplication)s.readObject();
    s.close();
    in.close();
   }
```

```
   catch (IOException ioe)
   {
    System.err.println("Error reading file: " +
                       ioe);
    System.exit(-1);
   }
catch (ClassNotFoundException cnfe)
   {
    System.err.println("Can't find class: " +
                       cnfe);
    System.exit(-1);
   }
```

The ClassNotFoundException can occur if the .class file for one of the serialized objects can't be found (the ObjectOutputStream does not save the entire class file for each class, only an object's data).

Only objects that implement the Serializable interface can be serialized. This is an empty interface, and implementing it just serves to signal Java that serializing your class is OK. It's probably a good habit to get into to make all of your classes implement Serializable, in case you ever decide to use serialization. Most of the Java API classes now implement this interface with a few notable exceptions. One exception is java.awt.Graphics. Classes should not implement Serializable when (like Graphics) they serve as an interface to some hardware-dependent information, or when they depend on time-dependent member data such as a socket connection (which could hardly be expected to continue to function after being saved to a file and transported to another machine!).