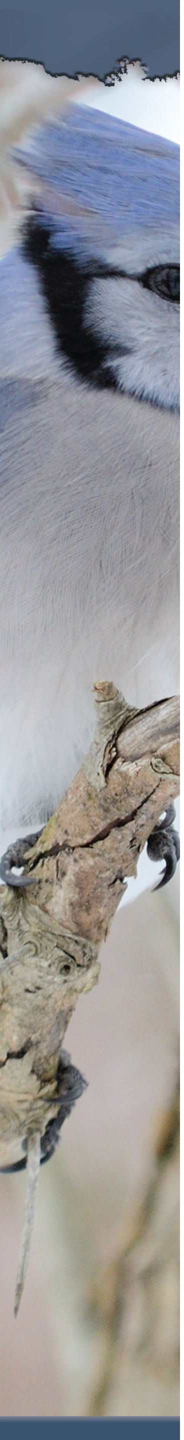
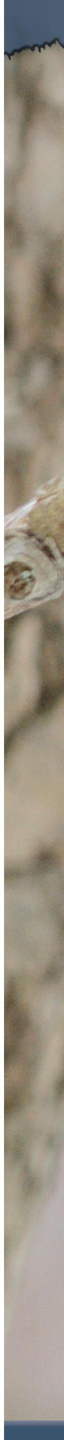
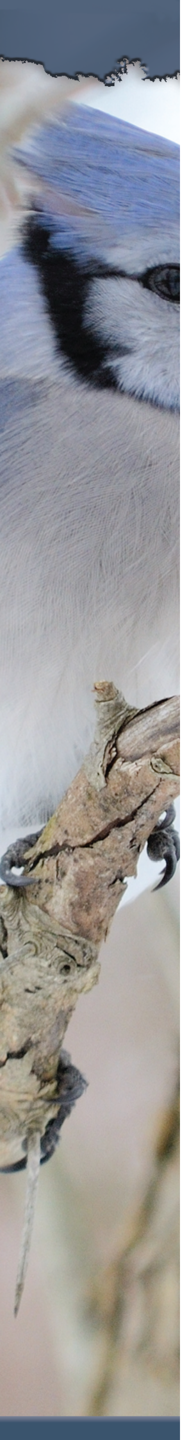


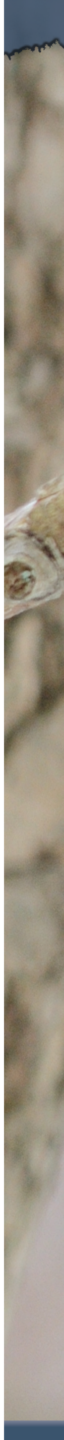
Well-behaved objects



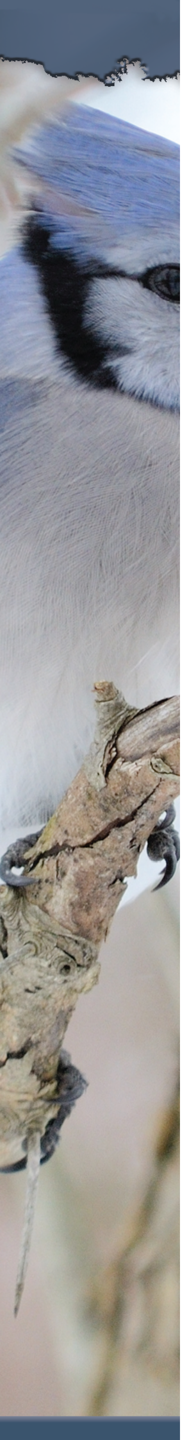


Main concepts to be covered

- Testing
- Debugging
- Test automation
- Writing for maintainability



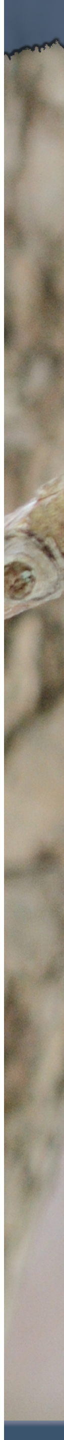
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.



Code snippet of the day

```
void test()
{
    int sum = 1;

    for (int i = 0; i <= 4; i++);
    {
        sum = sum + 1;
    }

    System.out.println("The result is: " + sum);
    System.out.println("Double result: " + sum+sum);
}
```

What is the output?



© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

Possible results

```
The result is: 5
The result is: 6
The result is: 11
The result is: 2

Double result: 12
Double result: 4
Double result: 22
Double result: 66
```

Which is printed?

