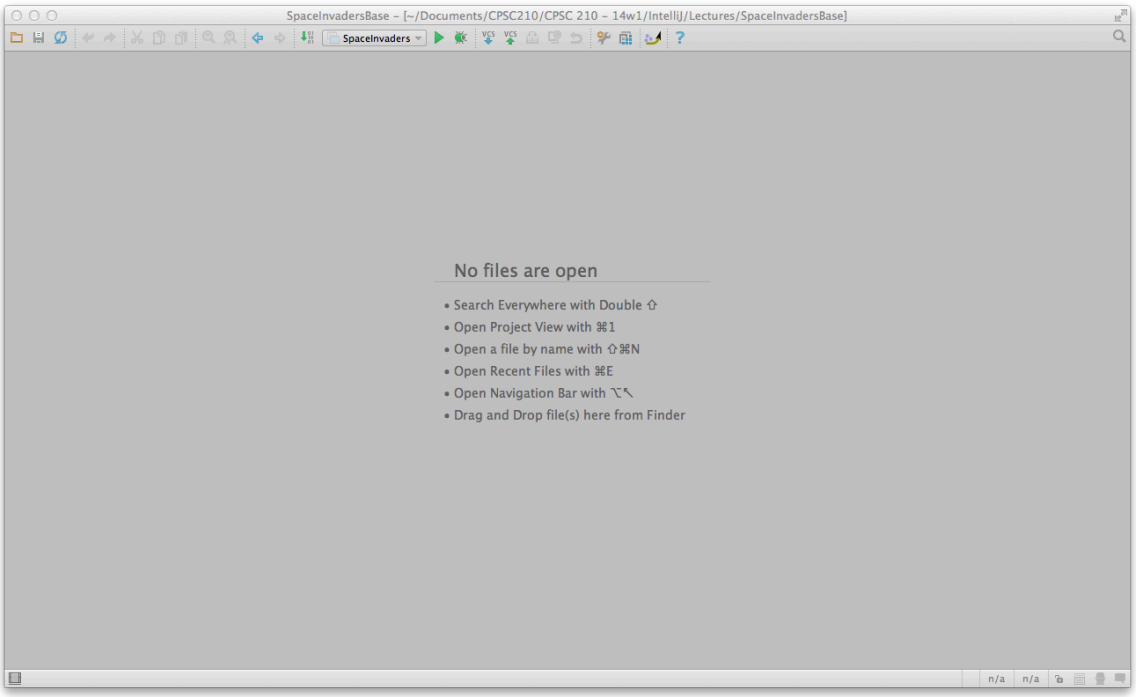
Introduction to IntelliJ

IntelliJ is a large software package used by professional software developers. This document will give you a brief introduction but is by no means exhaustive. If you have questions after working through this material, please ask in lab, DLC or office hours.

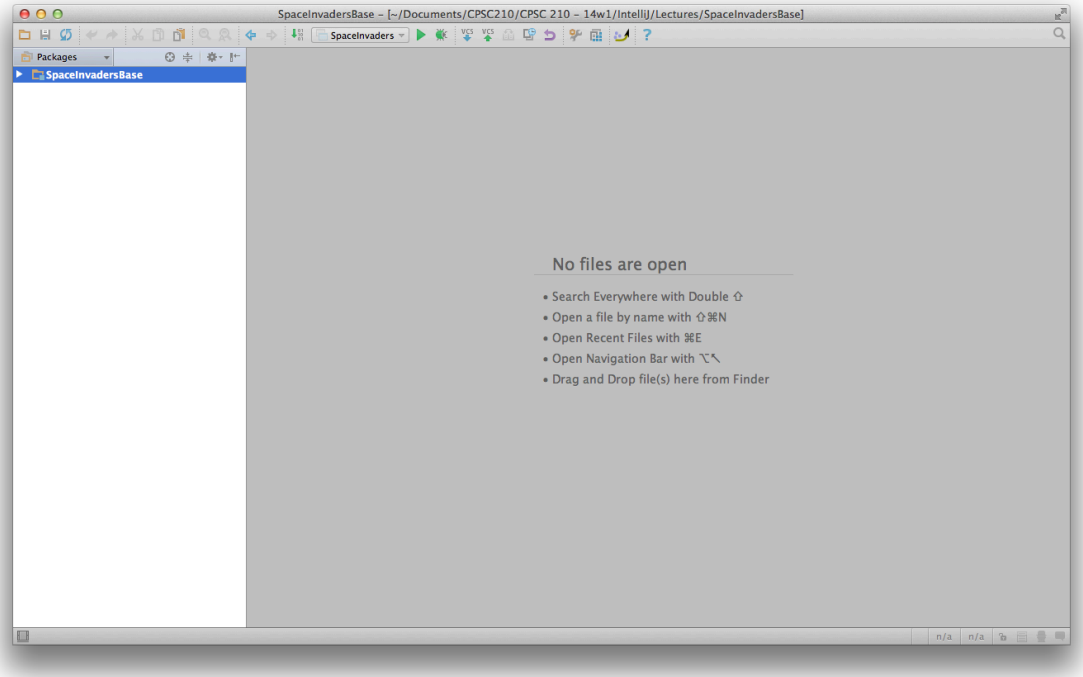
Terminology

When you first open IntelliJ, you will either see the Welcome dialog or the last project that you had opened. If a project is open and you want to return to the Welcome dialog, just close the project (File -> Close Project).

