CSE 1325: Object-Oriented Programming

Lecture 05

Testing, Errors, and Debugging

Mr. George F. Rice

george.rice@uta.edu

Office Hours:
Prof Rice 12:30 Tuesday and Thursday in ERB 336
For TAs see this web page

What do you get when you put root beer in a square glass? Beer.

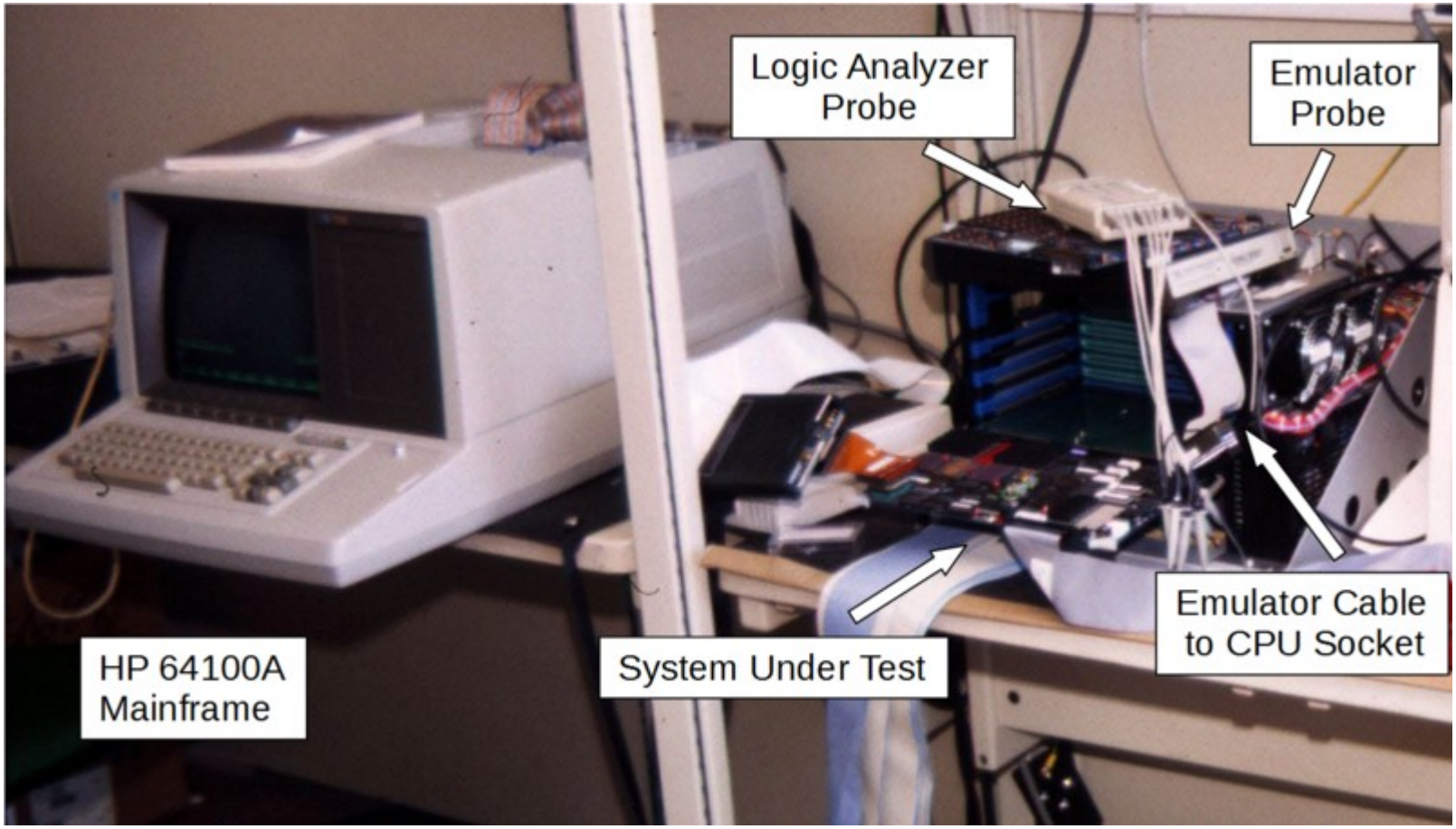
Today's Topics

- Errors
  - Avoiding Errors
  - Planning for Errors
  - Reporting Errors
    - System.out vs System.err
    - Assert
    - Simple exception handling
- Testing
- Debugging

Image by Tumisu of Pixabay – No attribution is required
https://pixabay.com/photos/ethics-right-wrong-ethical-moral-2991600/

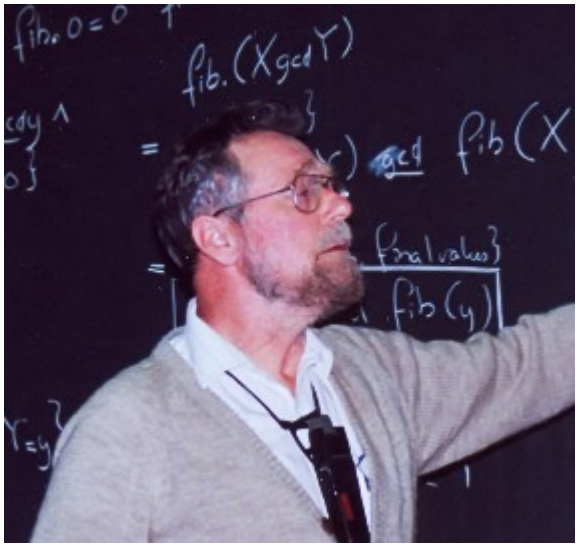
Definitions

- Debugging: The process of removing bugs
- Programming: The process of adding bugs



Errors

- " … I realized that from now on a large part of my life would be spent finding and correcting my own mistakes."
  - Maurice Wilkes, 1949
- When we write programs, errors are natural and unavoidable; the question is, how do we deal with them?
  - Organize software to minimize errors.
  - Eliminate most of the errors we made anyway.
    - Debugging
    - Testing
  - Make sure the remaining errors are not serious.
- My guess is that avoiding, finding, and correcting errors is 95% or more of the effort for serious software development.
  - You can do much better for small programs.
    - or worse, if you're sloppy



"Program testing can be used to show the presence of bugs, but never to show their absence!"

