# Object Intro and Miscellaneous

Checkout *ObjectIntroAndMisc* project from SVN

# Help from Peers

- Having a peer help you with some strange bug or specific problem – Great Idea!
- Discussing your approach to a problem with a peer – still OK
- Letting a peer copy your code/Emailing code to a peer – NEVER OK
- Every person has a unique code style, it's easy to tell when two sets of code are too similar

Q1

```
/**
 * Has a static method for computing n!
 * (n factorial) and a main method that
 * computes n! for n up to Factorial.MAX.
 *
 * @author Mike Hewner & Delvin Defoe
 */
public class Factorial {
    /**
     * Biggest factorial to compute.
     */
    public static final int MAX = 17;

    /**
     * Computes n! for the given n.
     *
     * @param n
     * @return n! for the given n.
     */
    public static int factorial (int n) {
        ...
    }

    ...

}
```

# Javadoc comments

We left out something important on the previous slide – comments!

Java provides Javadoc comments (they begin with /**) for both:

- Internal documentation for when someone reads the code itself
- External documentation for when someone re-uses the code

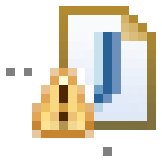
Comment your own code now, as indicated by this example. Don't forget the @author tag in HelloPrinter.

# Writing Javadocs

- Written in special comments: /** … */
- Can come before:
  - Class declarations
  - Field declarations
  - Constructor declarations
  - Method declarations
- Eclipse is your friend!
  - It will generate Javadoc comments automatically
  - It will notice when you start typing a Javadoc comment

# In all your code:

- See http://www.rose-hulman.edu/class/csse/csse220/201710/Homework/programGrading.html
- Write appropriate comments:
  - Javadoc comments primarily for classes.
  - Explanations of anything else that is not obvious in any spot.
- Give self-documenting variable and method names:
  - Use name completion in Eclipse, Ctrl-Space, to keep typing cost low and readability high
- Use Ctrl-Shift-F in Eclipse to format your code.
- Take care of all auto-generated TODO's.
  - **Then delete the TODO comment**.
- Correct ALL compiler warnings. Quick Fix is your friend!



# Debugging—Key Concepts

- Breakpoint
- Single stepping
- Inspecting variables

Q4

# Debugging—Demo

- Debugging Java programs in Eclipse:
  - Launch using the debugger
  - Setting a breakpoint
  - Single stepping: *step over* and *step into*
  - Inspecting variables
- Complete **WhackABug** exercise

# Primitive types

| Primitive Type | What It Stores | Range |
|---|---|---|
| `byte` | 8-bit integer | –128 to 127 |
| `short` | 16-bit integer | –32,768 to 32,767 |
| `int` | 32-bit integer | –2,147,483,648 to 2,147,483,647 |
| `long` | 64-bit integer | $-2^{63}$ to $2^{63} - 1$ |
| `float` | 32-bit floating-point | 6 significant digits ( $10^{-46}$, $10^{38}$ ) |
| `double` | 64-bit floating-point | 15 significant digits ( $10^{-324}$, $10^{308}$ ) |
| `char` | Unicode character | |
| `boolean` | Boolean variable | `false` and `true` |

**figure 1.2**

The eight primitive types in Java

Most common number types in Java code

# Gotcha!!!

- int vs. double:
  - int num1 = 1
  - double result = num1 / 2;
  - //what is result??

- How do we fix this?

