CS 2334 Lab 8
Exceptions

- Exceptions provide a means of handling error conditions that arise in the course of program execution
- Example: The program prompts the user for a filename, and the use provides a nonexistent file
  - Should the program crash?
  - Or would it be better to notify the user and ask again?

Handling Exceptions

Exceptions can be handled robustly using the try-catch-finally syntax

- try: execute some code that may throw an exception
- If an exception is encountered, immediately start executing code in catch
- finally:
  - If no exception is thrown, execute after all code in the try block has executed
    - Execute after a return() in the try block before returning
  - If an exception is thrown, execute after all code in the catch block has executed

Handling Exceptions (cont.)

```
try{
  foo();
  thrower(); // might throw an exception
  bar();
}
catch (Exception e){
  caught();
}
finally {
  baz();
}
```

Throwing Exceptions

- Exceptions can be thrown using the *throw* statement:
  - throw *exception*
    - *exception* is an instance of an Exception class
- Example:

```
throw new IllegalArgumentException(“Value of
  x should be greater than 5”);
```

- Exceptions can even be thrown from within a catch block

Catching Multiple Types of Exceptions

- A block of code may throw more than one type of exception
- Multiple catch blocks can be used to handle different types of exceptions that might be thrown from within a try block
- Only one catch block is executed per exception thrown within a try block
- The first catch block to match the thrown exception is the one executed

Catching Multiple Types of Exceptions (cont.)

```
try{
   thrower1(); // might throw ArithmeticException
   thrower2(); // might throw ArrayIndexOutOfBoundsException
}
catch(ArithmeticException e){
   foo();
}
catch(ArrayIndexOutOfBoundsException e){
   bar();
}
catch(Exception e){
   baz();
}
```

Utilizing Caught Exceptions

- When an exception is thrown, we may wish to identify where the exception originates, for debugging purposes
- We can use the following function call:

```
catch (Exception e){
  e.printStackTrace();
}
```

```
java.lang.IllegalArgumentException: Bad
  parameter
  at Main.thrower(Main.java:99)
  at Main.exceptionTest(Main.java:105)
  at Main.main(Main.java:125)
```

Assertions

- An assertion is a statement that allows the programmer to test certain assumptions about some code
- If the assumption is correct, nothing happens
- Otherwise, an error is thrown

- Can be used to check preconditions and postconditions for some methods

Assertions (cont.)

- The syntax of assertions is as follows:

```
assert Expression;
```

- *Expression* is some Java expression that returns a boolean value
- For example:

```
int i = 5;
String s = “Hello World”;
…
assert i > 0;
assert s.equalsIgnoreCase(“Hello World”);
assert 1 == 2;
```

Assertions (cont.)

`assert` *`Expression`*`;`

- If *Expression* evaluates to true, nothing happens
- If *Expression* evaluates to false, an AssertionError is thrown

Errors vs. Exceptions

- Exceptions indicate an error condition that a reasonable application might want to handle
  - E.g.: a user-specified file is not found
- Errors indicate serious problems that a reasonable application should not try to handle
  - The Java virtual machine has run out of memory

Assertions (cont.)

Assertions throw Errors

- Thus, assertions should not be used to indicate problems within the scope of normal program execution
  - Assertions should indicate conditions that should never occur if the program is functioning properly
  - Assertions are generally removed from released software
- Exceptions should be used to indicate problems that can reasonably be expected to occur in various circumstances

Conventions for Using Assertions

- Don’t use assertions to check public method preconditions (i.e., that parameters have acceptable values)
  - Instead: use exceptions
- You can use assertions to check nonpublic method preconditions
- You can use assertions to check postconditions on any method

Lab 8

- Modify your existing Project 3 code to add exception handling
- By the end of this lab, you should be able to:
    1. Create exception classes
    2. Throw exceptions to indicate errors
    3. Catch exceptions to yield robust behavior

Milestone 1

Create a FinchException class

- Extends Exception
- Contains a string that describes the error
- Implements toString()

Milestone 2

Throw FinchExceptions

- Must be thrown in response to illegal constructor parameters for all FinchActions
- FinchOrientationGuarded and FinchObstacleGuarded
  - If the orientation/obstacle type string is wrong, throw an exception
- Implementation should be straightforward

Milestone 3

Catch FinchExceptions

- Create a completely new driver class
- main() should attempt to create a set of FinchActions
  - Some constructor calls should contain illegal parameters, others shouldn’t
- Report each thrown exception
- Add successfully created FinchActions to a FinchActionList

The General “To-Do”

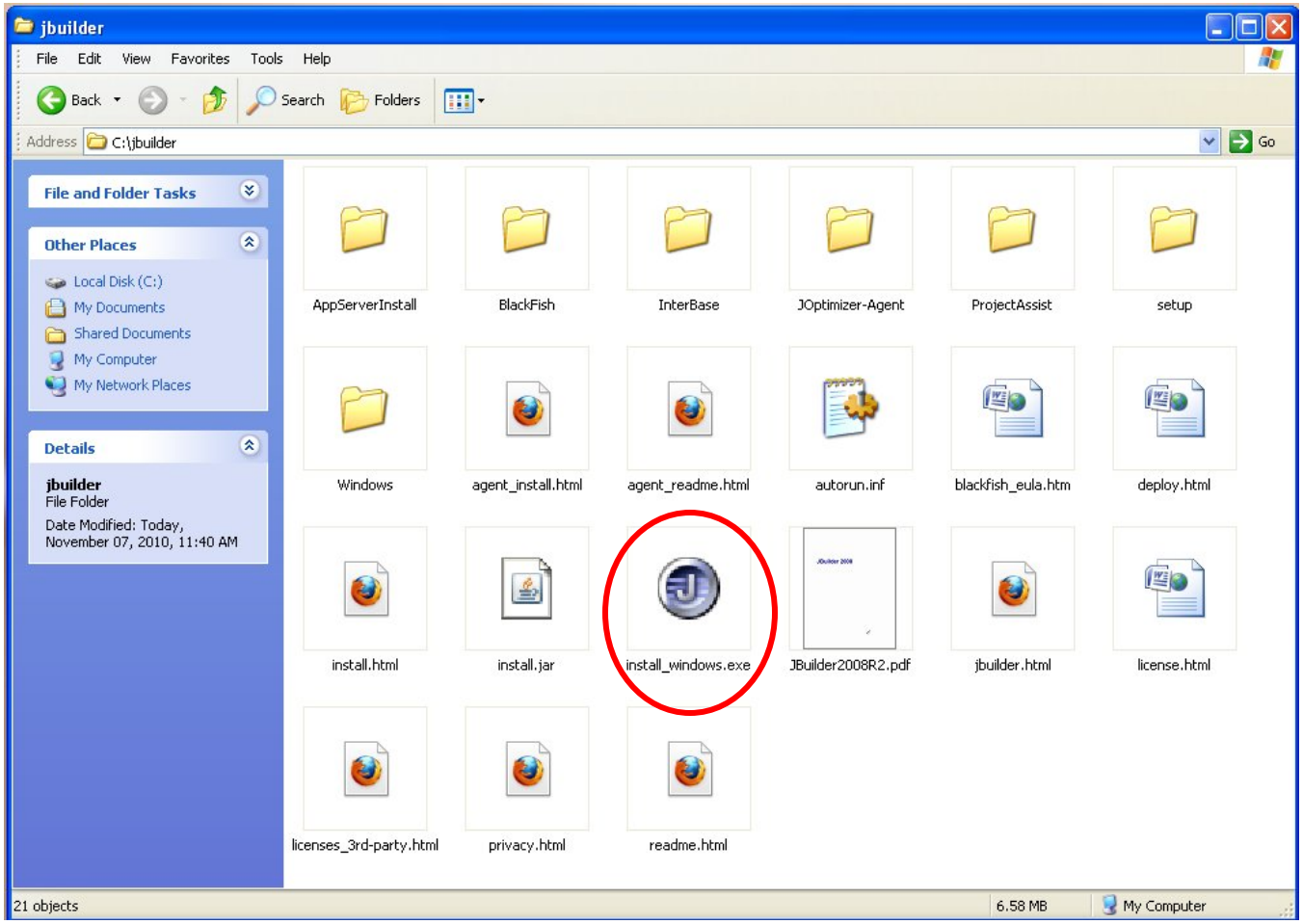
- Work on Lab 8 Today (avoid Project 4)
- Lab is due at 11:29 AM on Friday, November 12th
  - Demo
  - Electronic Submission