Dr. Edsger Dijkstra

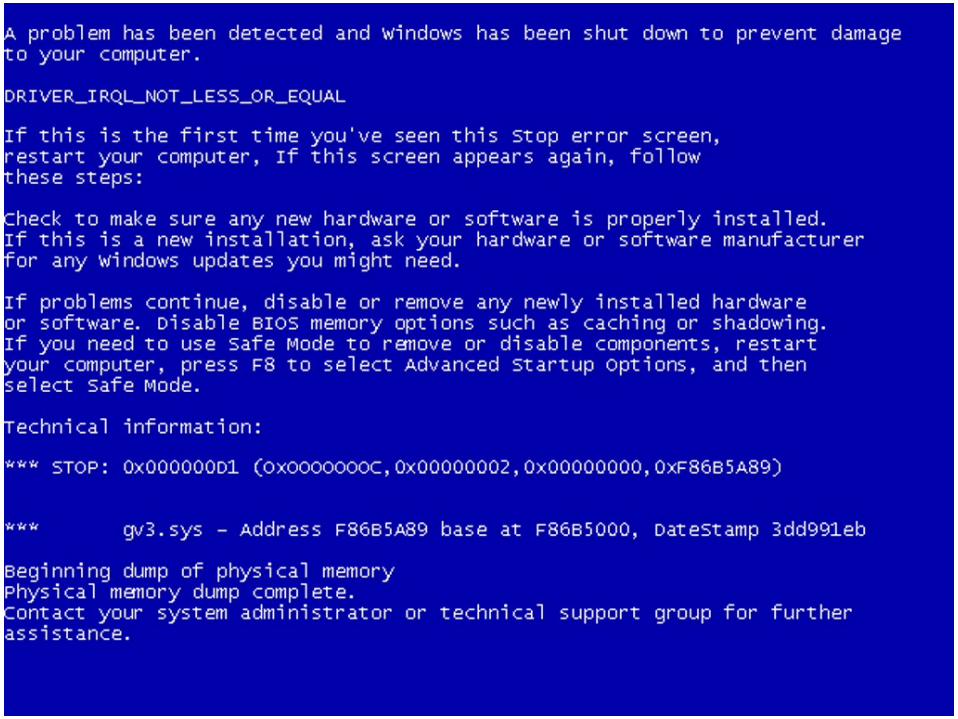
Sources of errors

- Ambiguous Requirements
  - “What’s this *actually* supposed to do?”
- Incomplete programs
  - “I’ll get around to that ... tomorrow”
  - Traditionally, pending work is marked with TODO (recently, #TODO)
- Unexpected arguments
  - “But **sqrt()** isn’t *supposed* to be called with **-1** as its argument”
- Unexpected input
  - “But the user was supposed to input an *integer*”
- Code that simply doesn’t do what it was supposed to do
  - “I don’t always test my code, when when I do, I test it in production!”

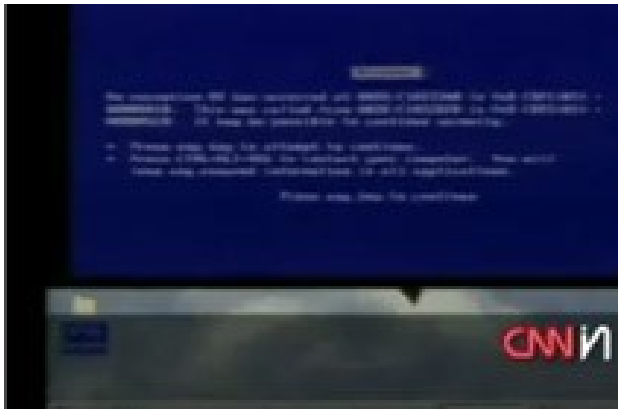
Your JOB is to reasonably respond to all possible events <u>without</u> introducing undue burden on the primary use case(s)

Kinds of Errors

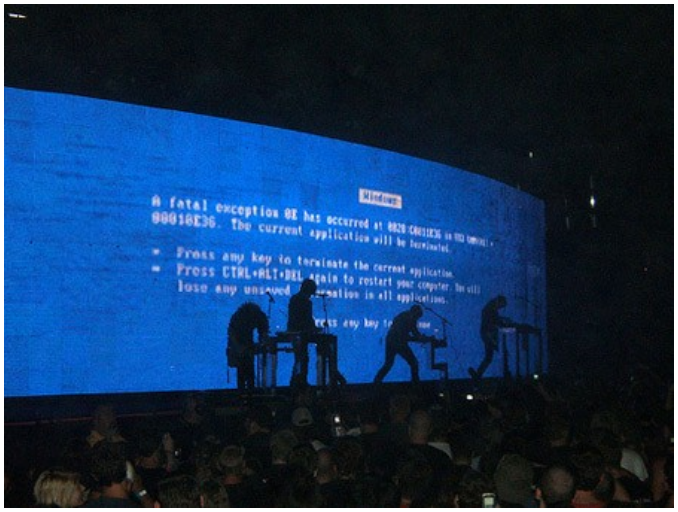
- Compile-time errors
  - Syntax errors
  - Type errors
- Link-time errors
  - Missing libraries
  - Missing .class files
- Run-time errors
  - Unreported (crash)
  - Reported (exceptions and errors)
  - Handled (exception handlers)
- Logic errors
  - Detected by automated tests (regression test, specification tests)
  - Detected by programmer (code runs, but produces incorrect output)



