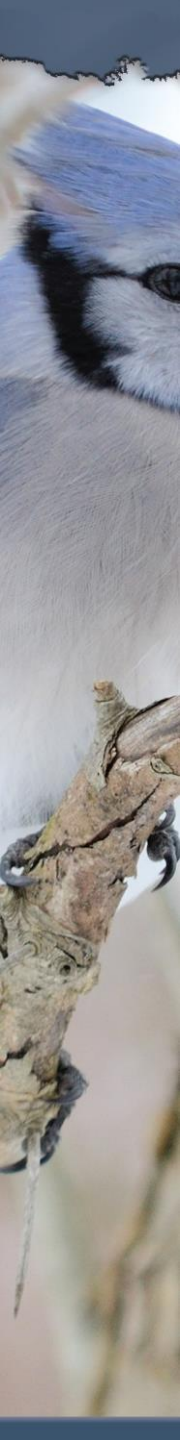
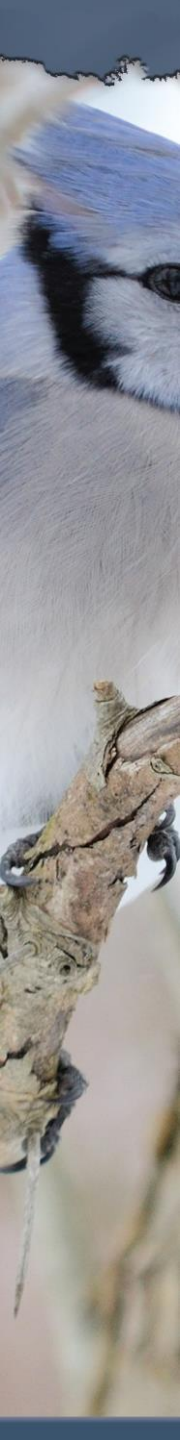


Well-behaved objects



Code snippet of the day

```
public 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?

Results

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

Double result: 10
Double result: 12
Double result: 22
Double result: 66
```

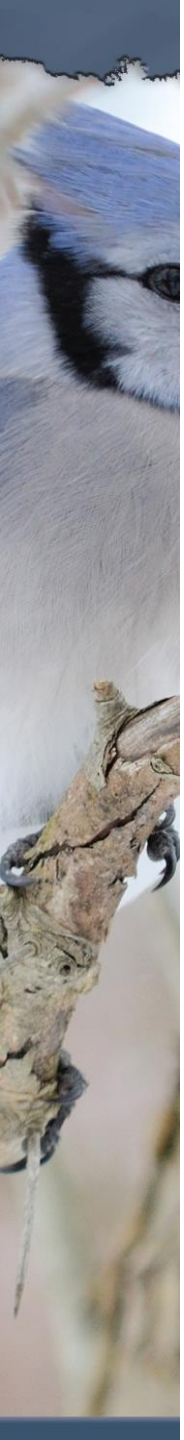
Which ones are printed?

Results

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

Code snippet of the day

```
public 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);
}
```



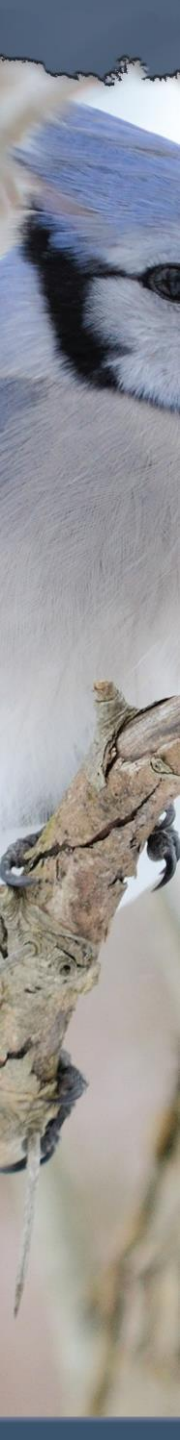
We have to deal with errors

- 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.

What is the output?