Testing…

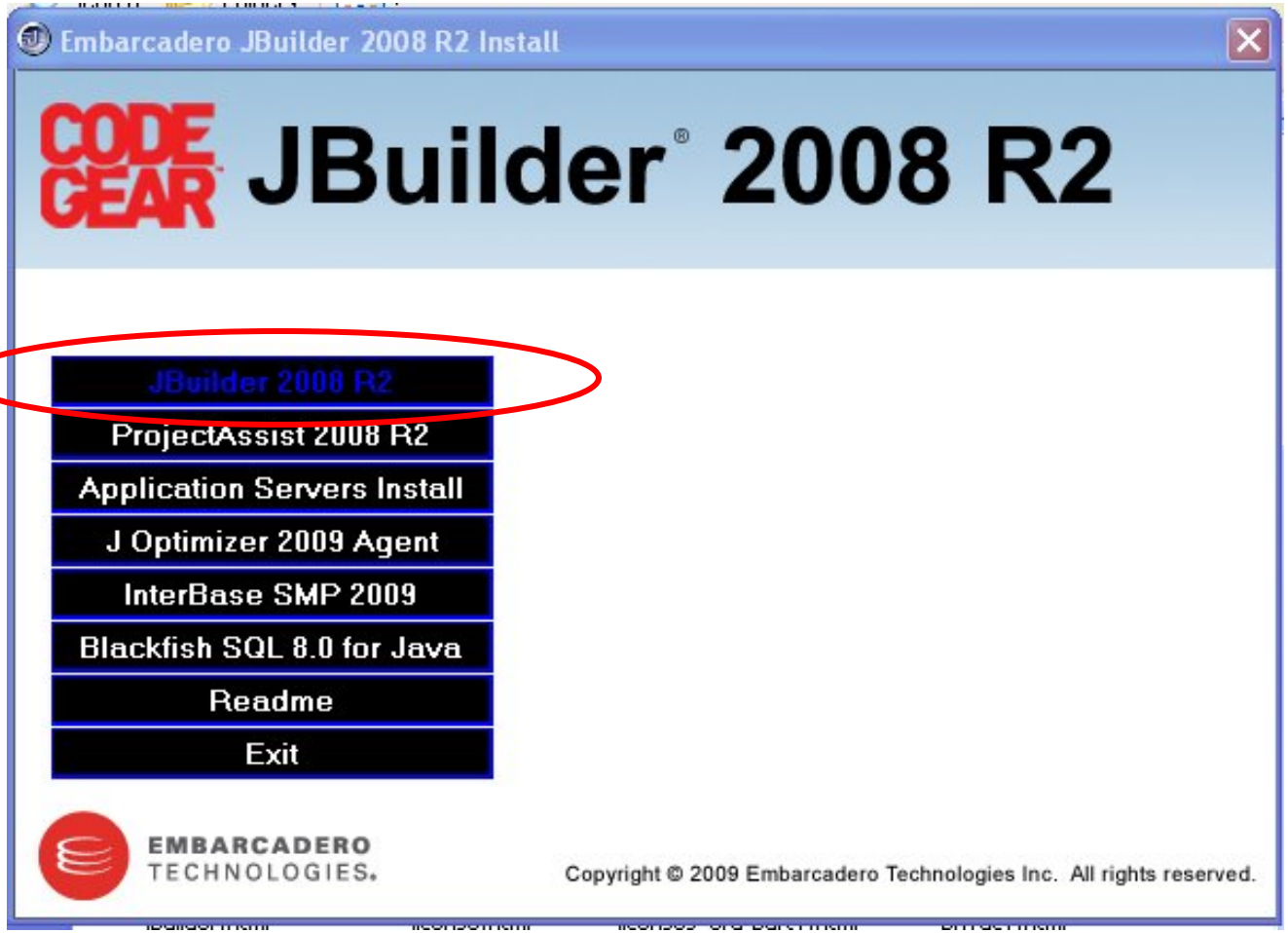
In the driver, for each FinchAction:

- Make a good constructor call
- Make a bad constructor call that does one of the following:
  - Too many/few parameters
  - Invalid orientation values
  - Invalid obstacle values
  - Invalid number formatting (if parsing strings)