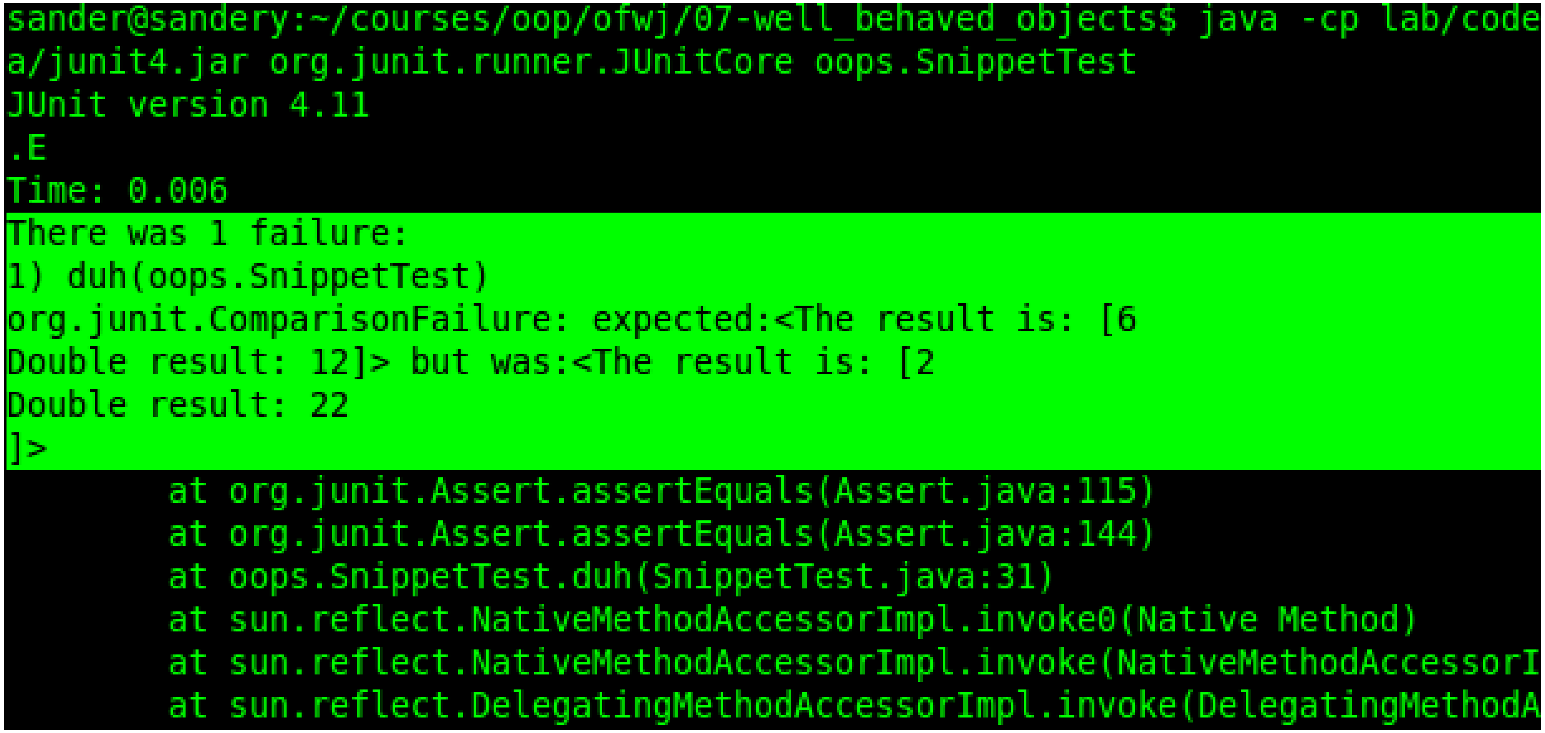
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

Possible results

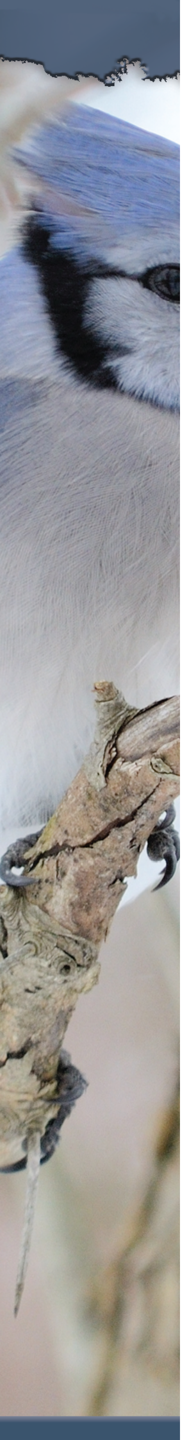
Expected results

```
The result is: 5
The result is: 6   ✔
The result is: 11
The result is: 2

Double result: 12   ✔
Double result: 4
Double result: 22
Double result: 66
```

(Im)Possible results



```
sander@sandery:~/courses/oop/ofwj/07-well_behaved_objects$ java -cp lab/code
a/junit4.jar org.junit.runner.JUnitCore oops.SnippetTest
JUnit version 4.11
.E
Time: 0.006
There was 1 failure:
1) duh(oops.SnippetTest)
org.junit.ComparisonFailure: expected:<The result is: [6
Double result: 12]> but was:<The result is: [2
Double result: 22
]>
        at org.junit.Assert.assertEquals(Assert.java:115)
        at org.junit.Assert.assertEquals(Assert.java:144)
        at oops.SnippetTest.duh(SnippetTest.java:31)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorI
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodA
```