Embarrassing BSoDs



Microsoft USB Demo at Comdex



Nine Inch Nails, in concert

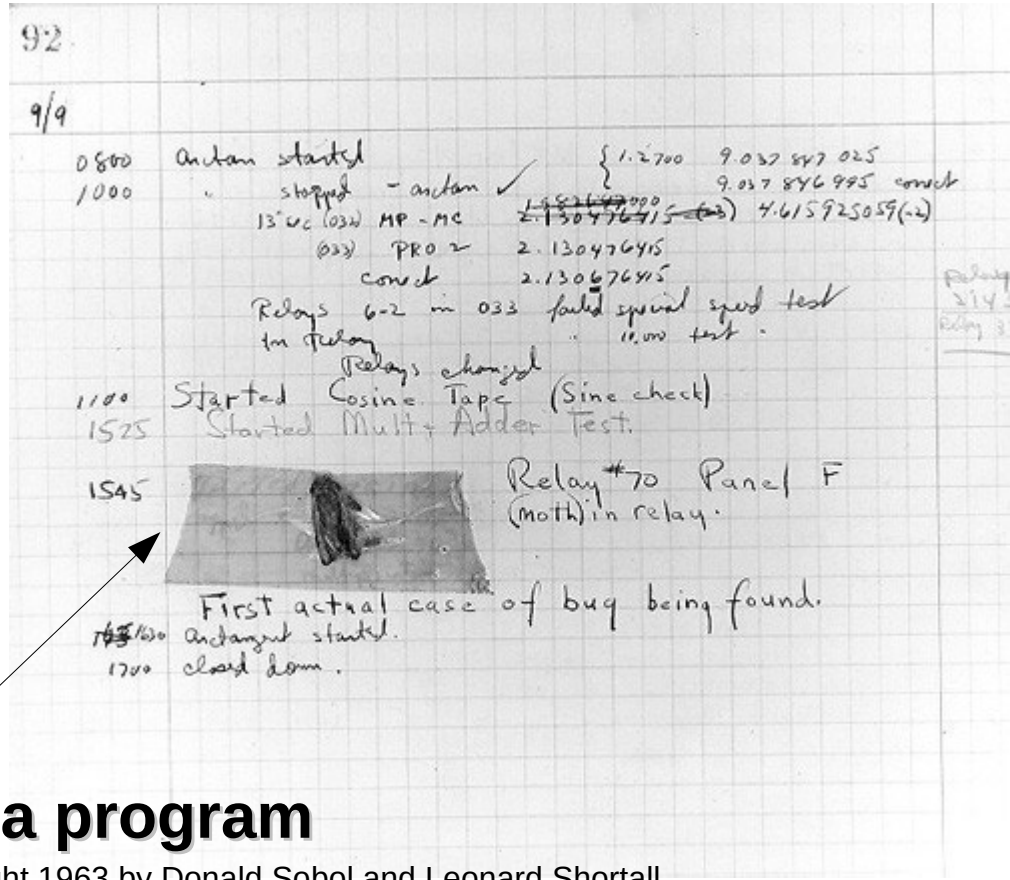
Bank of America ATM

2008 Olympics in Beijing

Even highly regarded business software fails sometimes. Don’t let this happen to you, though...

Bugs

- Every program that you write **will** have errors (commonly called "**bugs**")
  - It won't do what you want
  - It will do what you don't want
  - It will exit with an exception, stack trace, and / or core dump
  - It will lock up the machine (!)
- Fixing a program is called "**debugging**"
- Handling errors well simplifies debugging



First bug ever found in a program

First bug image Courtesy of the Naval Surface Warfare Center, Dahlgren, VA., 1988.
U.S. Naval Historical Center Online Library Photograph NH 96566-KN

A Note on Error Handling

- Error handling is HARDER than “ordinary code”
  - There is basically just one way things can work right
  - There are many ways that things can go wrong
- Wider distribution increases the cost of poor error handling
  - Break your own code, that’s your own problem
  - Break your friends’ code, it’s a few people’s problem
  - Break widely used code, it’s potentially a *disaster*
    - And they may have no way of recovering

303,189 views | Oct 6, 2018, 07:40pm

Microsoft Warns Windows 10 Update Deletes Personal Data

Gordon Kelly Senior Contributor ⓘ
Consumer Tech
I write about technology's biggest companies

Although Microsoft MSFT +1.5% introduced new Windows 10 monthly charges, that has not improved the quality of its updates. In fact, Microsoft has just warned all Windows 10 users that its latest upgrade may permanently delete their personal data…

https://www.forbes.com/sites/gordonkelly/2018/10/06/microsoft-windows-10-

The Compiler is Your Friend

Get the Program to Compile

- We LOVE compiler errors – much easier than debugging!
- Any compiler errors in this program?

```
class Errorful {
  int main() {
  {
    string name;
    in.nextLine(name);
    out.println(“Hello, “,  name, ‘!’)
    if (Name = 'George' out.println(" (prof)");
  }
```

```
ricegf@antares:~/dev/202108/05/code_from_slides$ javac Errorful.java
Errorful.java:6: error: illegal character: '\u201c'
    out.println(“Hello, “,  name, ‘!’);
                ^
Errorful.java:6: error: not a statement
    out.println(“Hello, “,  name, ‘!’);
                 ^
Errorful.java:6: error: ';' expected
    out.println(“Hello, “,  name, ‘!’);
                  ^
Errorful.java:6: error: illegal character: '\u201c'
    out.println(“Hello, “,  name, ‘!’);
                  ^
Errorful.java:6: error: not a statement
    out.println(“Hello, “,  name, ‘!’);
                       ^
Errorful.java:6: error: ';' expected
    out.println(“Hello, “,  name, ‘!’);
                         ^
Errorful.java:6: error: illegal character: '\u2018'
    out.println(“Hello, “,  name, ‘!’);
                           ^
Errorful.java:6: error: illegal character: '\u2019'
    out.println(“Hello, “,  name, ‘!’);
                            ^
Errorful.java:7: error: unclosed character literal
    if (Name = 'George' out.println(" (prof)");
             ^
Errorful.java:7: error: unclosed character literal
    if (Name = 'George' out.println(" (prof)");
                  ^
Errorful.java:7: error: not a statement
    if (Name = 'George' out.println(" (prof)");
                   ^
Errorful.java:8: error: reached end of file while parsing
  }
  ^
12 errors
ricegf@antares:~/dev/202108/05/code_from_slides@
```

A few...