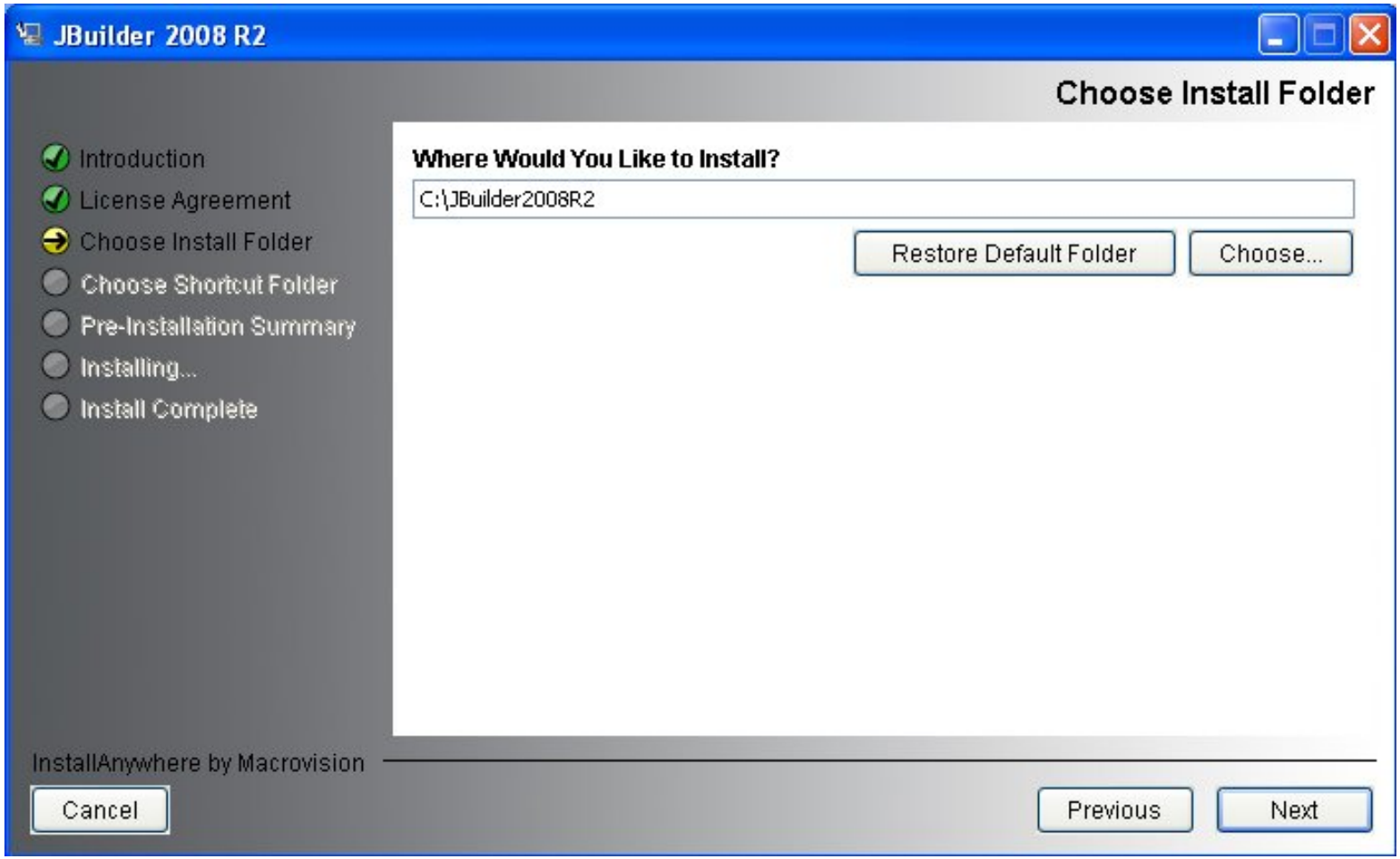
Installing JBuilder



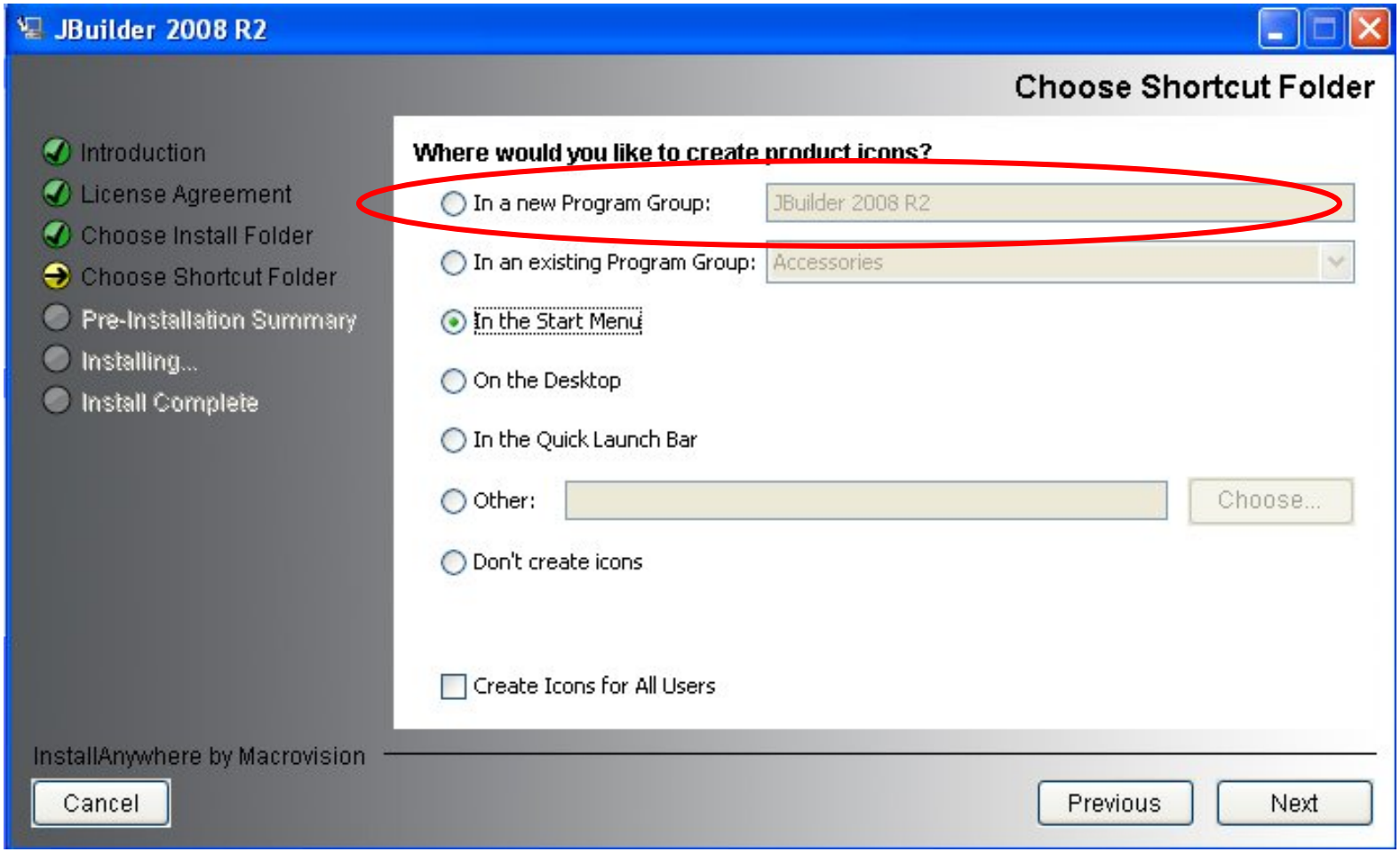
Choose JBuilder 2008 R2



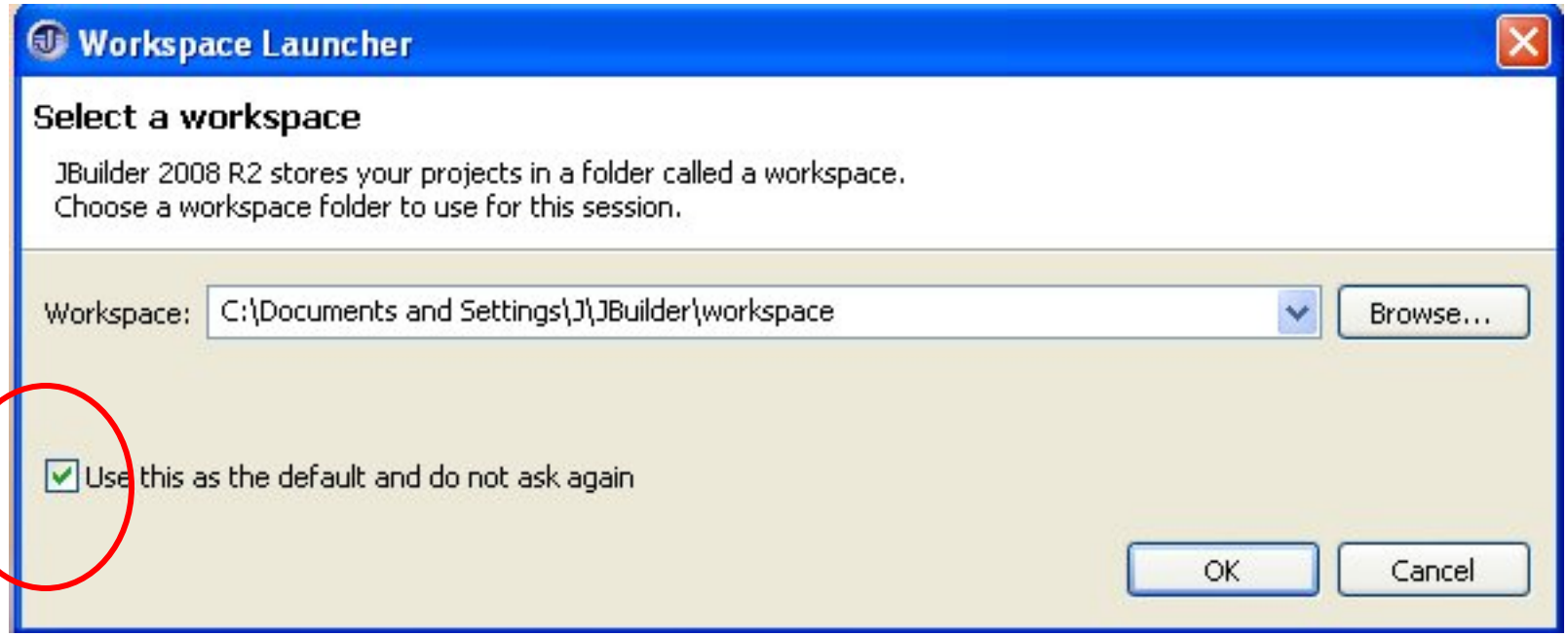
Choose an install path