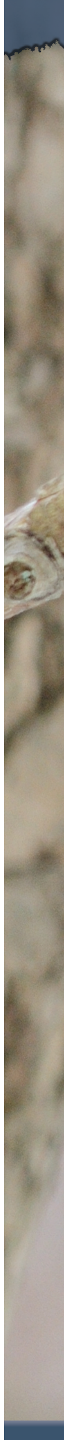
Possible results

```
The result is: 5
The result is: 6
The result is: 11
The

Doub
Doub
Double result: 22
Double result: 66
```

Which is printed?

```
The result is: 2
Double result: 22
```

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.



Code snippet of the day

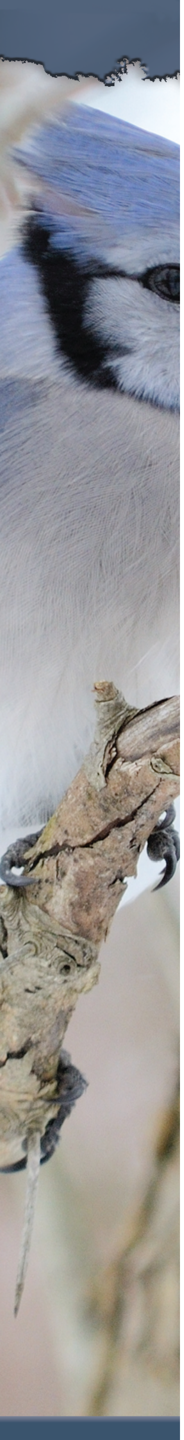
```
void test()
{
    int sum = 1;

    for (int i = 0; i <= 4; i++);
    {
        sum = sum + 1;
    }

    System.out.println("The result is: " + sum);
    System.out.println("Double result: " + sum+sum);
}
```

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

We have to deal with errors

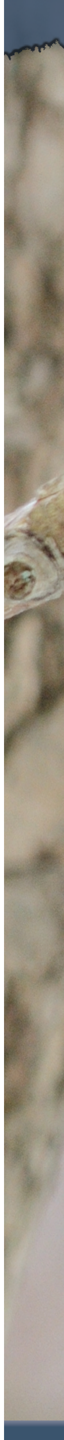
- Early errors are usually *syntax errors*.
  - The compiler will spot these.
- Later errors are usually *logic errors*.
  - The compiler cannot help with these.
  - Also known as bugs.
- Some logical errors have no immediately obvious manifestation.
  - Commercial software is rarely error free.

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

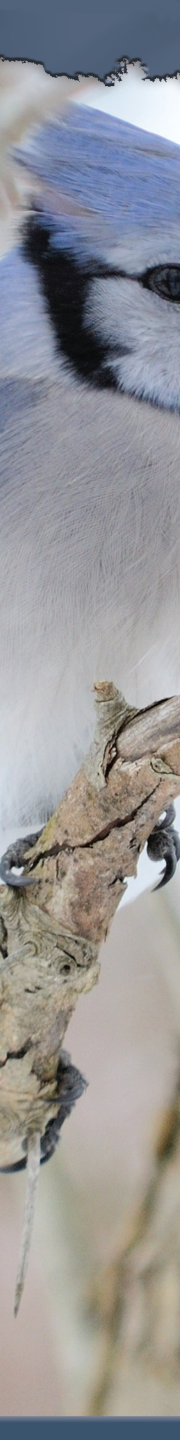


Prevention vs Detection
(Developer vs Maintainer)

- We can lessen the likelihood of errors:
  - Use software engineering techniques, like encapsulation.
  - Pay attention to cohesion and coupling.
- We can improve the chances of detection:
  - Use software engineering practices, like modularization and good documentation.
- We can develop detection skills.

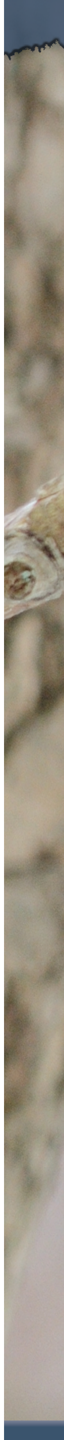


© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

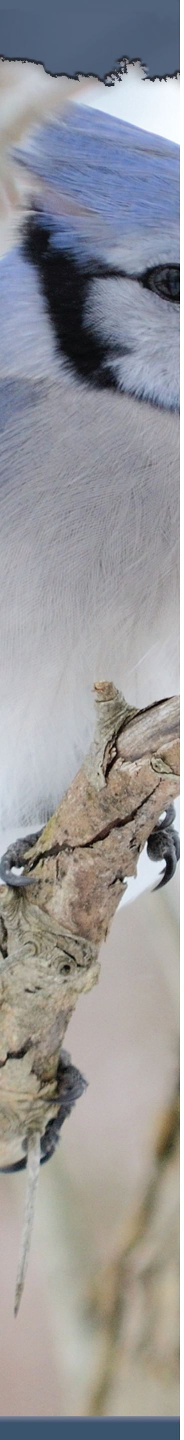


Testing and debugging

- These are crucial skills.
- Testing searches for the *presence* of errors.
- Debugging searches for the *source* of errors.
  - The manifestation of an error may well occur some ‘distance’ from its source.