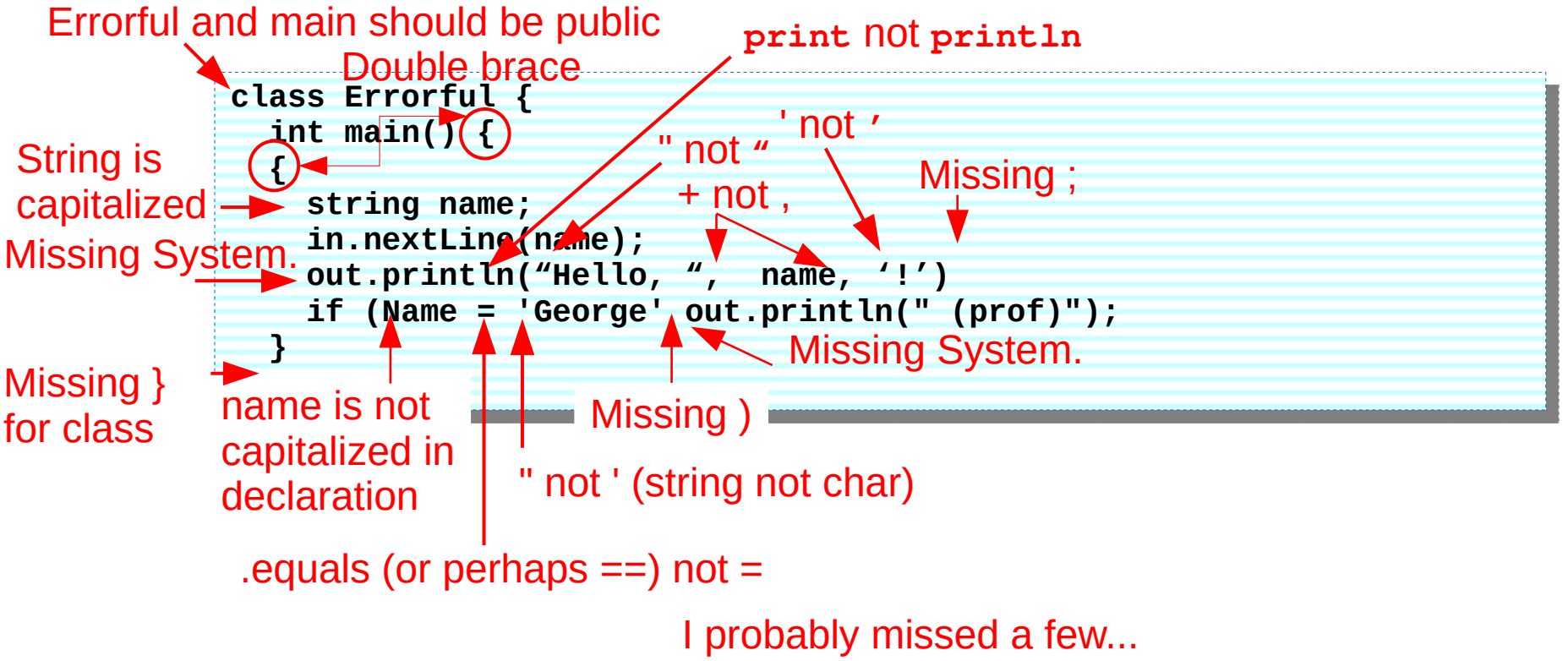
But fewer than g++ →

```
student@cse1325:/media/sf_dev/04$ g++ --std=c++17 bad_code.cpp
bad_code.cpp:5:15: error: stray '\342' in program
   std:cout << ◆◆◆Hello, << name << '!'
               ^
bad_code.cpp:5:16: error: stray '\200' in program
   std:cout << ◆◆◆Hello, << name << '!'
                ^
bad_code.cpp:5:17: error: stray '\234' in program
   std:cout << ◆◆◆Hello, << name << '!'
                 ^
bad_code.cpp:5:36: error: stray '\342' in program
   std:cout << "Hello, << name << ◆◆◆!'
                                  ^
bad_code.cpp:5:37: error: stray '\200' in program
   std:cout << "Hello, << name << ◆◆◆!'
                                   ^
bad_code.cpp:5:38: error: stray '\230' in program
   std:cout << "Hello, << name << ◆◆◆!'
                                    ^
bad_code.cpp:5:40: error: stray '\342' in program
   std:cout << "Hello, << name << '!◆◆◆
                                  ^
bad_code.cpp:5:41: error: stray '\200' in program
   std:cout << "Hello, << name << '!◆◆◆
                                   ^
bad_code.cpp:5:42: error: stray '\231' in program
   std:cout << "Hello, << name << '!◆◆◆
                                    ^
bad_code.cpp:6:14: error: stray '\342' in program
   if (Name = ◆◆◆George' cout << " (prof)" << endl;
              ^
bad_code.cpp:6:15: error: stray '\200' in program
   if (Name = ◆◆◆George' cout << " (prof)" << endl;
               ^
bad_code.cpp:6:16: error: stray '\230' in program
   if (Name = ◆◆◆George' cout << " (prof)" << endl;
                ^
bad_code.cpp:6:23: error: stray '\342' in program
   if (Name = 'George◆◆◆ cout << " (prof)" << endl;
                       ^
bad_code.cpp:6:24: error: stray '\200' in program
   if (Name = 'George◆◆◆ cout << " (prof)" << endl;
                        ^
bad_code.cpp:6:25: error: stray '\231' in program
   if (Name = 'George◆◆◆ cout << " (prof)" << endl;
                         ^
bad_code.cpp:6:35: error: stray '\342' in program
   if (Name = 'George' cout << ◆◆◆ (prof)" << endl;
                               ^
bad_code.cpp:6:36: error: stray '\200' in program
   if (Name = 'George' cout << ◆◆◆ (prof)" << endl;
                                ^
bad_code.cpp:6:37: error: stray '\234' in program
   if (Name = 'George' cout << ◆◆◆ (prof)" << endl;
                                 ^
bad_code.cpp:6:45: error: stray '\342' in program
   if (Name = 'George' cout << " (prof)◆◆◆ << endl;
                                       ^
bad_code.cpp:6:46: error: stray '\200' in program
   if (Name = 'George' cout << " (prof)◆◆◆ << endl;
                                        ^
bad_code.cpp:6:47: error: stray '\235' in program
   if (Name = 'George' cout << " (prof)◆◆◆ << endl;
                                         ^
bad_code.cpp: In function 'int main()':
bad_code.cpp:3:3: error: 'String' was not declared in this scope
   String name;
   ^~~~~~
bad_code.cpp:4:3: error: 'cin' was not declared in this scope
   cin >> name;
   ^~~
bad_code.cpp:4:3: note: suggested alternative: 'main'
   cin >> name;
   ^~~
   main
bad_code.cpp:4:10: error: 'name' was not declared in this scope
   cin >> name;
          ^~~~
bad_code.cpp:5:7: error: 'cout' was not declared in this scope
   std:cout << "Hello, << name << '!'
       ^~~~
bad_code.cpp:5:18: error: 'Hello' was not declared in this scope
   std:cout << "Hello, << name << '!'
                  ^~~~~
bad_code.cpp:5:25: error: expected primary-expression before '<<' token
   std:cout << "Hello, << name << '!'
                         ^~
bad_code.cpp:6:3: error: expected primary-expression before 'if'
   if (Name = 'George' cout << " (prof)" << endl;
   ^~
bad_code.cpp:7:2: error: expected '}' at end of input
 };
```

Building Your Code with Fewer Bugs

Get the Program to Compile

Missing import java.util.Scanner and `Scanner in = new Scanner(System.in);`
Errorful and main should both be public
Should be `public static void main(String[] args)`



A good syntax-checking editor is a great tool! Experience helps, too...

Avoiding Errors

Defensive Coding

- Clearly written code helps to avoid bugs
  - Comments
    - Explain design ideas, NOT how Java works*
  - Use meaningful names, e.g., studentName not n
  - Indent
    - Use spaces, not tabs! Editor tab settings vary
    - Use a consistent visual layout – whitespace is *important*
    - A good code-sensitive editor will help
  - Break code into small methods
    - Try to avoid methods longer than a screen (though that varies)
  - Avoid complicated code sequences
    - Try to avoid deeply nested loops, nested if-statements, etc.
      (But, obviously, you sometimes need those)
  - Use library facilities
    - The most reliable code you will ever write is a library call

* The latter is called “comment inversion”

Planning for Errors
Data Validation

- Verify your inputs!
  - Check constructor parameters
  - Check method parameters
  - Check (**carefully!!!**) user-provided data
  - (Sometimes) check results - “sanity test”
- In case of error, you have options
  - **Log an error** and continue – often in production code
  - **Abort** with a message – typically in test only
  - **Print an error** to System.err – usually with command line tools, for user-created error condition
  - **Throw an exception or error** – in classes

Check your inputs

- Before trying to use an input value, check that it meets your expectations/requirements
  - Function arguments
  - Data from input (**System.in**)

“Data Validation”

- EXAMPLE: SQL Injection Attack (common against poorly coded websites)

```
// userName has been authenticated by matching password
Scanner in = new Scanner(System.in);
itemName = in.nextLine();
String query = "SELECT * FROM items WHERE owner = '" + userName
             + "' AND itemname = '" + itemName + "'";
sda = new SqlDataAdapter(query, conn);
```