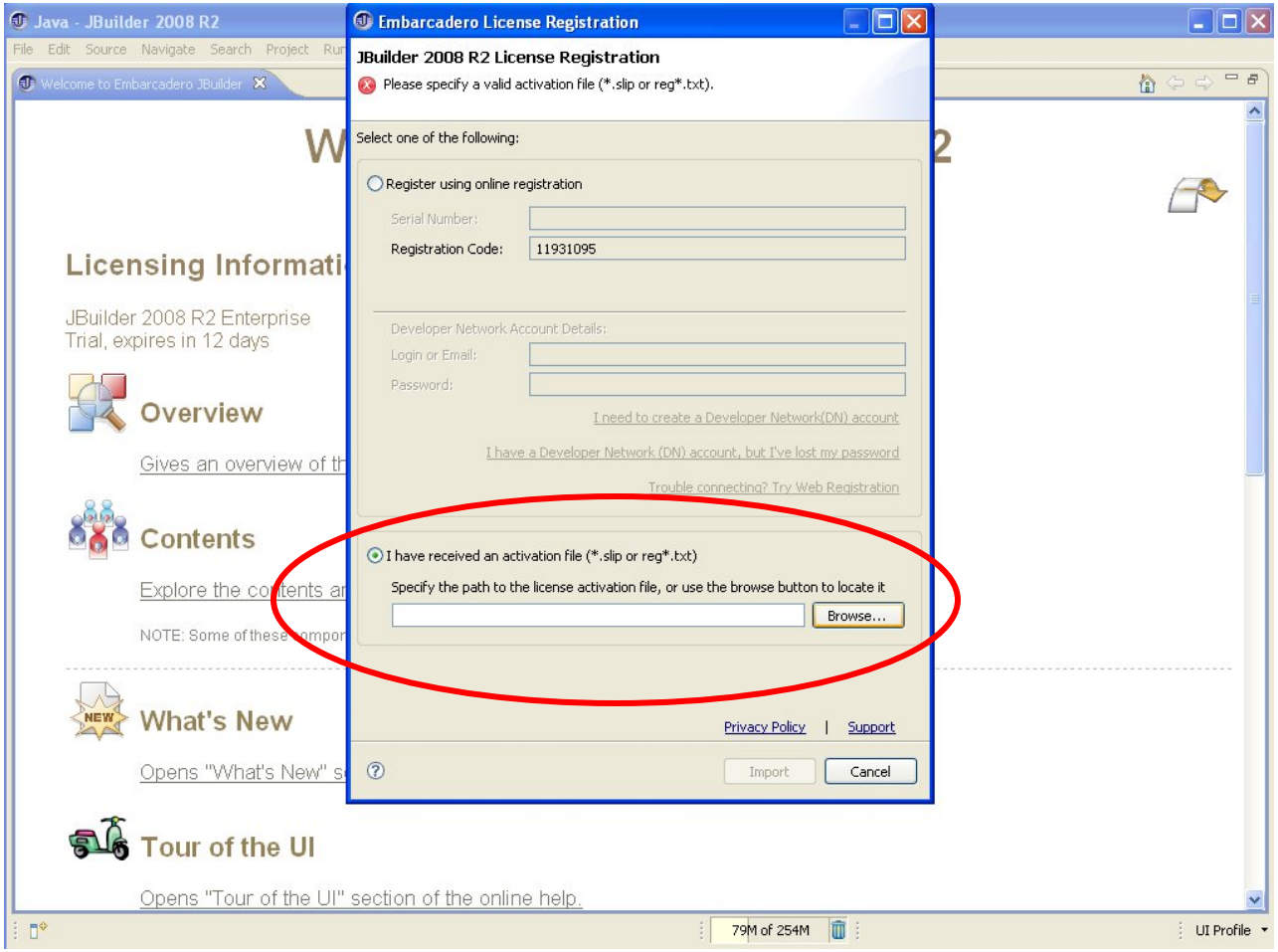
Choose where the shortcuts go



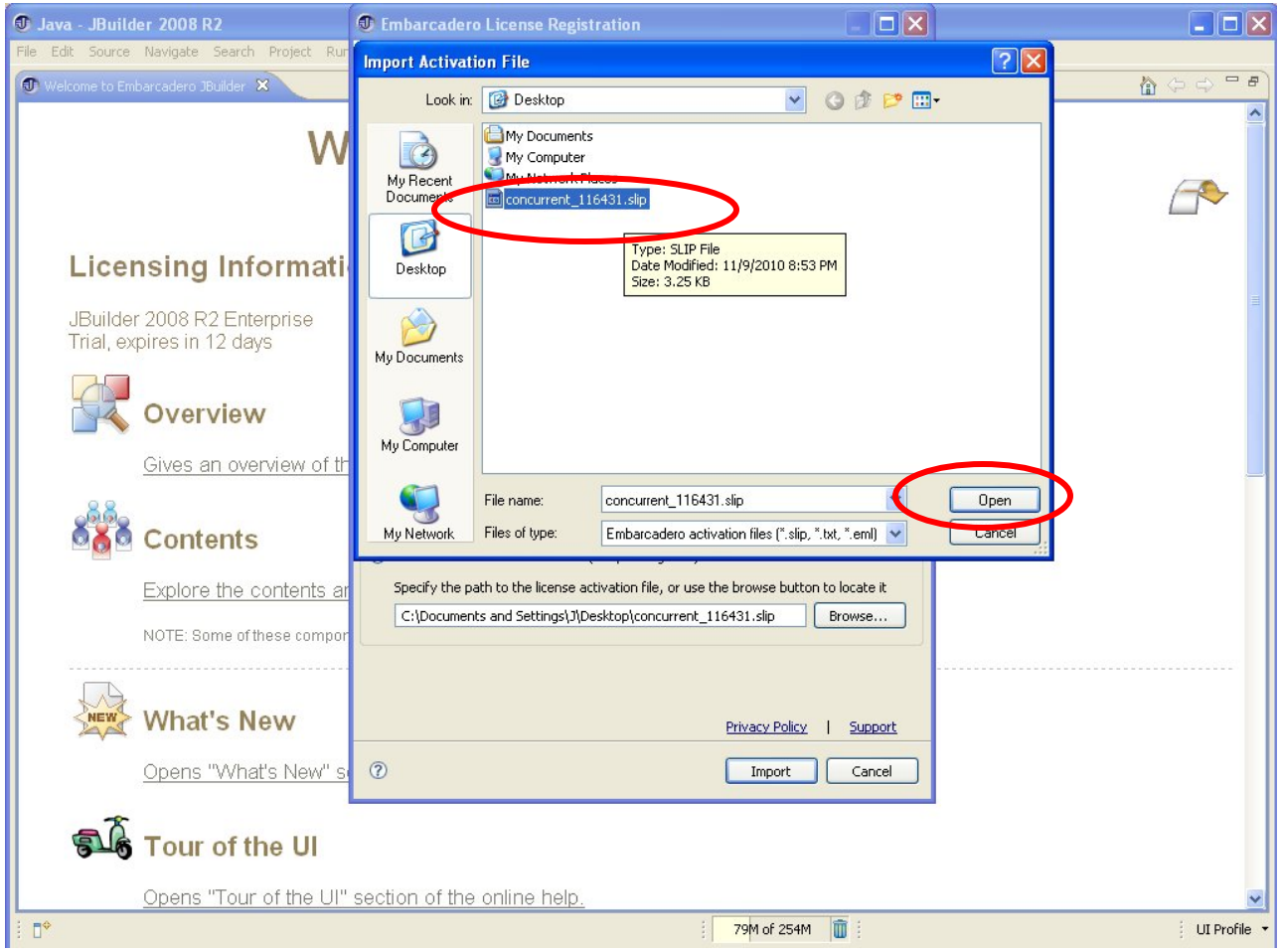
Select your workspace folder



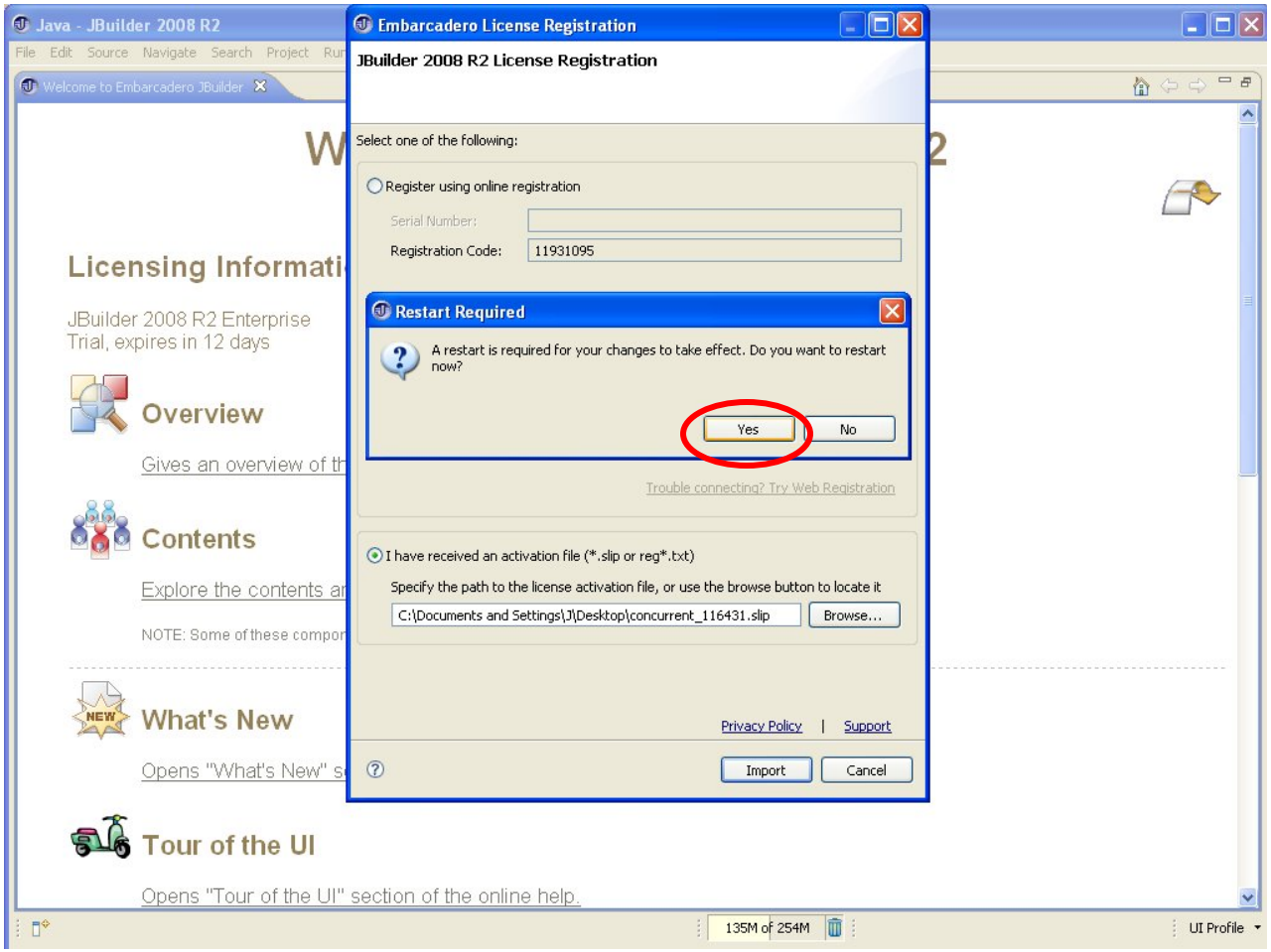
Choose to load an activation file