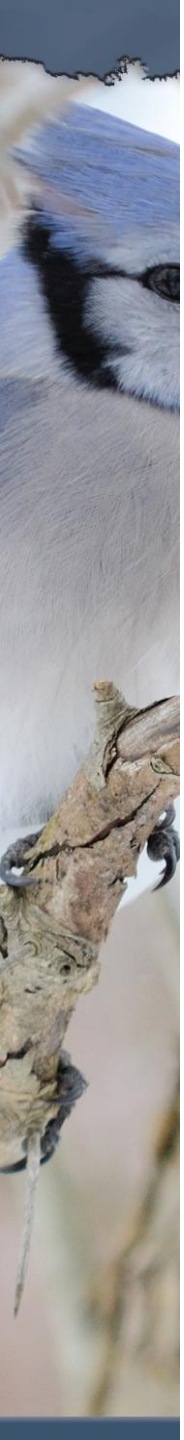
```
public void doSomething() {
    int a;
    for (int i = 0; i < 10; i++) {
        a += i;
    }
    System.out.println(a);
}
```

Compile error!



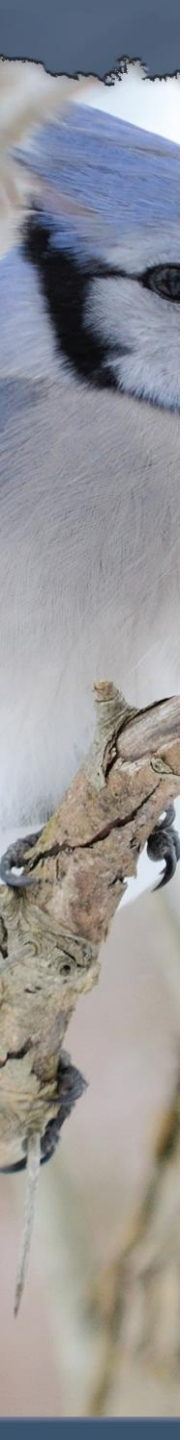
Java Default Values

- Fields that are declared but not initialized will be set to a reasonable default by the compiler.
  - The default will be zero or null, depending on the data type.
- Local variables are slightly different; the compiler never assigns a default value to an uninitialized local variable.
  - Accessing an uninitialized local variable will result in a compile-time error.
- A floating-point literal is of type float if it ends with the letter F or f.
- Java automatically initializes the elements of arrays to the default values.
  - Java does not initialize non-array local variables



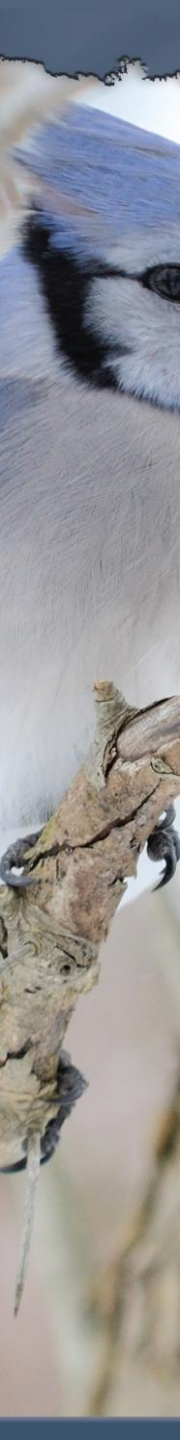
Prevention vs Detection
(Developer vs Maintainer)

- We can lessen the likelihood of errors.
  - Using software engineering techniques, like encapsulation \ information hiding.
- We can improve the chances of detection.
  - Using software engineering practices, like modularization and documentation.
- We can develop detection skills.



Testing and Debugging

- Crucial skills in software engineering.
- Testing is the activity of finding-out whether a piece of code produces the intended behavior.
- Debugging is the attempt to pin-point and fix the source of an error.
  - The manifestation of an error may well occur some 'distance' from its source.



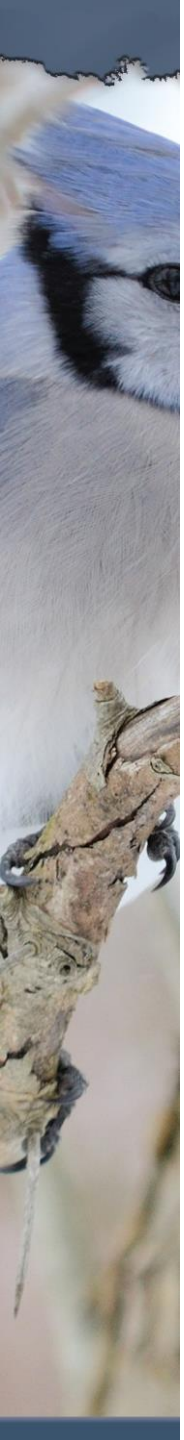
Testing

- Testing means running the program using a set of test-cases, and examining the results.
- A test-case is a pair of (inputs, outputs) that are designed to verify program functionality.
- Designing effective test-cases is the art of good software-testing.