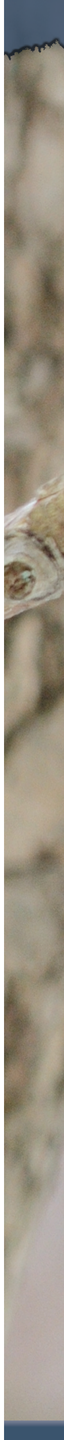


© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

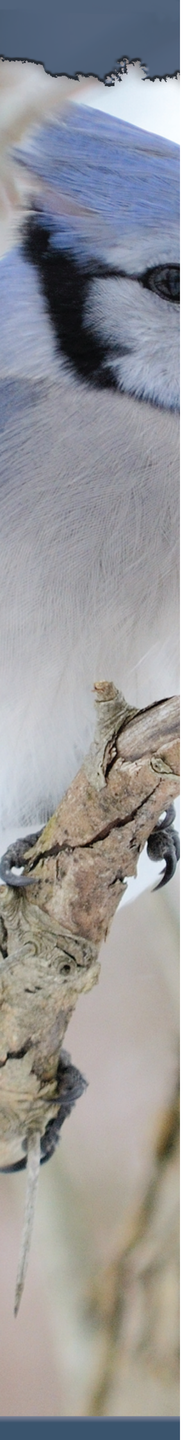


Testing and debugging techniques

- Unit testing
- Test automation
- Manual walkthroughs
- Print statements
- Debuggers

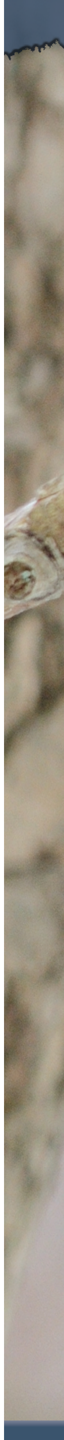


© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.



Unit testing

- Each unit of an application may be tested.
  - Method, class, module (package in Java).
- Can (should) be done during development.
  - Finding and fixing early lowers development costs (e.g. programmer time).
  - A test suite is built up.

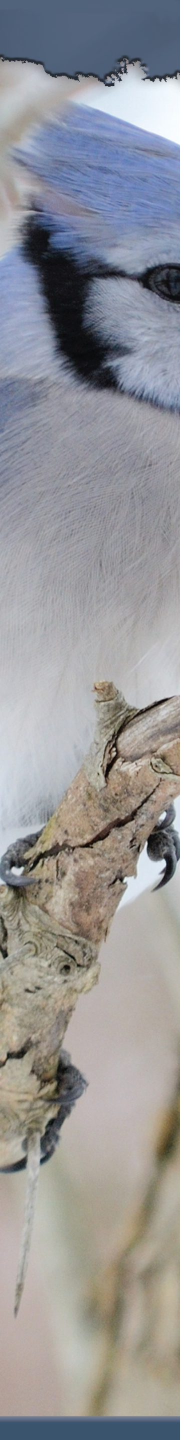


© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

Testing fundamentals

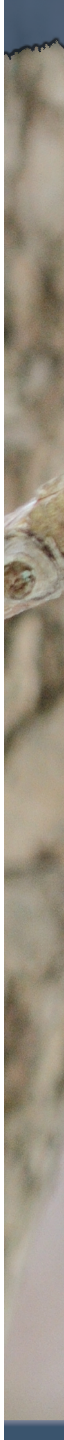
- Understand what the unit should do – its *contract*.
  - You will be looking for violations.
  - Use positive tests and negative tests.
- Test *boundaries*.
  - Zero, One, Full.
    - Search an empty collection.
    - Add to a full collection.
    - Search for/remove the only element.

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.



Well-behaved objects

Test automation



Main concepts to be covered

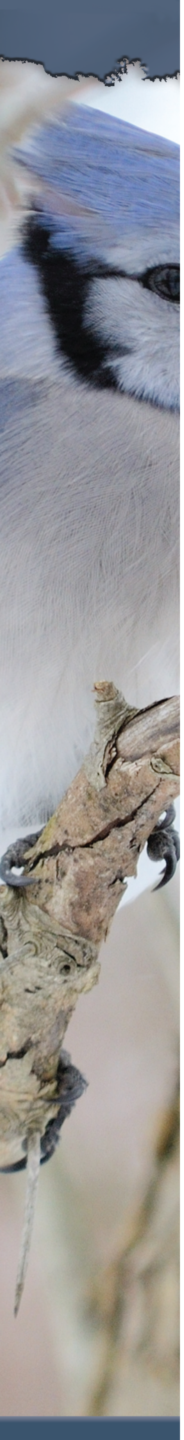
- Unit testing
- JUnit
- Regression testing
- Test cases
- Test classes
- Assertions
- Fixtures

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

Test automation

- Good testing is a creative process, but ...
- ... thorough testing is time consuming and repetitive.
- *Regression testing* involves re-running tests.
- Use of a *test rig* or *test harness* can relieve some of the burden.