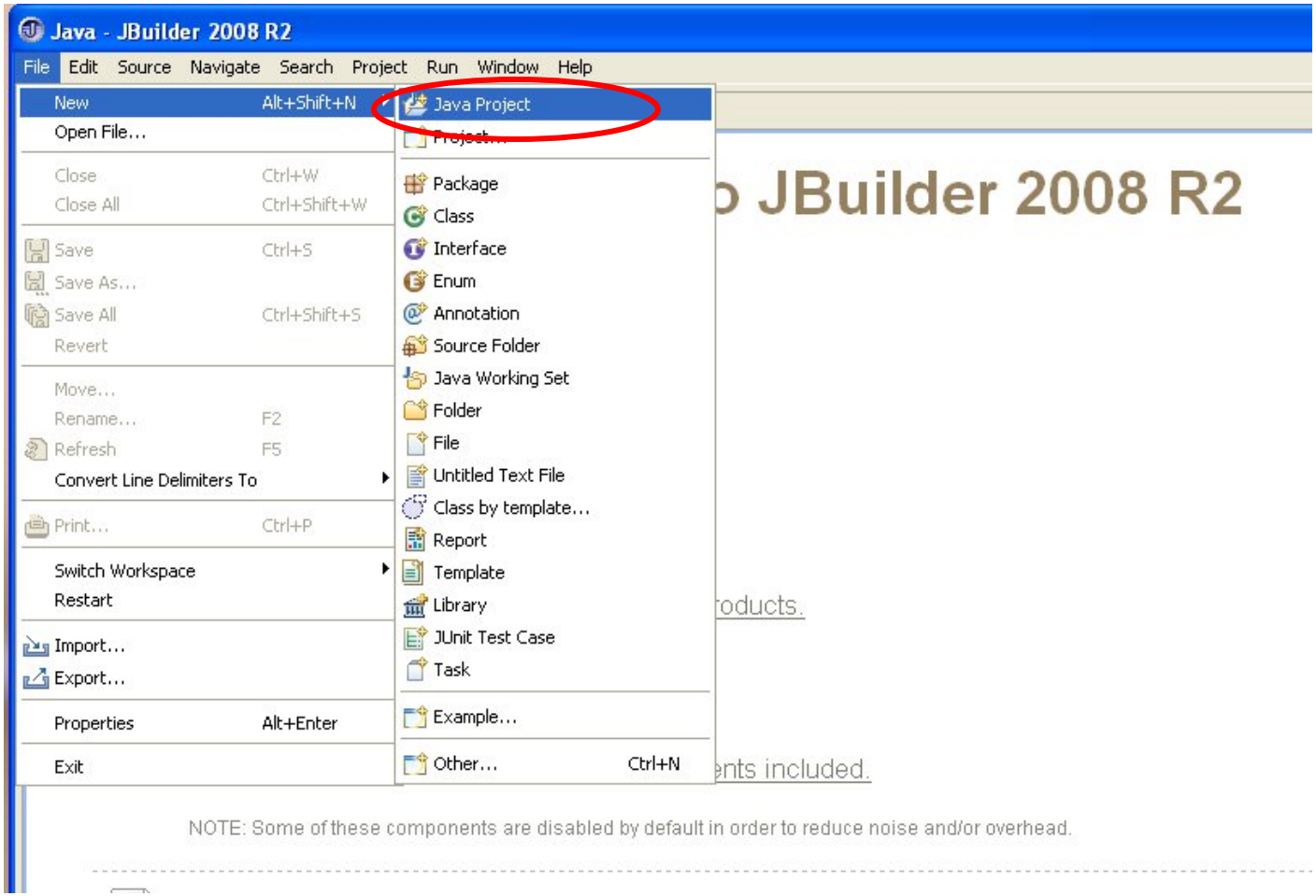
Select the slp license file



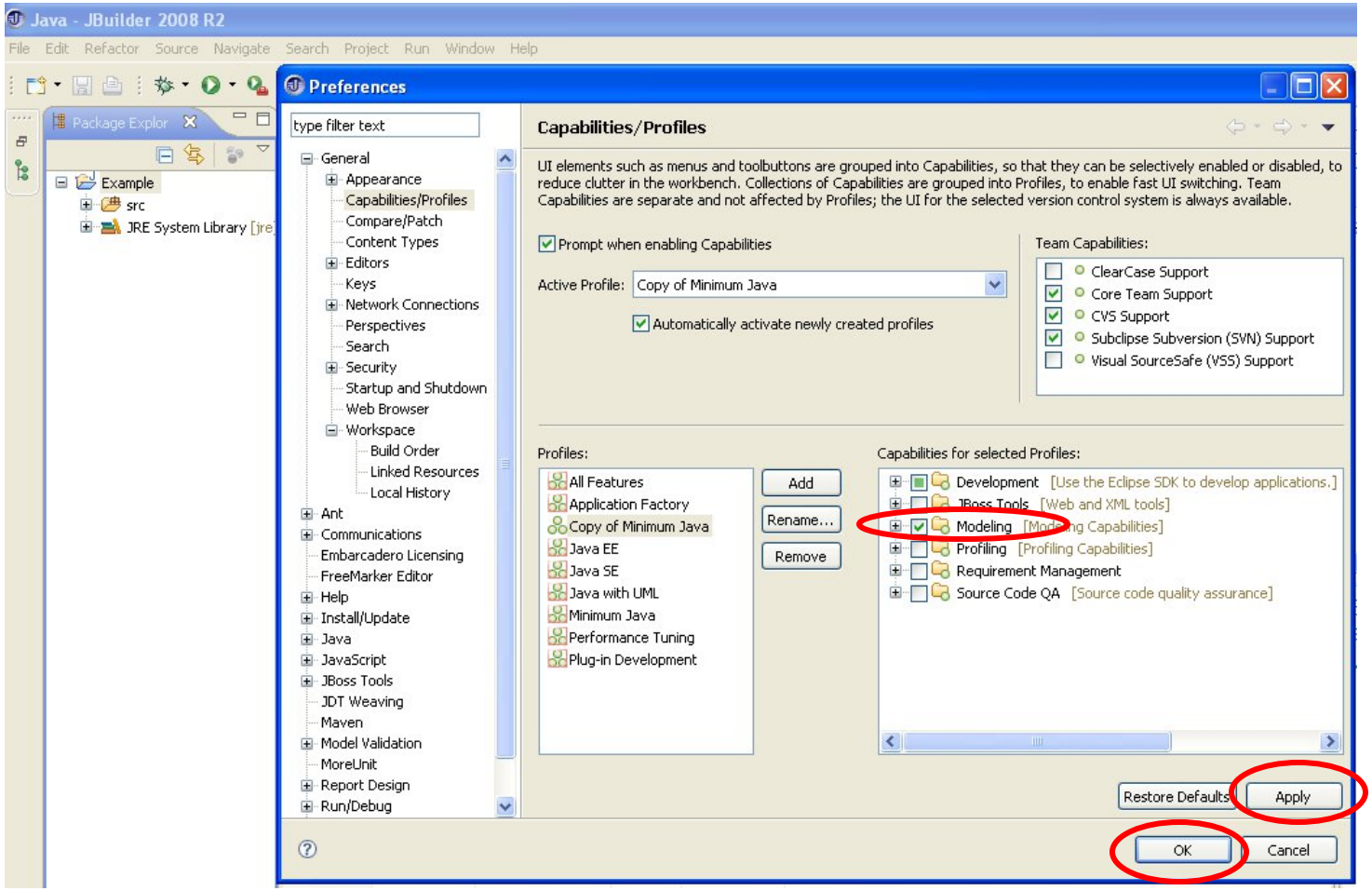
Finish up and restart JBuilder



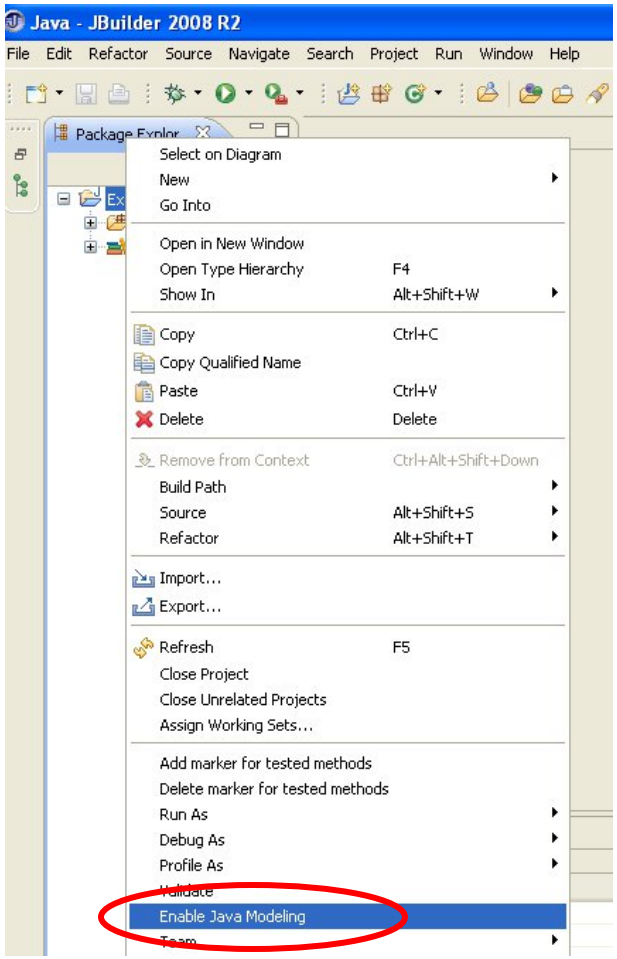