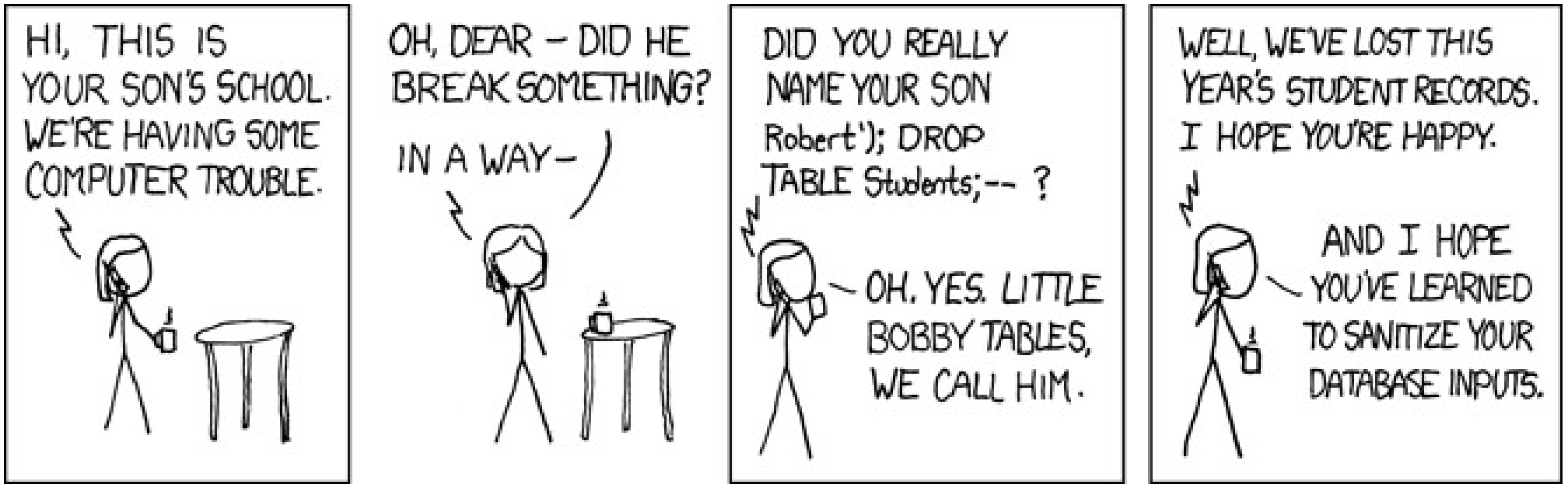
- sda where the user enters “book”:

```
SELECT * FROM items WHERE owner = 'ricegf' AND itemname = 'book';
```

- sda where the user enters “book' OR 'a'='a”:

```
SELECT * FROM items WHERE owner = 'ricegf' AND itemname = 'book' OR 'a'='a';
```

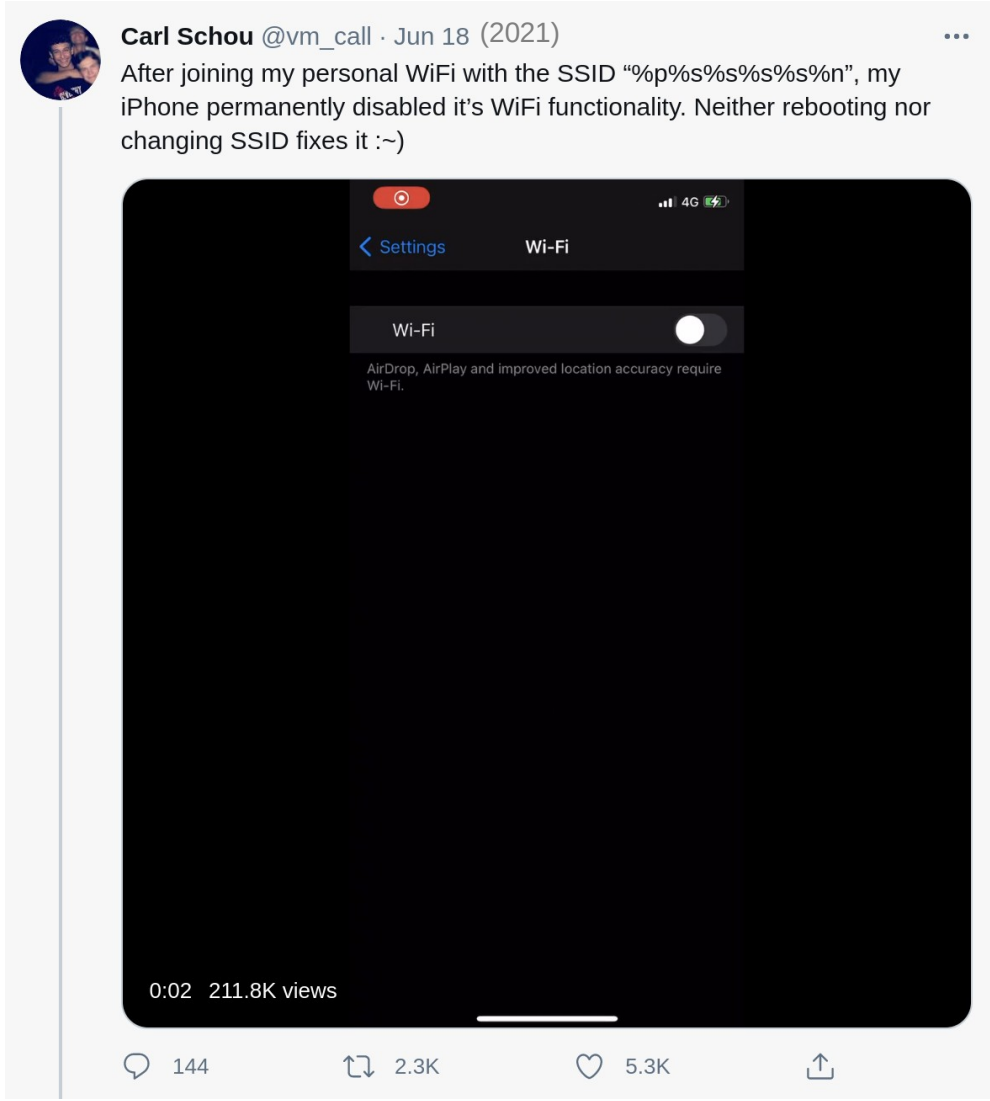
Messing with Modern Technology



• Planning for Errors

Check your inputs

- Another example “in the wild”
- A valid but unusual SSID can cause an iPhone to lose wifi
  - Recovery requires a hard reset, losing all setting and local data
- **DO NOT TEST THIS ON YOUR PHONE!**
  - And isn’t it sad that I have to add this warning?