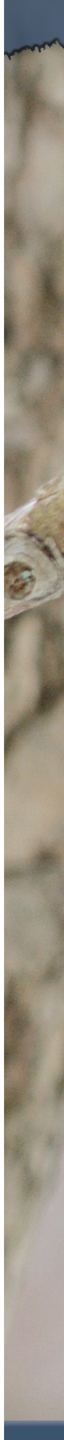
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

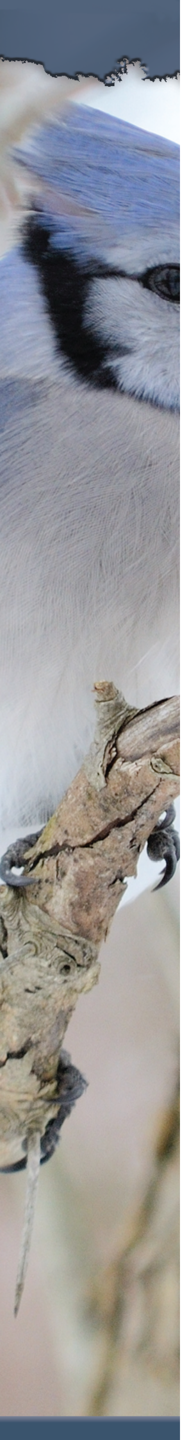


Test harness

- Additional test classes are written to automate the testing.
- Objects of the harness classes replace human interactivity.
- Creativity and imagination required to create these test classes.
- Test classes must be kept up to date as functionality is added.

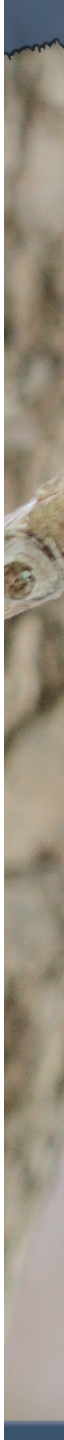
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.



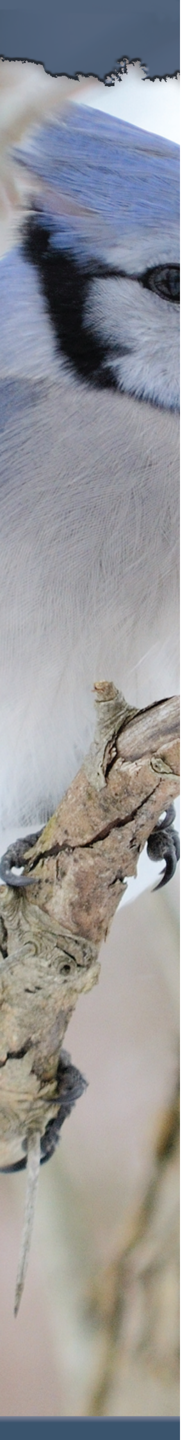


Test automation

- Test frameworks exist to support automation.
- Explore fuller automation through the *online-shop-junit* project.
  - Intervention only required if a failure is reported.

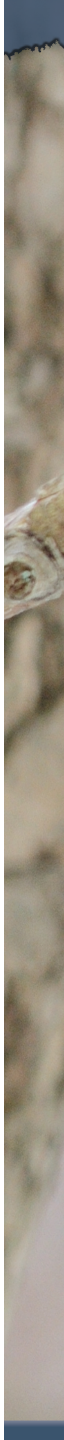


© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

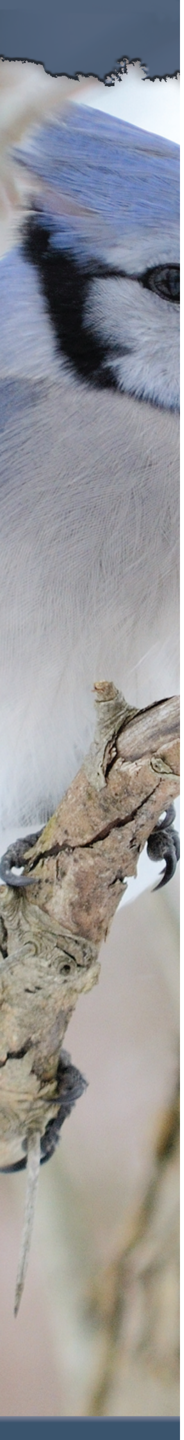


JUnit

- JUnit is a Java test framework
- Test cases are methods that contain tests
- Test classes contain test methods
- Assertions are used to assert expected method results
- Fixtures are used to support multiple tests

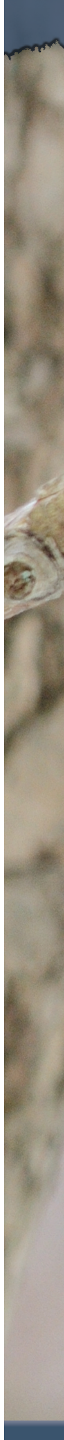


© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.