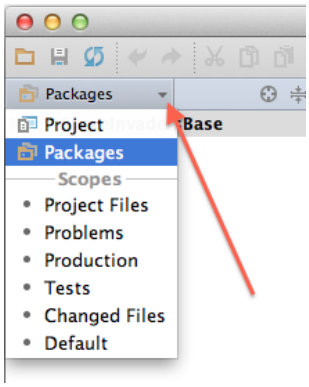
When you first checkout a project from the repository, you will likely be presented with an empty view:



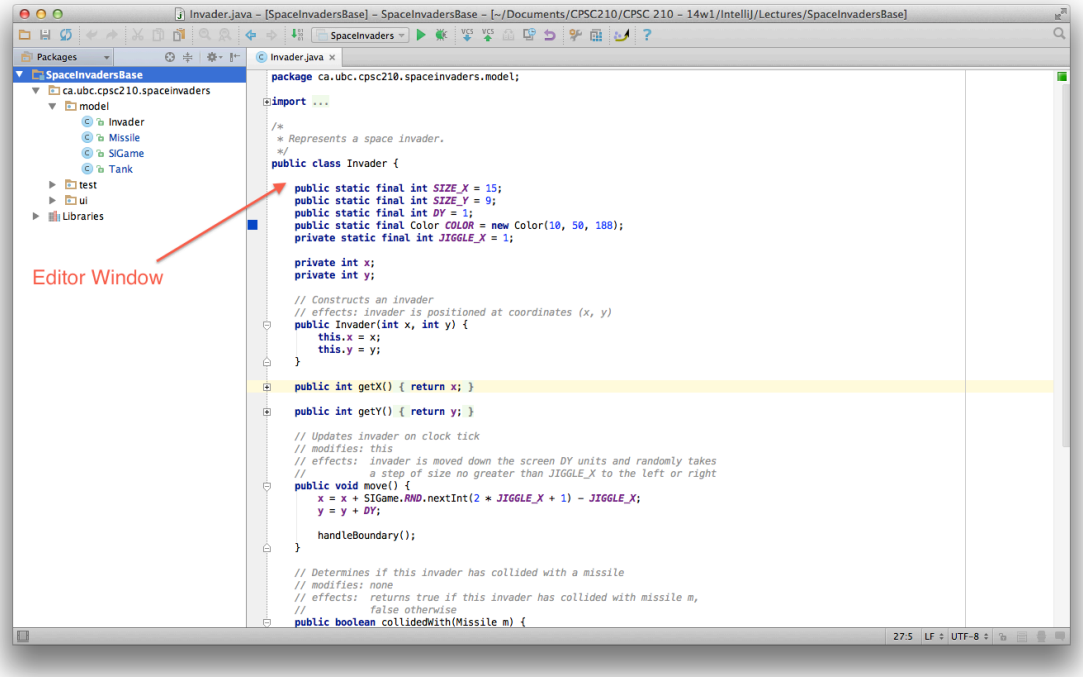
You'll probably want to start by opening the Project View. Note that the instructions for doing this appear in the middle of the screen. After opening the Project View, IntelliJ will look as follows:



The Project View enables you to access all the files associated with your project, including IntelliJ configuration files, which aren't usually of interest. The Packages view exposes only the Java packages and source files, and is often more convenient. To access the Packages view, click the down-arrow in the Project view tab and select Packages.



To open a file, expand the packages, if necessary, by clicking the triangle to the left of the package and then double-click the file to open it in an editor window:



Writing, running and debugging Java code

Scheme and BSL are interpreted languages. The interpreter parses the code and executes the instructions immediately.

Java is a compiled language. The compiler parses the code, compiles it to Java bytecode and saves it to be run later by the JVM (Java Virtual Machine). By default, IntelliJ will recompile your code whenever you try to run it. It will also warn you right away if any of the code that you type contains a syntax error.

When you write code, you are working on a file. The code is not actually running!

When you debug code, you are exploring code that is actually running. In other words, IntelliJ is running the executable file and allowing you to observe what's happening while it's running.

The debugger can therefore help you figure out what's happening when the code runs. Without the debugger, figuring out what happens when the code runs can be much more difficult. In other words, investing some time in learning how to use the debugger will really pay off in the long run.

Techniques for understanding code

1. Run the application. This can help you to understand the purpose of the application. To run an application, you must right-click on a particular class in the **Packages** view and choose **Run SomeClass.main()** where **SomeClass** is the name of the class. At the start of the course, we will always tell you which class you must select. In the case of the `SpaceInvadersBase` application illustrated in this tutorial, it is the `SpaceInvaders` class in the `ui` package.