Create a new Java project (the same as using Eclipse)



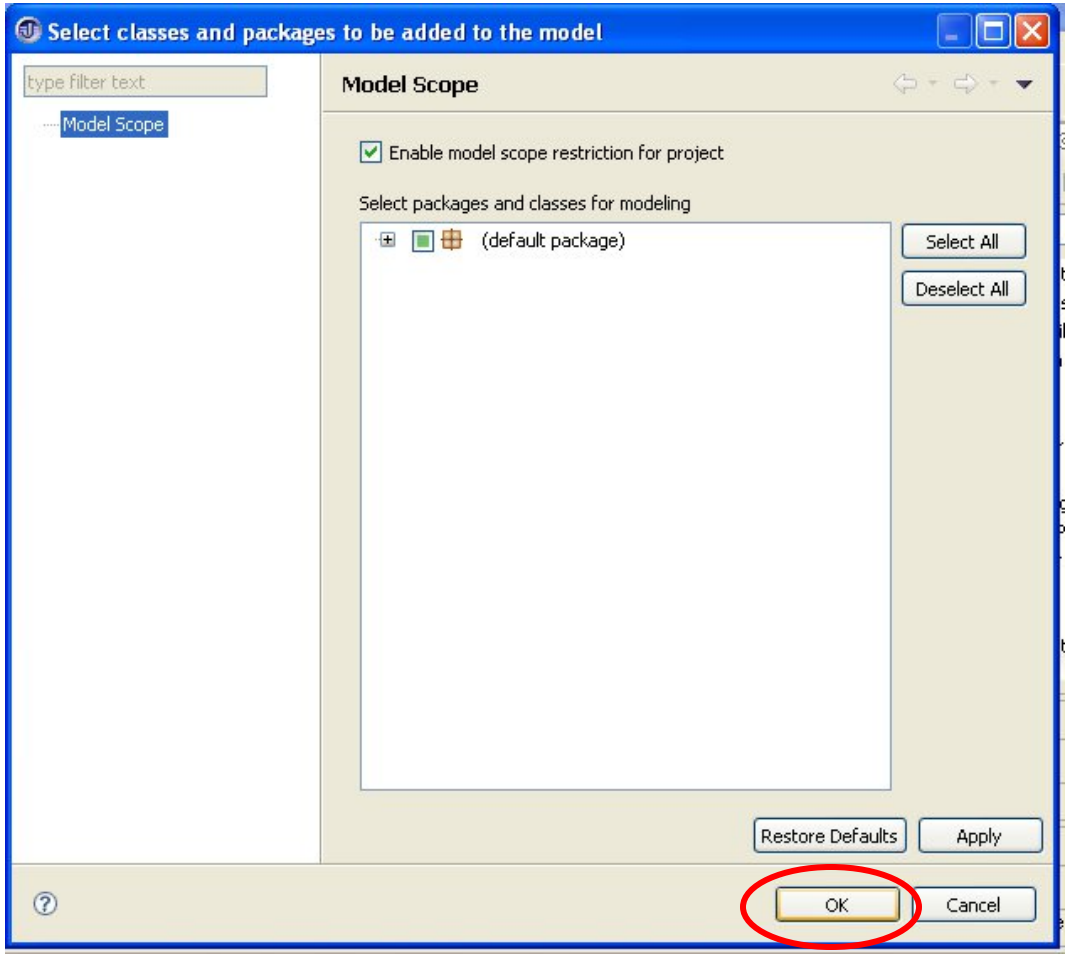
Make sure modeling is enabled



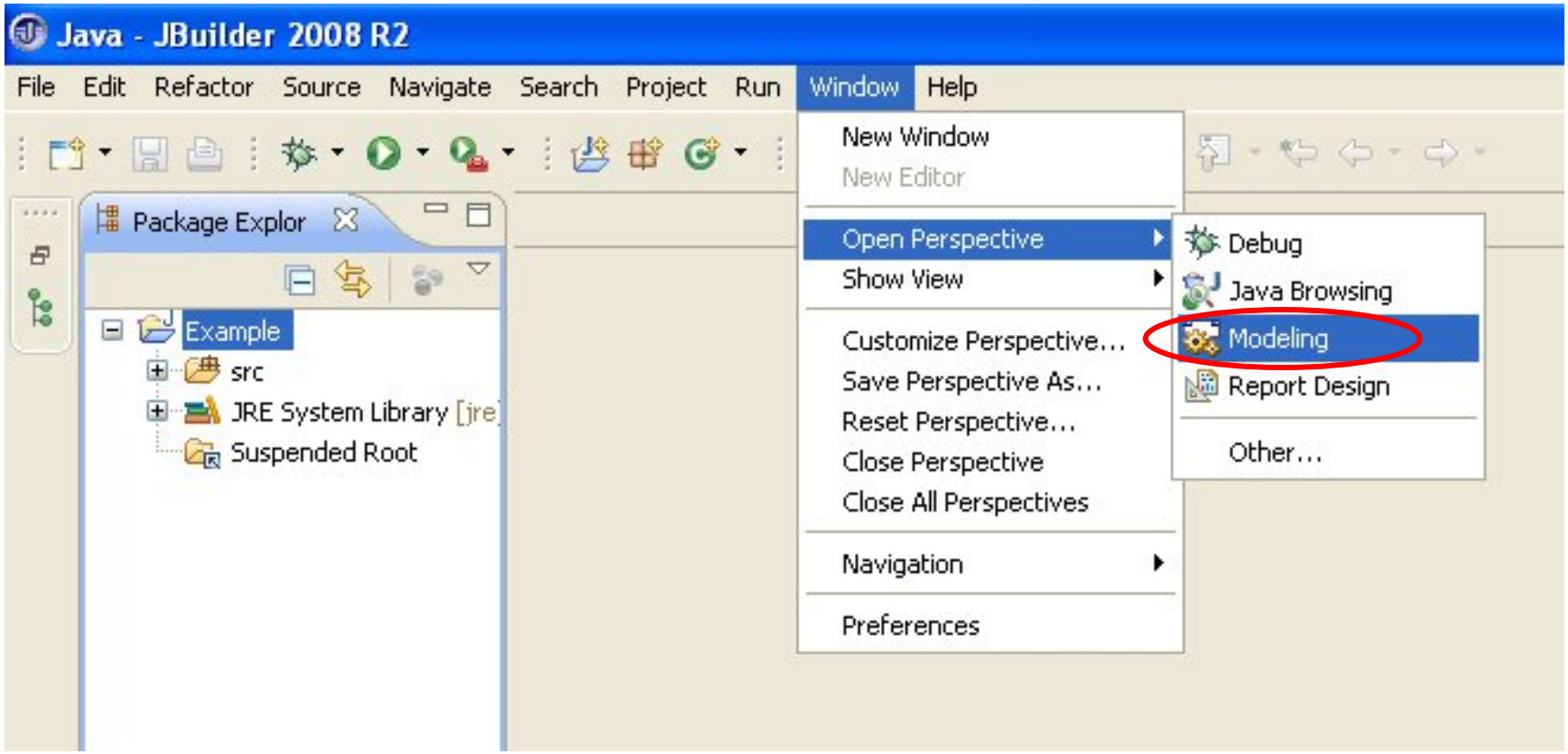
Right click on your project and “Enable Java Modeling”