Well-behaved objects

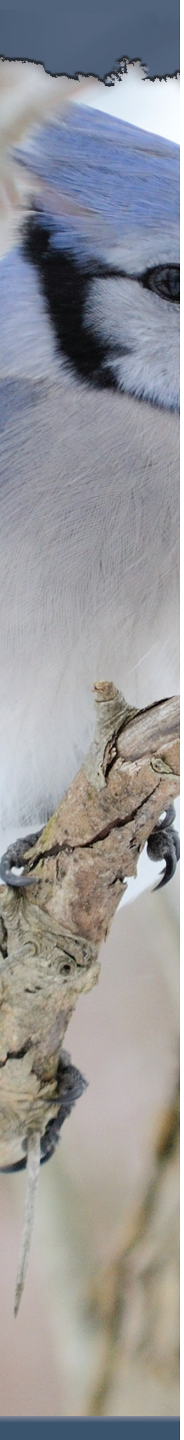
Debugging



Prevention vs Detection (reprise)

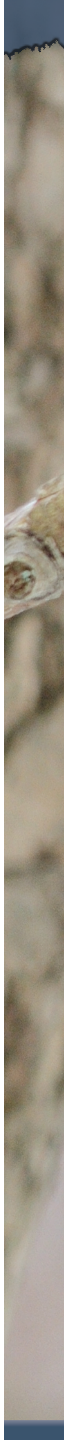
- We can lessen the likelihood of errors:
  - Use software engineering techniques, like encapsulation.
  - Pay attention to cohesion and coupling.
- We can improve the chances of detection:
  - Use software engineering practices, like modularization and good documentation.
- We can develop detection skills.

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

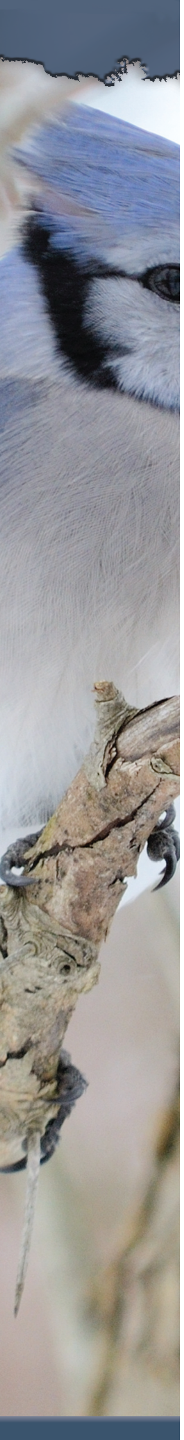


Debugging techniques

- Manual walkthroughs
- Print statements
- Debuggers

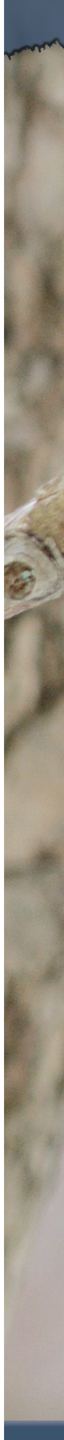


© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.



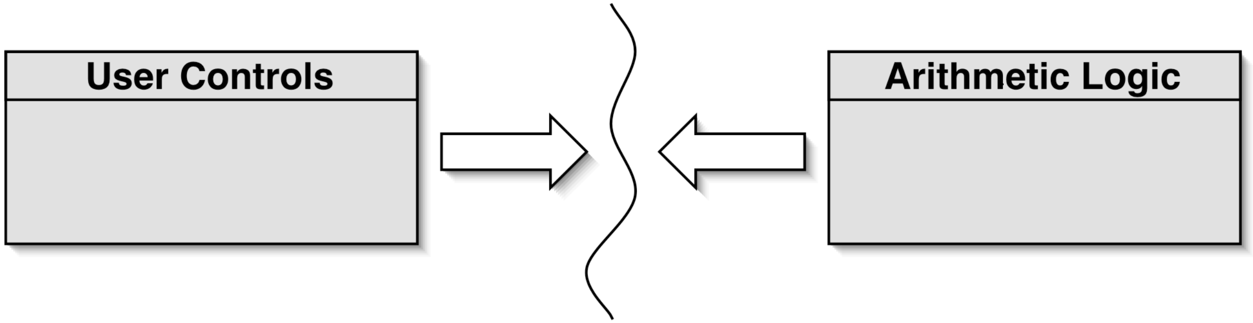
Modularization and interfaces

- Applications often consist of different modules:
  - E.g. so that different teams can work on them.
- The *interface* between modules must be clearly specified.
  - Supports independent concurrent development.
  - Increases the likelihood of successful integration.



© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

Modularization in a calculator

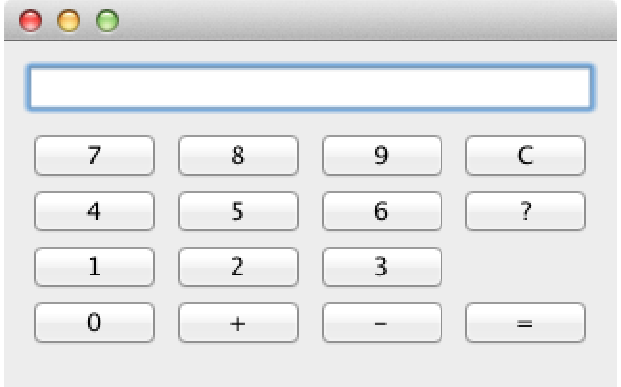


- Each module does not need to know implementation details of the other.
  - User controls could be a GUI or a hardware device.
  - Logic could be hardware or software.

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

Method headers as an interface

```
// Return the value to be displayed.
int getDisplayValue();

// Call when a digit button is pressed.
void numberPressed(int number);

// Plus operator is pressed.
void plus();

// Minus operator is pressed.
void minus();

// Call to complete a calculation.
void equals();

// Call to reset the calculator.
void clear();
```

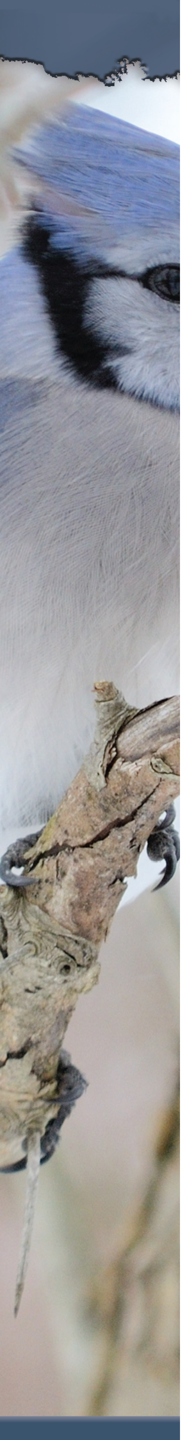


© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

Debugging

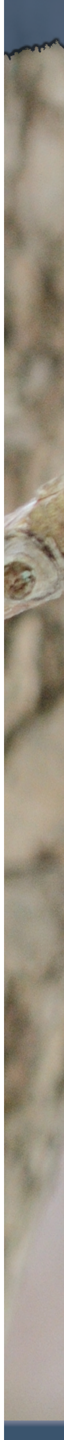
- It is important to develop code-reading skills.
  - Debugging will often be performed on others' code.
- Techniques and tools exist to support the debugging process.
- Explore through the *calculator-engine* project.

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.



Manual walkthroughs

- Relatively underused.
  - A low-tech approach.
  - More powerful than appreciated.
- Get away from the computer!
- ‘Run’ a program by hand.
- High-level (Step) or low-level (Step into) views.

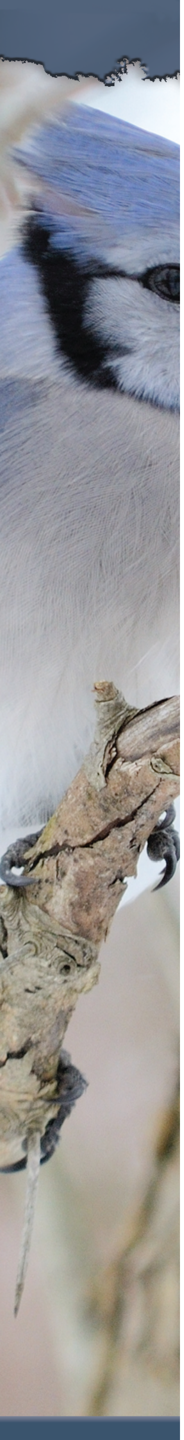


© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

Tabulating object state

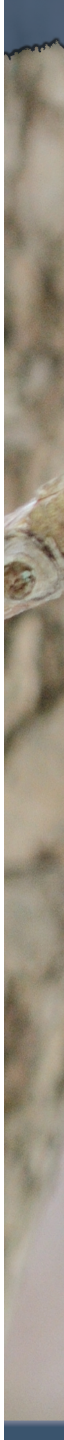
- An object’s behavior is largely determined by its state …
- … so incorrect behavior is often the result of incorrect state.
- Tabulate the values of key fields.
- Document state changes after each method call.

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.



Verbal walkthroughs

- Explain to someone else what the code is doing:
  - *They* might spot the error.
  - *You* might spot the error, through the process of explaining.
- Group-based processes exist for conducting formal walkthroughs or *inspections*.