Test Cases?

```
public char gradeRank(double studentGrade) {
    char grade;
    if (studentGrade >= 90) {
        grade = 'A';
    } else if (studentGrade >= 80) {
        grade = 'B';
    } else if (studentGrade >= 70) {
        grade = 'B';
    } else if (studentGrade >= 60) {
        grade = 'D';
    } else {
        grade = 'F';
    }
    return grade;
}
```

Input	Output
100	A
85	B
73	C
68	D
55	F

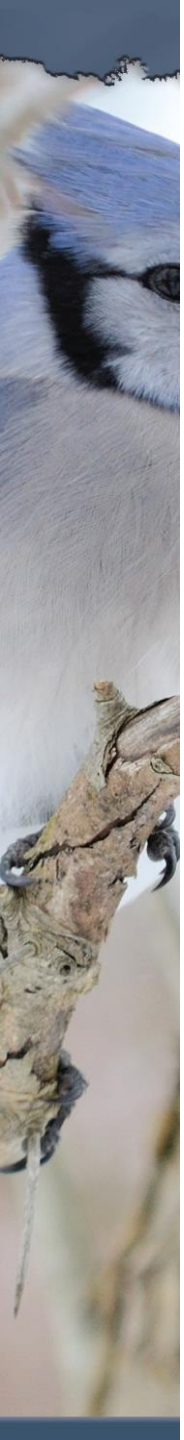


Testing Fundamentals

- How to design effective test-cases?
- Understand the “contract”:
  what the program should, and shouldn’t do.
- Test *boundaries*.
  - Zero, One, Full.
    - Search an empty collection.
    - Add to a full collection.
- Use positive tests and negative tests.

Boundary Value Testing

```
public char gradeRank(double studentGrade) {
    char grade;
    if (studentGrade >= 90) {
        grade = 'A';
    } else if (studentGrade >= 80) {
        grade = 'B';
    } else if (studentGrade >= 70) {
        grade = 'C';
    } else if (studentGrade >= 60) {
        grade = 'D';
    } else {
        grade = 'F';
    }
    return grade;
}
```



Positive vs Negative Testing

- Positive testing is the testing of cases that are expected to succeed.
- Negative testing is the testing of cases that are expected to fail.
- A common pitfall for inexperienced testers is to conduct only positive tests! Negative tests are crucial for a good test procedure.

Positive vs Negative Testing

```
public char gradeRank(double studentGrade) {
    char grade;
    if (studentGrade >= 90) {
        grade = 'A';
    } else if (studentGrade >= 80) {
        grade = 'B';
    } else if (studentGrade >= 70) {
        grade = 'C';
    } else if (studentGrade >= 60) {
        grade = 'D';
    } else {
        grade = 'F';
    }
    return grade;
}
```

Input	Positive vs Negative
55	Positive
85	Positive
-50	Negative
110	Negative

Positive vs Negative Testing

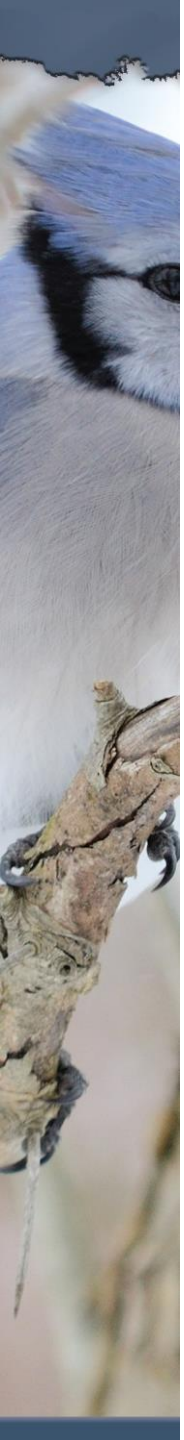
```
public char gradeRank(double studentGrade) {
    char grade;
    if (studentGrade > 100 || studentGrade < 0) {
        grade = '-';
    } else if (studentGrade >= 90) {
        grade = 'A';
    } else if (studentGrade >= 80) {
        grade = 'B';
    } else if (studentGrade >= 70) {
        grade = 'C';
    } else if (studentGrade >= 60) {
        grade = 'D';
    } else {
        grade = 'F';
    }
    return grade;
}
```