Validating Inputs

- Try it and catch any errors
  - This assumes invalid inputs throw exceptions
  - We'll cover exceptions shortly
- Write code to ensure the data is valid
  - This can take some significant, tedious code
- Write a "regular expression"
  - For example, ^[a-zA-Z]+$ will match English names
  - Yes, extremely concise (and cryptic!)
  - See the Appendix if you'd like to learn more!

Catching Bad Function Arguments

- The compiler can check “parametric signature”
- The compiler cannot check **assumptions** about arguments and implicit conversion

```
public int area(int length, int width) {
    return length*width;
}

int x3 = area(7, 10);       // correct
int x1 = area(7);           // error: wrong number of arguments
int x2 = area("seven", 2); // error: 1st argument has a wrong type
int x5 = area(7.5, 10);     // ok, but dangerous: 7.5 truncated to 7;
                            //      compiler may warn you
int x = area(10, -7);       // undetected error: bad assumptions
                            //   the types are correct,
                            //   but the values make no sense
```

Planning for Errors

Checking Pre-conditions

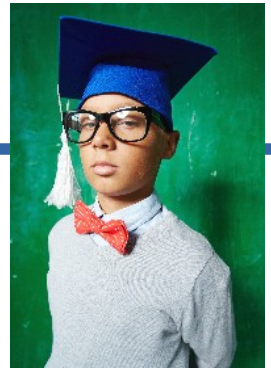
- What does a function require of its arguments?
  - Such a requirement is called a pre-condition
  - Sometimes, it's a good idea to check it

```
public int area(int length, int width) {  // calculate area of a rectangle
    // length and width must be positive
    if (length<=0 || width <=0) throw new IllegalArgumentException("Bad area");
    return length*width;
}
```

We'll get to exceptions shortly

Challenge

How could the result for area() fail
even if the pre-condition held (succeeded)?



Planning for Errors

Edge Cases and Post-conditions

```java
public class Area {
    public static int area(int length, int width) {  // calculate area of a rectangle
        // length and width must be positive
        if (length<=0 || width <=0) throw new IllegalArgumentException("Bad area");
        return length*width;
    }
    public static void main(String[] args) {
        int length = 14;
        int width = 10;

        System.out.println("Area of " + length + " x " + width + " is "
            + area(length, width));

        length = Integer.MAX_VALUE; // The largest 2's complement int in Java
        width = 2;                  // positive integer supported by Java

        System.out.println("Area of " + length + " x " + width + " is "
            + area(length, width));
    }
}
```

```
ricegf@antares:~/dev/202108/05/code_from_slides$ javac Area.java
ricegf@antares:~/dev/202108/05/code_from_slides$ java Area
Area of 14 x 10 is 140
Area of 2147483647 x 2 is -2
ricegf@antares:~/dev/202108/05/code_from_slides$
```

Checking Post-conditions

- What must be true when a function returns?
  - Such a requirement is called a post-condition

```
public static int area(int length, int width) {  // calculate area of a rectangle
    // length and width must be positive
    if (length <= 0 || width <= 0) throw new IllegalArgumentException("Bad area");
    int result = length*width;
    if (result < 0) throw new IllegalArgumentException("Bad area");
    return result;
}
```

```
ricegf@antares:~/dev/202108/05/code_from_slides$ javac Area.java
ricegf@antares:~/dev/202108/05/code_from_slides$ java Area
Area of 14 x 10 is 140
Exception in thread "main" java.lang.IllegalArgumentException: Bad area
        at Area.area(Area.java:6)
        at Area.main(Area.java:20)
ricegf@antares:~/dev/202108/05/code_from_slides@
```

Options for Error Reporting
Exit with Message

- Display message on System.err
- Exit with a non-zero error code

```
public class AreaErr {
    // calculate area of a rectangle
    public static int area(int length, int width) {
        // length and width must be positive
        if (length <= 0 || width <= 0) {
            System.err.println("Invalid length or width!");
            System.exit(-1);
        }
        return length*width;
    }
    public static void main(String[] args) {
        int length = 14;
        int width = -10;

        System.out.println("Area of " + length + " x " + width + " is "
            + area(length, width));
    }
}
```

Why System.err and Not System.out?

```
ricegf@antares:~/dev/202108/05/code_from_slides$ diff AreaErr.java AreaOut.java
1c1
< public class AreaErr {
---
> public class AreaOut {
6c6
<             System.err.println("Invalid length or width!");
---
>             System.out.println("Invalid length or width!");
ricegf@antares:~/dev/202108/05/code_from_slides$ javac AreaErr.java AreaOut.java
ricegf@antares:~/dev/202108/05/code_from_slides$ java AreaOut > results.txt
ricegf@antares:~/dev/202108/05/code_from_slides$ java AreaErr > results.txt
Invalid length or width!
ricegf@antares:~/dev/202108/05/code_from_slides$
```

Send DATA to out
Send ERRORS to err

Redirecting **stdout** DOES cause errors on **out** to "disappear"

Redirecting **stdout** does NOT cause errors on **ERR** to "disappear"

Why return != 0?

Because the environment (e.g., bash) can react based on the returned int

```
ricegf@antares:~/dev/202108/05/code_from_slides$ diff AreaGood.java AreaErr.java
1c1
< public class AreaGood {
---
> public class AreaErr {
12,13c12,13
<         int length = 14;
<         int width  = 10;
---
>         int length =  14;
>         int width  = -10;
ricegf@antares:~/dev/202108/05/code_from_slides@ javac AreaGood.java AreaErr.java
ricegf@antares:~/dev/202108/05/code_from_slides$ java AreaGood
ricegf@antares:~/dev/202108/05/code_from_slides$ if [ $? -eq 0 ]; then echo Success; else echo Failure; fi
Success
ricegf@antares:~/dev/202108/05/code_from_slides$ java AreaErr
Invalid length or width!
ricegf@antares:~/dev/202108/05/code_from_slides@ if [ $? -eq 0 ]; then echo Success; else echo Failure; fi
Failure
ricegf@antares:~/dev/202108/05/code_from_slides$
```

AreaGood doesn't fail, and returns 0
AreaErr fails, and returns -1

The $ bash prompt shows a 0 return, which is the $? bash variable

The @ bash prompt shows a non-0 return (this is a customization in the CSE-VM)

The "make" command that you may have used with C/C++ halts the build on a non-zero return value.

Options for Error Reporting
Log an Error and Continue

- Extensive logging facilities are available
  - But I aim lower :)

```
package qlogger;

/**
 * Trivial logger to System.err or a file.
 *
 * Logging is disabled until Qlogger.enabled is set to true.
 *
 * To redirect output to a file, set Qlogger.out to a PrintStream instance.
 *
 * @author              Professor George F. Rice
 * @version             1.0
 * @license.agreement   Gnu General Public License 3.0
 */
public class Qlogger {
    // Logging is disabled when this field is false (default).
    public static boolean enabled = false;

    // Destination for the logged messages (System.err by default).
    public static java.io.PrintStream out = System.err;

    //Logs the String to the current PrintStream, if enabled.
    public static void log(String s) {if(enabled) out.println(s);}
```

Options for Error Reporting

Log an Error and Continue

- Beware that logs aren't visible by default!