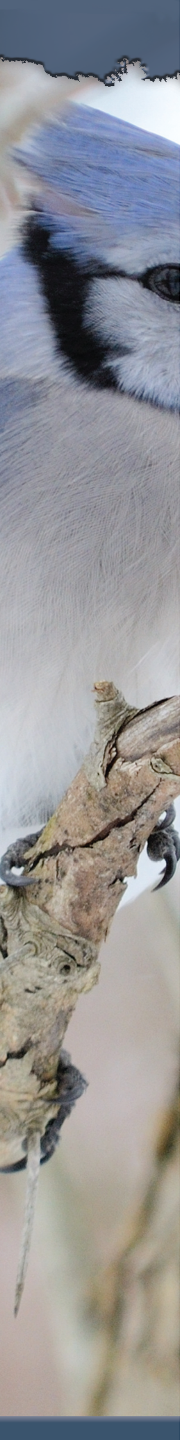


© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

Print statements

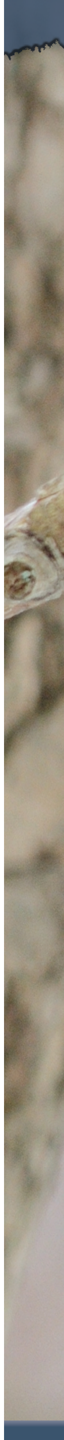
- The most popular technique.
- No special tools required.
- All programming languages support them.
- Only effective if the right methods are documented.
- Output may be voluminous!
- Turning off and on requires forethought.

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.



Choosing a test strategy

- Be aware of the available strategies.
- Choose strategies appropriate to the point of development.
- Automate whenever possible.
  - Reduces tedium.
  - Reduces human error.
  - Makes (re)testing more likely.

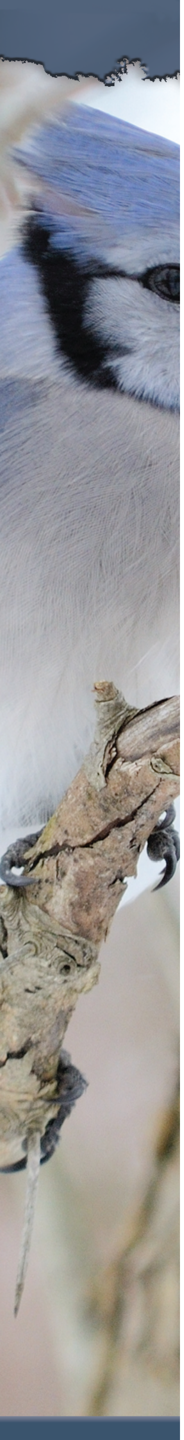


© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

Debuggers

- Debuggers are both language- and environment-specific.
    - Eclipse has an integrated debugger.
- Support breakpoints.
- *Step* and *Step-into* controlled execution.
- Call sequence (stack).
- Object state.

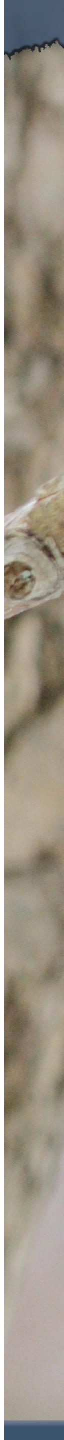
© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.



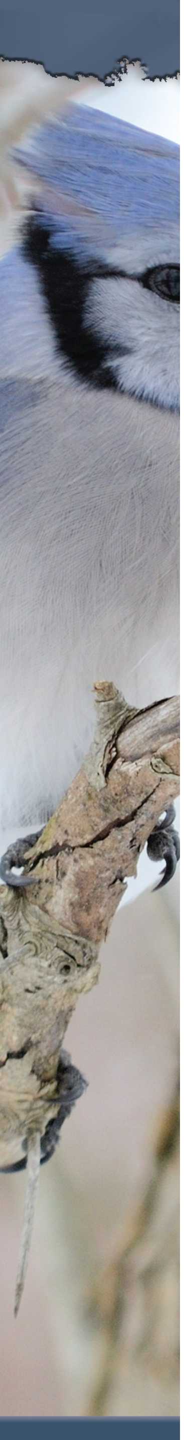
Debugging streams (advanced)

- A pipeline of multiple operations might be hard to debug.
- The **`peek`** operation can provide insights.
- Consumer that passes on its input unchanged; e.g.:

```
peek(s -> System.out.println(s))
```

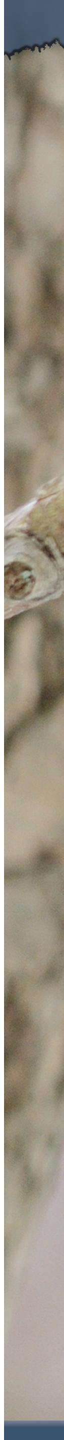


© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.



Review

- Errors are a fact of life in programs.
- Good software development techniques can reduce their occurrence.
- Testing and debugging skills are essential.
- Make testing a habit.
- Automate testing where possible.
- Continually repeat tests.
- Practice a range of debugging skills.



© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.