Input	Positive vs Negative
55	Positive
85	Positive
-50	Negative
110	Negative

Positive vs Negative Testing

```
public int convertAdd(String num) {
    int intValue = Integer.parseInt(num) + 1;
    return intValue;
}
```

Input	Positive vs Negative
5	Positive
158	Positive
+10	Positive
-8	Positive
--20	Negative
A	Negative
25.8	Negative



Assertions

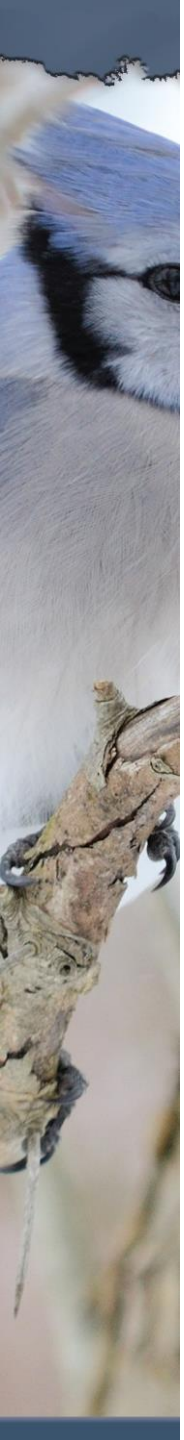
- An assertion is an expression that states a condition that we expect to be true.
- If the condition is false, we say that the assertion fails.
  - This indicates an error in our program.
- Some programming languages (like Java) have special statements for assertion.

Assertions

```
int sum = 1;
for (int i = 0; i <= 4; i++)
    sum = sum + 1;
assert (sum == 6);
```

```
assert list != null && list.size() > 0 : "list is null or empty";
```

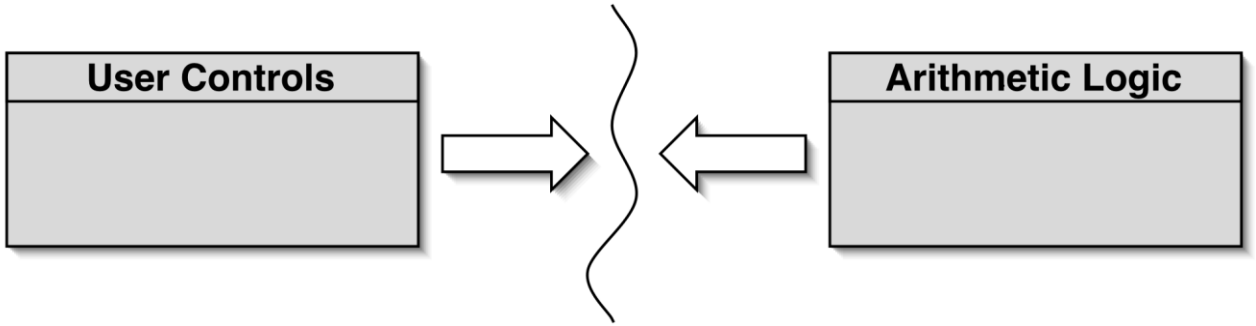
- If the assert condition is true, then nothing happens.
- If the assert condition is false, an exception is raised:
  - `java.lang.AssertionError`
  - NOTE: must be run using: `java -ea` to "enable assertions".



Modularization & 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.