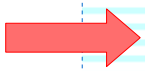
```java
import static qlogger.Qlogger.log; // This imports a single method

public class AreaLog {
    // calculate area of a rectangle
    public static int area(int length, int width) {
        // length and width must be positive
        if (length <= 0 || width <= 0) {
            log("Invalid length or width!"); return 0; // Default value
        }
        return length*width;
    }
    public static void main(String[] args) {
        if(args.length > 0 && args[0].equals("--log")) qlogger.Qlogger.enabled = true;
        int length =  14; int width  = -10;

        System.out.println("Area of " + length + " x " + width + " is "
            + area(length, width));
    }
}
```

```
ricegf@antares:~/dev/202108/05/code_from_slides$ javac AreaLog.java
ricegf@antares:~/dev/202108/05/code_from_slides$ java AreaLog
Area of 14 x -10 is 0
ricegf@antares:~/dev/202108/05/code_from_slides$ java AreaLog --log
Invalid length or width!
Area of 14 x -10 is 0
ricegf@antares:~/dev/202108/05/code_from_slides$
```

Options for Error Reporting
Assert and Abort if False

- Java assertions abort if the argument is false
  - Disabled by default – don't assume they'll execute!
    - Enable with the -ea compiler argument
  - Throws AssertionError if the parameter is false

If the assertion is true, nothing happens.
If the assertion is false, abort the program with an automatically created message.



```
public class AreaAssert {
    // calculate area of a rectangle
    public static int area(int length, int width) {
        // length and width must be positive
        assert length > 0 && width > 0;
        return length*width;
    }
    public static void main(String[] args) {
        if(args.length > 0 && args[0].equals("--log")) qlogger.Qlogger.enabled = true;
        int length =  14;
        int width  = -10;

        System.out.println("Area of " + length + " x " + width + " is "
            + area(length, width));
    }
}
```

Options for Error Reporting
Assert and Abort if False

- Better, you can provide an (optional) message to be printed with the abort message

```
public class AreaAssert {
    // calculate area of a rectangle
    public static int area(int length, int width) {
        // length and width must be positive
        assert length > 0 && width > 0 : "Invalid length or width!";
        return length*width;
    }
    public static void main(String[] args) {
        if(args.length > 0 && args[0].equals("--log")) qlogger.Qlogger.enabled = true;
        int length =  14;
        int width  = -10;

        System.out.println("Area of " + length + " x " + width + " is "
            + area(length, width));
    }
}
```

Options for Error Reporting

Assert and Abort if False

- The message is printed after AssertionError

```
ricegf@antares:~/dev/202108/05/code_from_slides$ javac AreaAssert.java
ricegf@antares:~/dev/202108/05/code_from_slides$ java AreaAssert
Area of 14 x -10 is -140
ricegf@antares:~/dev/202108/05/code_from_slides$ java -ea AreaAssert
Exception in thread "main" java.lang.AssertionError: Invalid length or width!
        at AreaAssert.area(AreaAssert.java:5)
        at AreaAssert.main(AreaAssert.java:14)
ricegf@antares:~/dev/202108/05/code_from_slides@
```

```
public class AreaAssert {
    // calculate area of a rectangle
    public static int area(int length, int width) {
        // length and width must be positive
        assert length > 0 && width > 0 : "Invalid length or width!";
        return length*width;
    }
    public static void main(String[] args) {
        if(args.length > 0 && args[0].equals("--log")) qlogger.Qlogger.enabled = true;
        int length =  14;
        int width  = -10;

        System.out.println("Area of " + length + " x " + width + " is "
            + area(length, width));
    }
}
```

Options for Error Reporting

Throwing / Catching an Exception

- Report an error by throwing an exception
  - This is usually in a class or a predefined library routine
- Handle the error by catching an exception elsewhere

```
public class AreaException {
    // calculate area of a rectangle
    public static int area(int length, int width) {
        // length and width must be positive
        if (length <= 0 || width <= 0)
            throw new IllegalArgumentException("Invalid length or width!");
        return length*width;
    }
    public static void main(String[] args) {
        int length =  14;
        int width  = -10;

        try {
            System.out.println("Area of " + length + " x " + width + " is "
                + area(length, width));
        } catch(Exception e) {
            System.err.println(e.getMessage());
            System.exit(-1);
        }
    }
}
```

1

2

Create (instance) and throw the exception object

Copy (catch) the exception object into e
Print the exception message
Exit and send -1 result to the OS

Exceptions

- Exception handling is general
  - You can't forget about an exception: the program will terminate if someone doesn't handle it (using a **try … catch**)
  - Just about every kind of error can be reported using exceptions
- You still have to figure out what to do about an exception (every exception thrown in your program)
  - Error handling is **never** really simple

Exception – An object created to represent an error or other unusual occurrence and then propagated via special mechanisms until caught by special handling code.

Exception Handling Outline

- You can (and often should) use a pre-defined exception
  - **`RuntimeException("Bad dates")`** is a popular generic choice
  - **`IllegalArgumentException("negative size")`** is also popular
- Throw an exception when an error occurs (**`throw new RuntimeException("[Your Error Message Goes Here]");`**)
  - Optional – many library methods already throw exceptions
- Define a scope in which to watch for an exception (**`try { }`**)
  - The code inside the curly braces (the “try scope”) is monitored for exceptions
- Immediately after the try scope, define one or more exception handling scopes (**`catch (Exception e) { }`**)
  - Add code inside the curly braces to handle each exception , e.g., **`System.err.println(e);`**
- If exiting, **`System.exit(-1)`** with non-zero result reports error to the OS
  - For console apps, but generally not for graphical apps

Java Exceptions

- You may **`catch`** multiple types of exceptions with one **`try`**
  - Simply include more than one **`catch`** clause

```
try {System.out.println(area(length, width));}
catch(InvalidArgumentException e) {System.exit(-1);}
catch(RuntimeException e)         {System.exit(-2);}
```

  - The type of exception matches the **type** of the catch clause
  - If catch clause doesn't match the exception type, the exception will NOT be caught by that catch expression
- Java distinguishes **Exceptions** from **Errors**
  - An Exception, e.g., RuntimeException, SHOULD be caught
  - An Error, e.g., VerifyError, SHOULD NOT be caught
    - **`catch(Exception e)`** will NOT catch an Error, only an Exception
    - **`catch(VerifyError e)`** WILL catch the Error – but don't do this