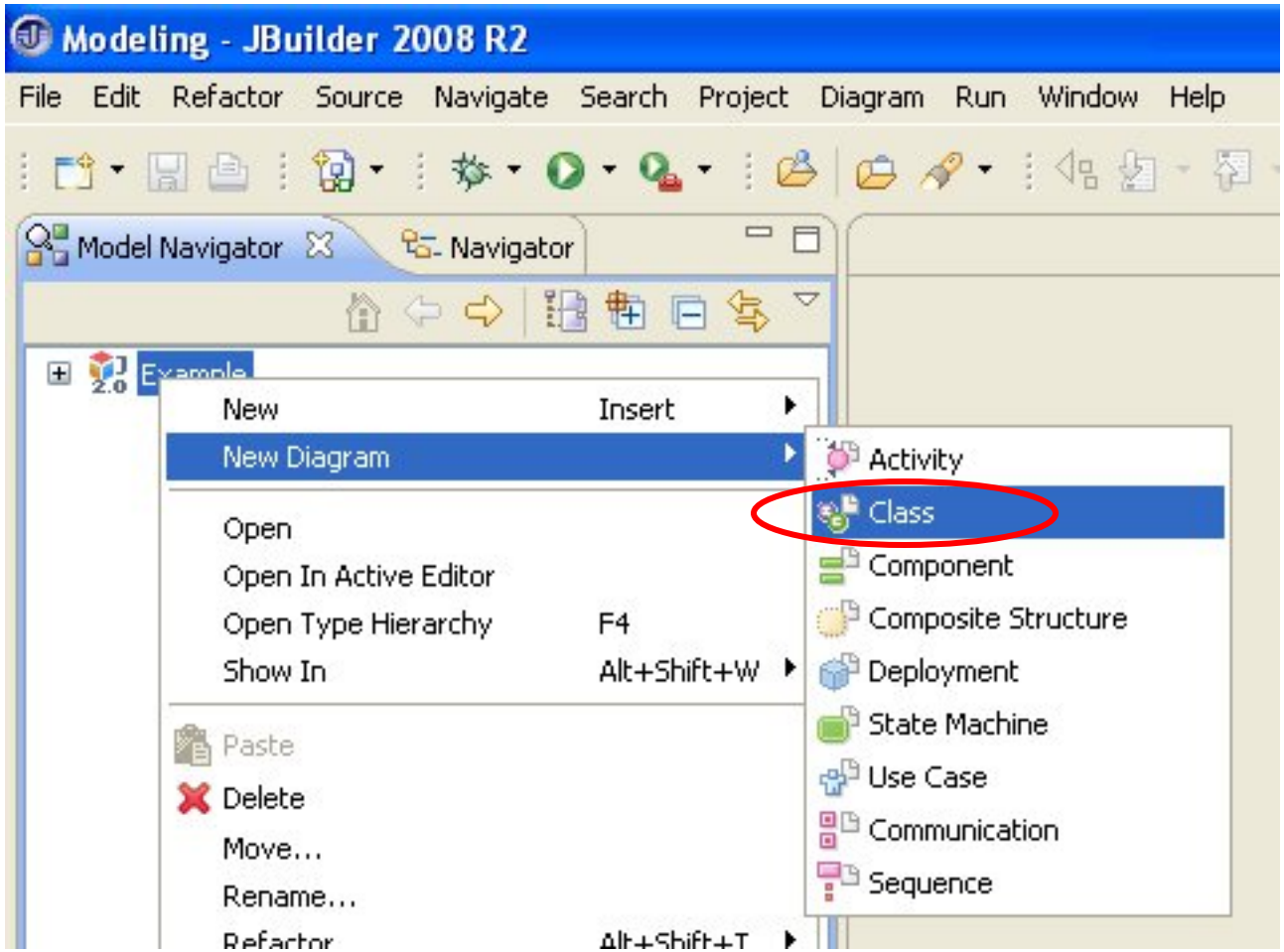
This window pops up (just hit ok)