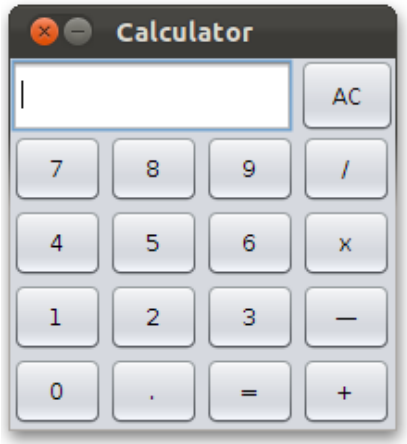
Modularization in a calculator

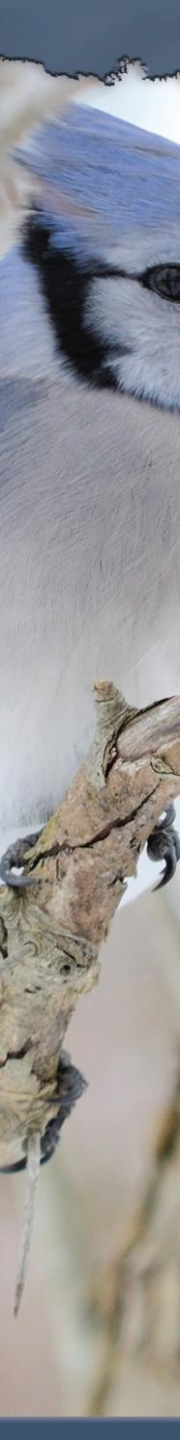


- Each module does not need to know implementation details of the other.
  - User controls could be a GUI or a hardware device.
  - Logic could be hardware or software.

Method signatures as Interfaces

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

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

// Call when a plus operator is pressed.
public void plus();

// Call when a minus operator is pressed.
public void minus();

// Call to complete a calculation.
public void calculate();

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



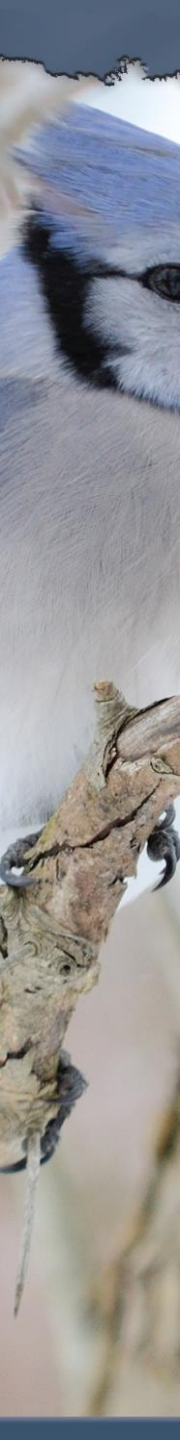


Unit Testing

- By far the most common testing strategy.
- Each unit of an application may be tested.
  - method, class, module (package in Java).
- Should be done during development.
  - Finding and fixing early, lowers development costs (e.g. programmer time).
  - A test suite is built up.

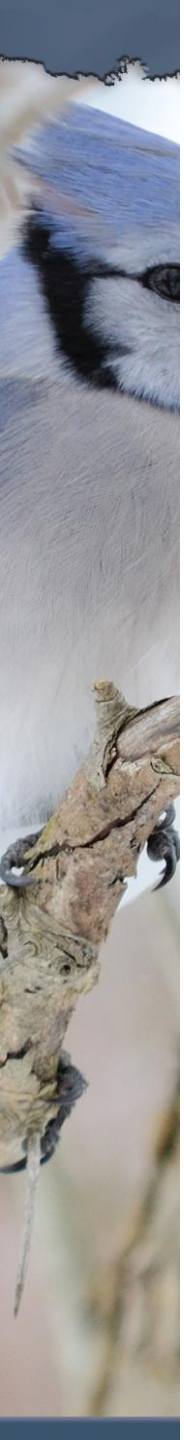
Unit Testing - Example

- Consider the following calculation:
  - $y = \sqrt{\sin\frac{x}{3}}$
  - We can create test-case pairs
  - (x = 0, y = 0), (x=$\frac{\pi}{2}$, y = 0.7071067), …
  - Two (or three?) main operations
    - Test each separately -> unit testing
    - sin(x), $\sqrt{x}$