   Having run the application once, you can easily run it again using the Run icon on the toolbar:

2. Extract models from the code, to help you understand what it does. As we go through the course, we will illustrate various techniques for extracting models from Java code.

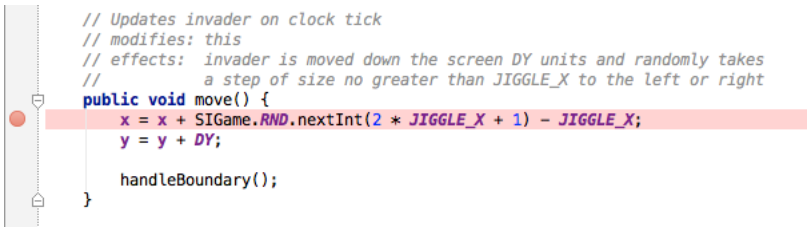
3. Run the code using the debugger. Use breakpoints, check the value of variables, step through the code and look at the call stack. To run code in the debugger, right-click the same class that you would select if you were to run the code and select **Debug SomeClass.main()**.

   Having run the application using the debugger once, you can easily do it again using the Debug button on the toolbar:

Debugging Tools

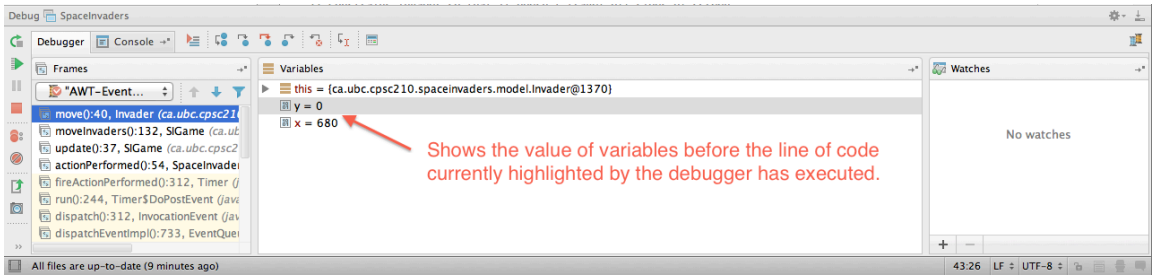
Breakpoints – the debugger will pause execution when it encounters a breakpoint. This will allow you to explore specific execution paths as the code runs. To set a breakpoint, click in the margin to the left of the code. A red dot will appear and the line will be highlighted in red:



```
// Updates invader on clock tick
// modifies: this
// effects:  invader is moved down the screen DY units and randomly takes
//           a step of size no greater than JIGGLE_X to the left or right
public void move() {
    x = x + SIGame.RND.nextInt(2 * JIGGLE_X + 1) - JIGGLE_X;
    y = y + DY;

    handleBoundary();
}
```

Check the value of variables – you can see the current value of variables in the Variables view.



Step through the code – once you've stopped at a breakpoint, you'll likely want to run the code step-by-step. You can step through the code one line at a time in order to see what happens when each line is executed.

You will see the following buttons in the toolbar of the Debug view. Hover your mouse over each of them to get a hint as to their purpose:

Show Execution Point – will highlight the statement at which the debugger has paused in an editor window

Step Over – will execute the current statement. If that statement contains a method call, it will **not** step into that method.

Step Into – will execute the current statement. If that statement contains a method call, it will move into that method (unless the method is in the Java library).

Force Step Into – will execute the current statement. If that statement contains a method call, it will move into that method (even when that method is in the Java library).

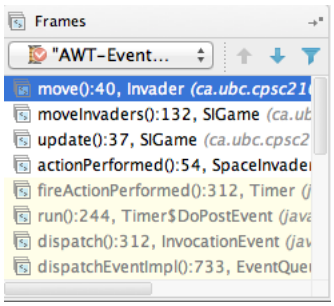
Step Out – will execute the rest of the current method, return from it, and then pause at the next statement to be executed.

Drop Frame – steps back out of the current method to the point from where it was called.

Run To Cursor – runs the code to the statement at which the cursor has been positioned.

Evaluate Expression – evaluate an expression that you specify based on the current value of variables.

Call Stack – the call stack provides information about the currently active methods. You may wish to click the filter button so that you see only the methods in the code that you've written and hide calls to methods in the Java library.



In the example shown above, `actionPerformed` called `update`, which called `moveInvaders`, which called `move`.

Controlling the Debugger – the toolbar on the left side of the Debug view allows more coarse-grained control over the debugger.

The first four – *Rerun*, *Resume*, *Pause* and *Stop* – are particularly useful. Hover your mouse over each button so you figure out which one is which.

The *Show Breakpoints* button opens up a dialog that shows you the breakpoints that have been set in your code. You can use this dialog to temporarily disable certain breakpoints.

The *Mute Breakpoints* button temporarily disables all breakpoints in your project.