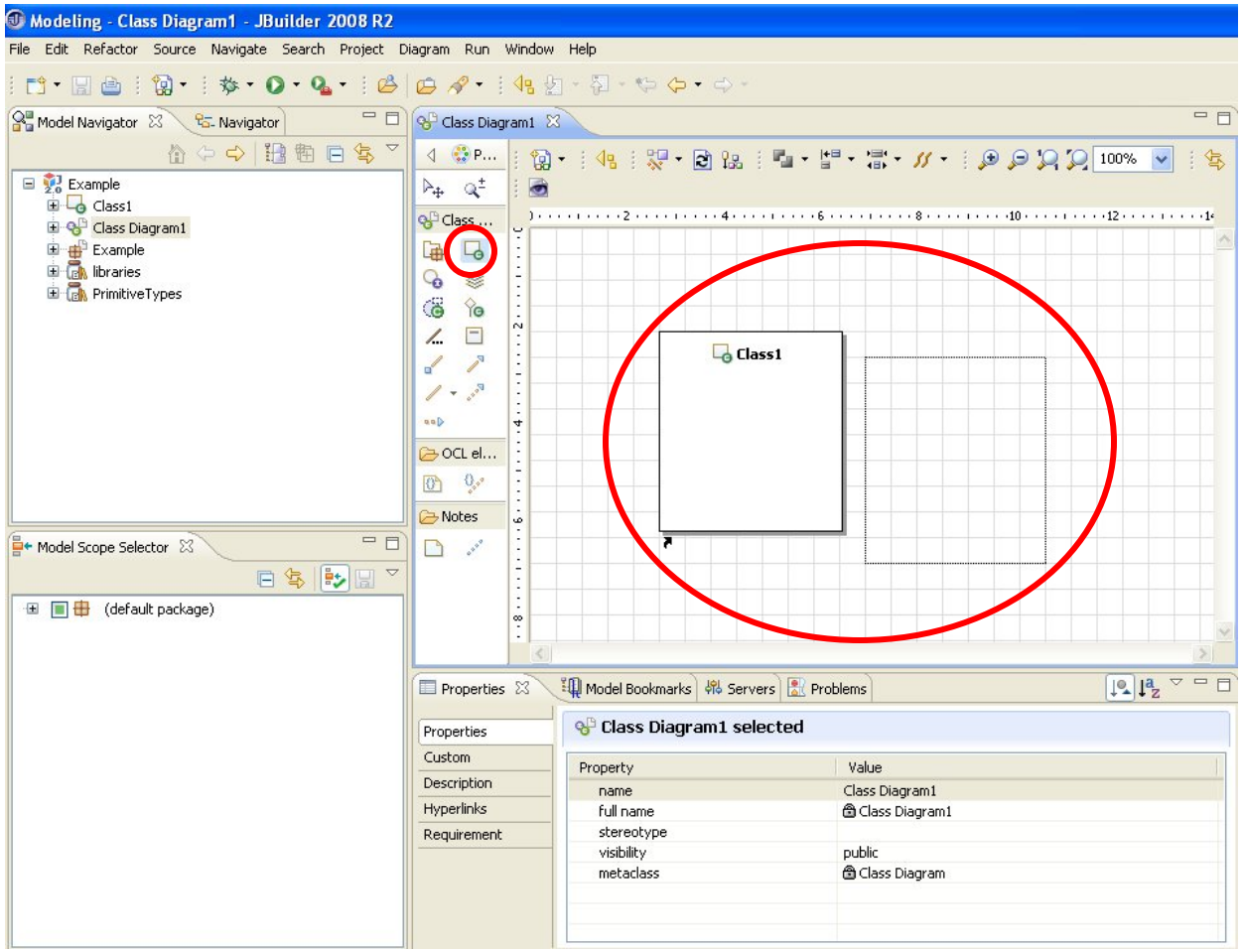
Open the Modeling Perspective



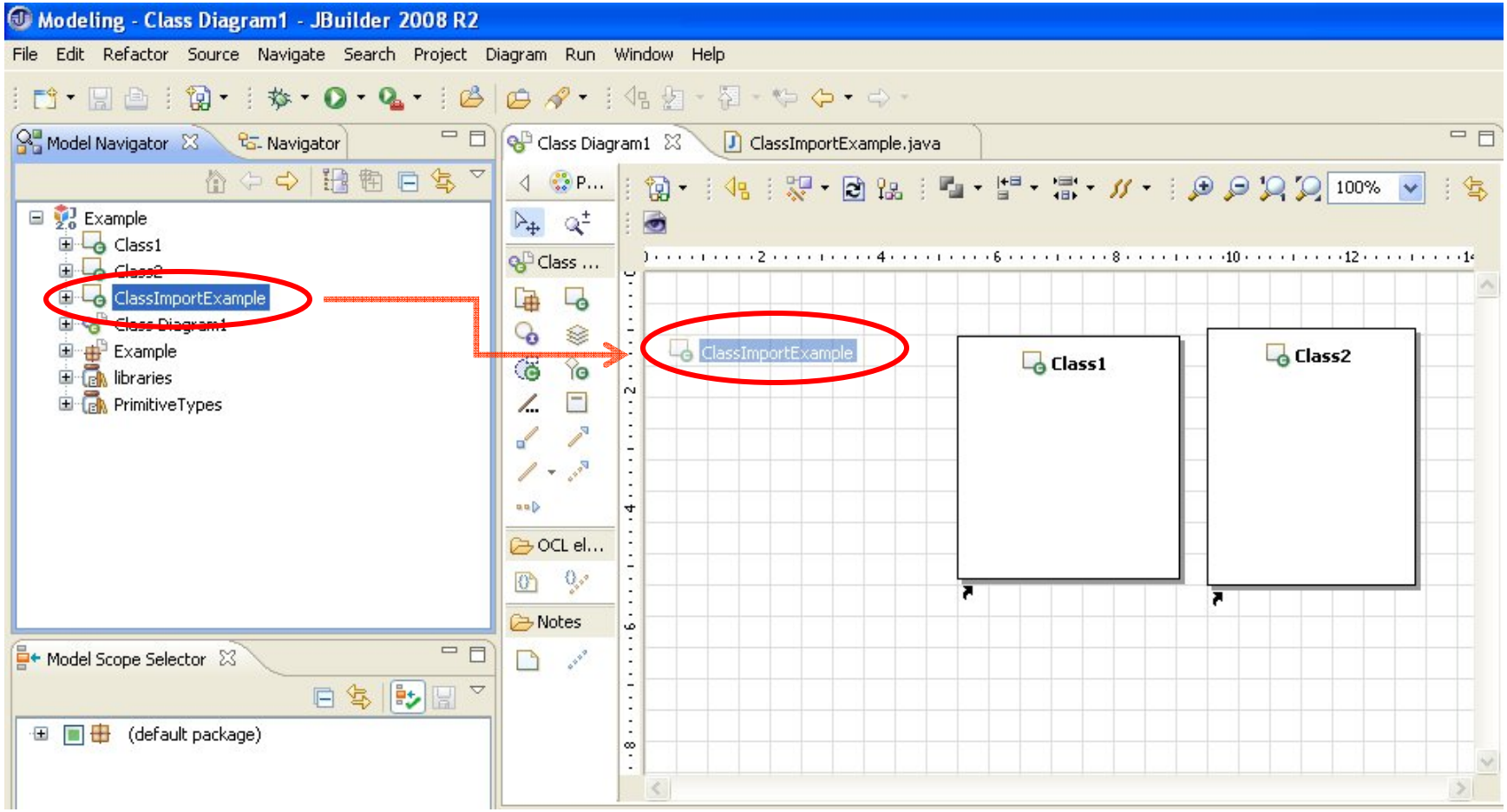
Create a New Class Diagram



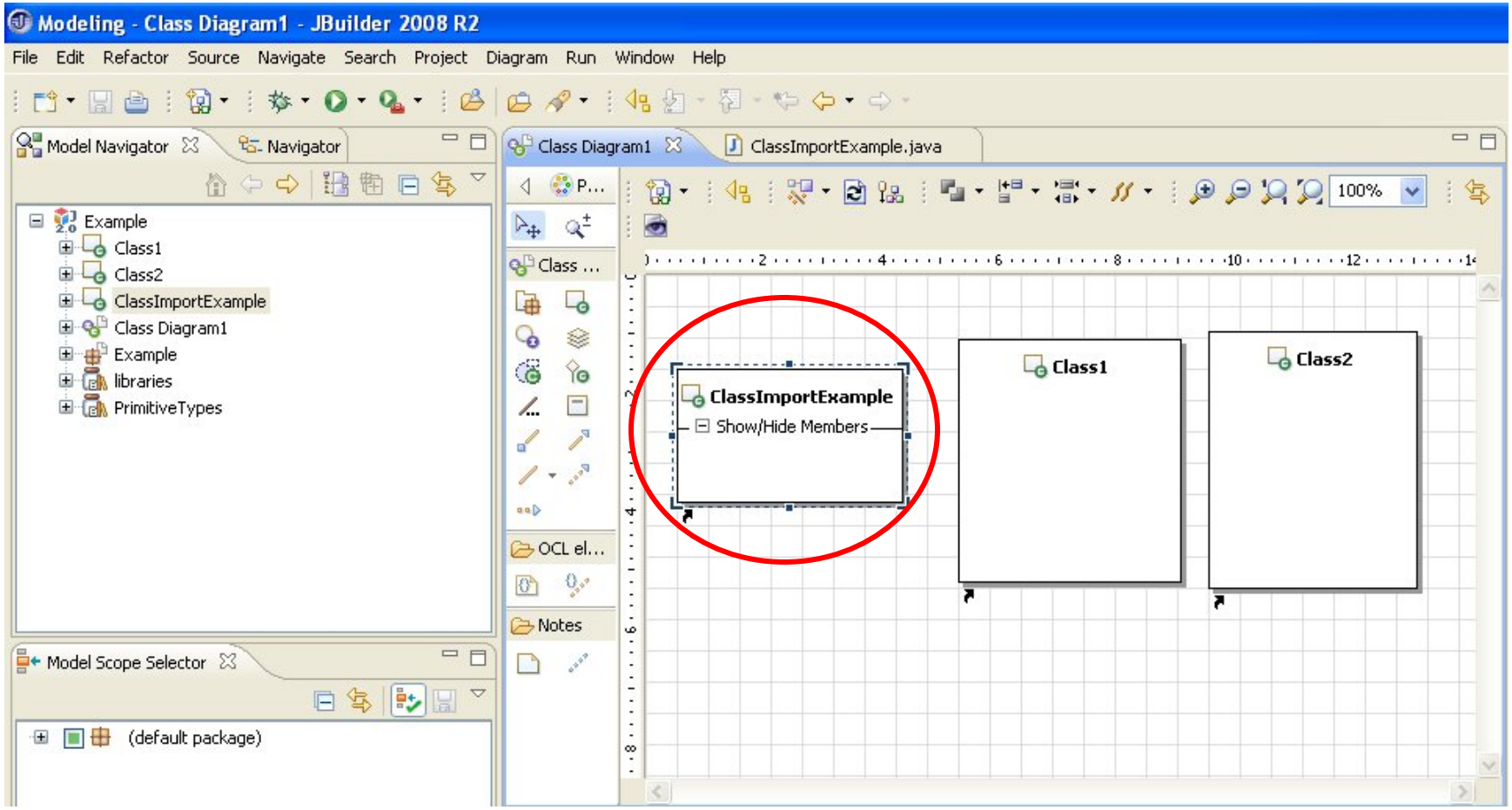
Use the class tool to create new classes



Drag existing classes onto the Class Diagram to add them



Tada!


Modeling - Class Diagram1 - JBuilder 2008 R2
File Edit Refactor Source Navigate Search Project Diagram Run Window Help
Model Navigator
Navigator
Example
Class1
Class2
ClassImportExample
Class Diagram1
Example
libraries
PrimitiveTypes
Model Scope Selector
(default package)
Class Diagram1
ClassImportExample.java
100%
Class ...
OCL el...
Notes
ClassImportExample
Show/Hide Members
Class1
Class2