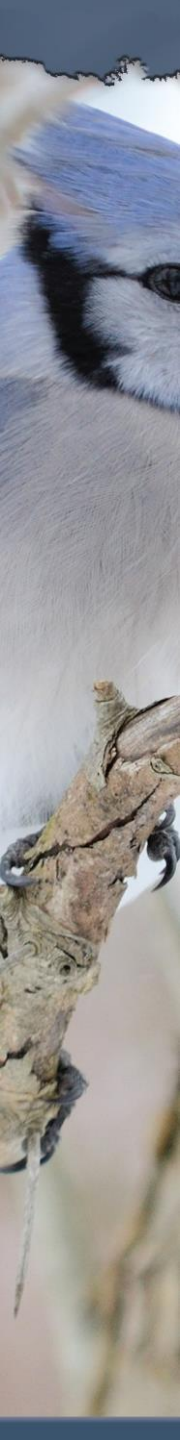
JUnit

- *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



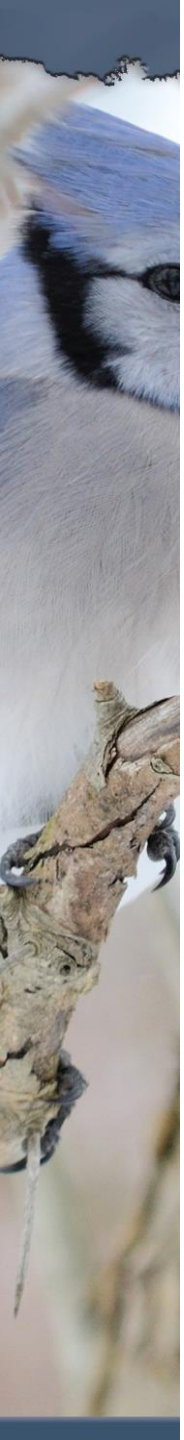
Test Automation

- Good testing is a creative process, but ... thorough testing is time consuming and repetitive.
- *Regression testing* involves re-running a set of tests.
- Use of a *test rig* or *test harness* can relieve some of the burden.
  - Classes are written to perform the testing.
  - Creativity focused in creating these.



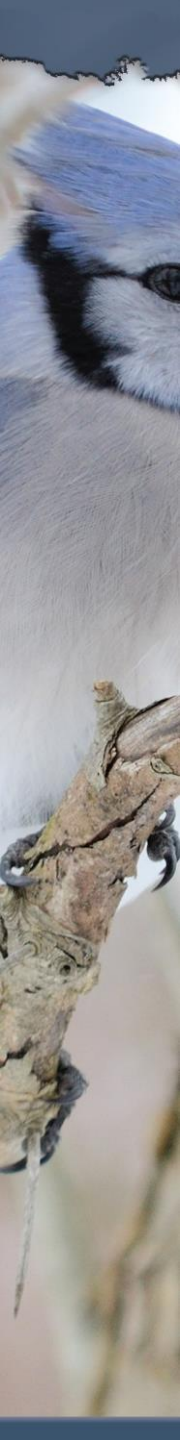
Choosing a test strategy

- 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.



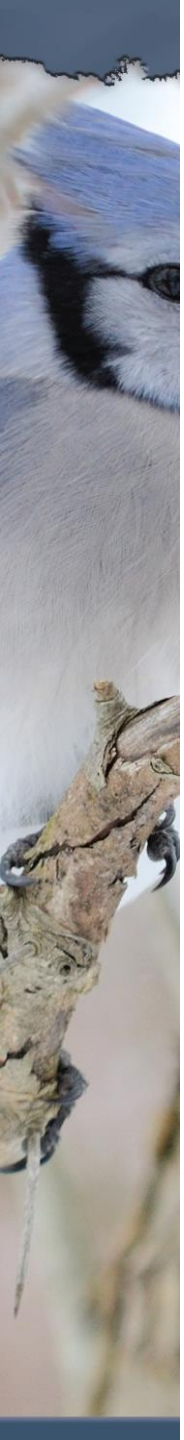
Debugging Skills

- 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.