# Exercise

- Work on SomeTypes.java

# Object Constructors

# Object Constructors

- int num = 5;
  - This works for primitive typed data

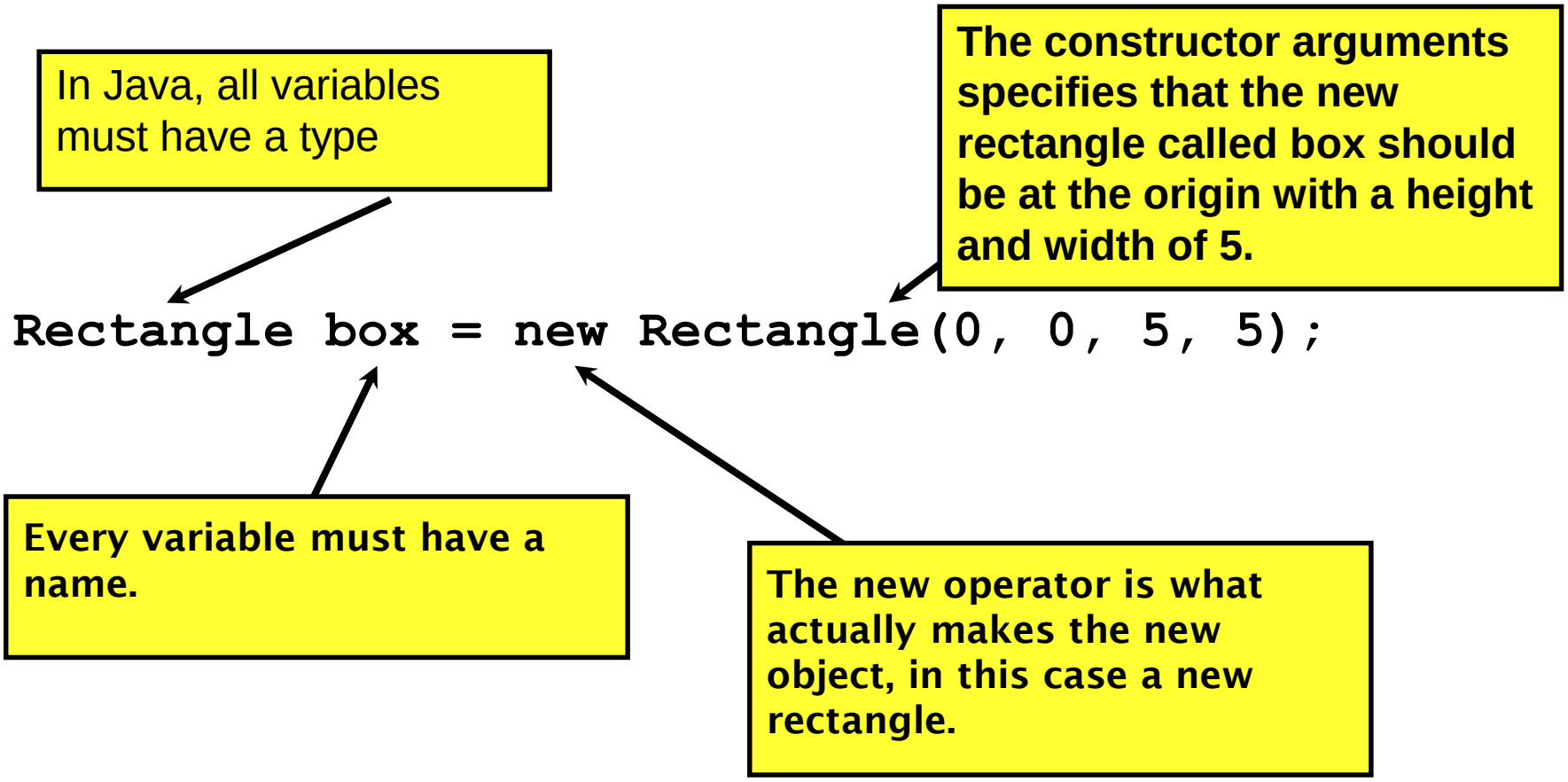
- What about “objects” (made from classes)?

# Object Constructors

- int num =  5;

- Rectangle  box   = new Rectangle(0, 0, 5, 5);

# Using Constructors




Q6

# Object Constructors

- Every “object” must be created
  - How do we create them?
- Open ObjectConstructorPractice.java
  - Let’s do the first couple of TODOs together
- On your own: Try creating a variable of the String class using a constructor (in the main method somewhere).

# Unit Testing

- Idea: Test “small pieces” of larger program
  - Do the expected values match what you ACTUALLY get?
- How to test in this manner?
  - Could make a main method that calls all the methods
  - JUnit!
    - Creating a Tester JUnit class

Q10

# Unit Tests (from the book)

1. Construct one or more objects of the class that is being tested
2. Invoke one or more methods
3. Print out one or more results
4. Print the expected results
5. Do 3 and 4 match?

*(Pages 102-103 in book)*

# What are good unit tests?

- Unit tests should be small pieces that test:
  1. The most common cases
  2. The edge cases (also when switching from positive to negative, etc.)
  3. All specific/special cases (e.g., when 0, the behavior is different than for any other value)
  4. When you find and fix a bug, you should have a unit test for this so it doesn't ever happen again. Fix things once and for all!
  5. Any overly complex code that 1-4 above don't cover

Q11

# Unit Testing

- Use “assert” to make sure results match
- Let’s look at BadFrac.java and BadFracTest.java
  - Let’s make some unit tests and figure out why this project has been yielding some strange results