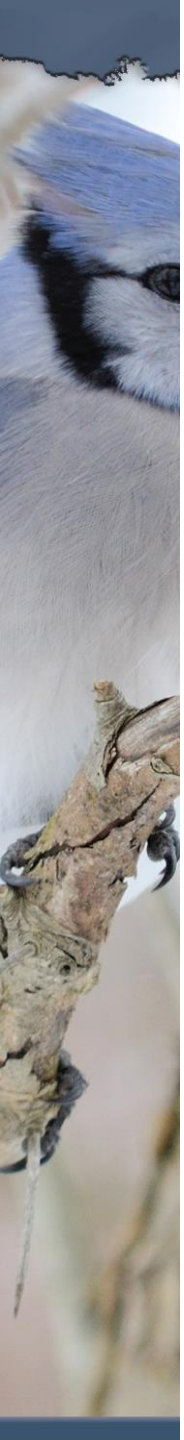
Debugging Techniques

- Manual walkthroughs
- Print statements
- Debuggers



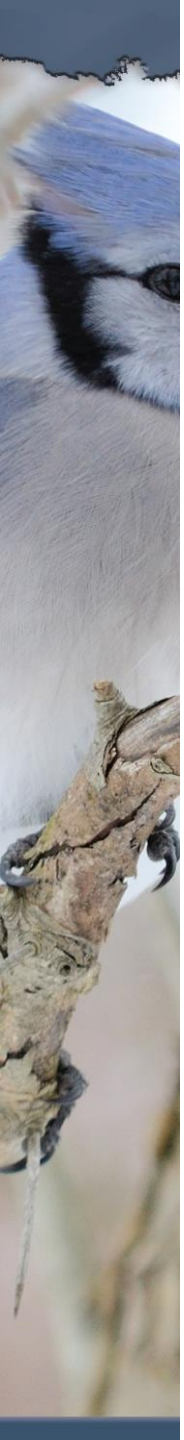
Manual walkthroughs

- 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.



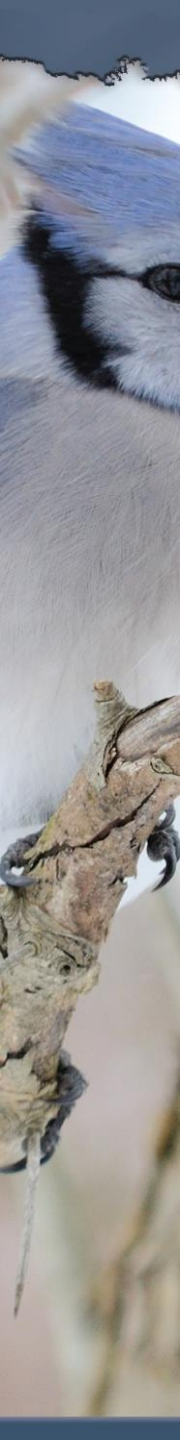
Tabulating object state

- An object's behavior is usually determined by its state.
- Incorrect behavior is often the result of incorrect state.
- Tabulate the values of all fields.
- Document state changes after each method call.



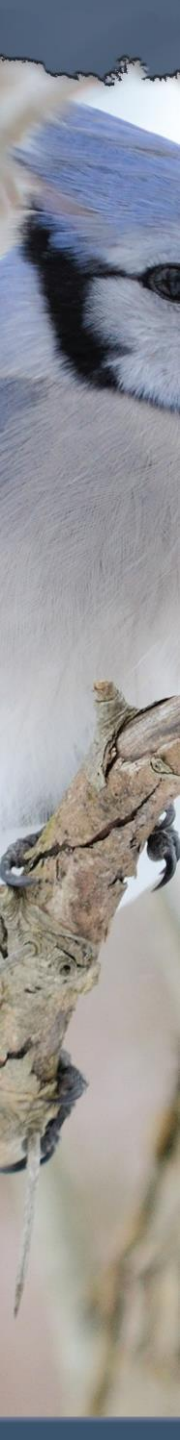
Verbal walkthroughs

- Explain to someone else what the code is doing.
  - They might spot the error.
  - The process of explaining might help you to spot it for yourself.
- Group-based processes exist for conducting formal walkthroughs or *inspections*.



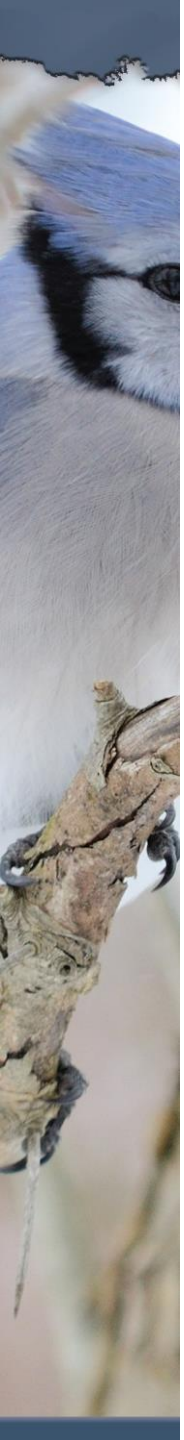
Print statements

- 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.



Debuggers

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



Review

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