Range Checking

- Unlike C/C++, Java throws an exception on an array index out of bounds (woot!)
  - Notice that unhandled exceptions ABORT the program

```
public class OutOfRange {
    public static void main(String[] args) {
        int[] ints = {1, 2, 3, 4, 5};
        for(int i=0; i<=5; ++i) System.out.println(" " + ints[i]);
    }
}
```

Uh oh

```
ricegf@antares:~/dev/202108/05/code_from_slides$ javac OutOfRange.java
ricegf@antares:~/dev/202108/05/code_from_slides$ java OutOfRange
 1
 2
 3
 4
 5
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5
out of bounds for length 5
        at OutOfRange.main(OutOfRange.java:4)
ricegf@antares:~/dev/202108/05/code_from_slides$
```

When To Use
Assert vs Error vs Exception

- In general, assert is used for an *interface* error
  - The caller violated the contract in some way
  - Assert throws AssertionError with an error message
  - The assert code can be removed when building for production by *omitting* -ea (“enable assertions”)
- Errors (including AssertionError) are *fatal* errors
  - These are usually thrown by the Java runtime itself
  - e.g., UnsupportedClassVersionError for old .class file
- Exceptions are used for *recoverable* errors
  - Provides an alternate, direct path to error handlers

IMPORTANT

The New Homework Rule

- CSE1325 assignments may NEVER “crash”
  - Unless the requirements specifically permit crashes
- If an exception is thrown, catch and report it to STDERR (**`System.err`** in Java, **`std::cerr`** in C++)
  - You may abort with a clean and helpful message or give the user another attempt if appropriate
  - The requirements take precedence over this rule
- In a few cases, you’ll need to poll for errors
  - For example, C++ doesn’t throw I/O exceptions, so you’ll need to manually check success for I/O
- Always write code to be as “bullet-proof” as practical!

Testing

Basic Testing Options

- Interactive Testing
  - Run the program like a nightmare user – try to break it
  - Some testing is likely to require a written “test procedure”
  - Labor intensive, so avoid this as much as possible
- Regression Testing
  - Write a test program (“unit test” or “regression test”)
  - Systematically try expected inputs for correct operation
  - Try forbidden inputs for proper exception handling
  - When you find a bug, write a new test to ensure it doesn't pop up again
- Test-Driven Development
  - Write tests for requirements first, then prove the tests fail
  - Write your new code, then prove the tests pass
  - Repeat for each requirement
  - Refactor to create a clean and maintainable solution

How **NOT** to Test

Testing

Interactive Testing

- Try the “Normal Case(s)”
  - Run through your Use Cases (the common scenarios you expect from the user – (a little) more on Use Cases later)
  - Be a picky user: Spelling errors? Inconsistencies? Unneeded extra steps? Unclear prompts? Document every possible bug!
- Then be the “User from the Dark Side”
  - Try to break your program – or enlist the help of a good “stress tester” (and treat them like the valuable asset they are!)
  - Enter invalid inputs, push buttons at the wrong time, enter weird Unicode sequences – try to confuse your code!
- Don’t Stop Until You Drop!
  - Write down each bug found, but don’t fix until the list is complete

Testing
Regression Testing

- Code what you expect the program to do!

```
public class Area {
  public static int area(int length, int width) {
    if (length <= 0 || width <= 0) throw new IllegalArgumentException("Invalid length or width!
    return length*width;
  }
}
public class TestArea {    // Regression test for method area
  public static void main(String[] args) {
    // TEST VECTOR #1: Normal Sides (or just use assert!)
    if (Area.area(14, 10) != 140)
        System.err.println("FAIL: 10x14 not 140 but " + Area.area(14,10));

    // TEST VECTOR #2: Identical Length Sides
    if (Area.area(10, 10) != 100)
        System.err.println("FAIL: 10x10 not 100 but " + Area.area(10,10));

    // TEST VECTOR #3: Zero Length Side
    if (Area.area(0, 10) != 0) System.err.println("FAIL: 0x10 not 0 but " + Area.area(0,10));

    // TEST VECTOR #4: Negative Length Side
    try {
        int i = Area.area(-1, -2);
        System.err.println("FAIL: Negative side not exception but " + Area.area(-1, -2));
    } catch (Exception e) { // discard the expected exception
    }
```

A “TEST VECTOR” is just one set of data against which to test the class. Write one block of code per test vector.

Testing Edge Cases

- Pay special attention to “edge cases” (beginnings and ends)
  - Did you initialize every variable?
    - To a reasonable value
  - Did the function get the right arguments?
    - Did the function return the right value?
  - Did you handle the first element correctly?
    - The last element?
  - Did you handle the empty case correctly?
    - No elements
    - No input
    - Null input / end of file
  - Did you open your files correctly?
    - More on this in chapter 11
  - Did you actually read that input?
    - Write that output?

https://www.youtube.com/watch?v=PK_yguLapgA
https://www.bugsnag.com/blog/bug-day-ariane-5-disaster http://www.cs.jhu.edu/~jorgev/cs106/bug.pdf

Test-Driven Development (TDD)

- Test-Driven Development swaps steps 0-1 and 2
  - FIRST convert your requirements into test code
  - THEN verify that any existing code fails the tests
  - FINALLY update the code as little as possible to make the tests pass
- The resulting product is likely sub-optimal
  - We schedule “code refactoring” to “clean up” the product code and improve supportability
  - The functionality of the code doesn’t change – only its structure (for the better, we hope!)
  - Refactoring requires good regression tests

Debugging a Failing Program

Regression Tests Work Great!

(Until they don't...)

- Our regression test from a few slides ago...

```
public class Area {
  public static int area(int length, int width) {
    if (length <= 0 || width <= 0) throw new IllegalArgumentException("Invalid length or width!
    return length*width;
  }
}
public class TestArea {    // Regression test for method area
  public static void main(String[] args) {
    // TEST VECTOR #1: Normal Sides (or just use assert!)
    if (Area.area(14, 10) != 140)
        System.err.println("FAIL: 10x14 not 140 but " + Area.area(14,10));

    // TEST VECTOR #2: Identical Length Sides
    if (Area.area(10, 10) != 100)
        System.err.println("FAIL: 10x10 not 100 but " + Area.area(10,10));

    // TEST VECTOR #3: Zero Length Side
    if (Area.area(0, 10) != 0) System.err.println("FAIL: 0x10 not 0 but " + Area.area(0,10));

    // TEST VECTOR #4: Negative Length Side
    try {
        int i = Area.area(-1, -2);
        System.err.println("FAIL: Negative side not exception but " + Area.area(-1, -2));
    } catch (Exception e) { // discard the expected exception
    }
```

Debugging a Failing Program

Regression Tests Work Great!

(Until they don't...)

- Our regression test from a few slides ago...

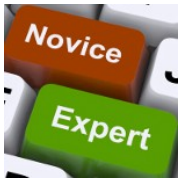
```
ricegf@antares:~/dev/202108/05/code_from_slides$ javac TestArea.java
ricegf@antares:~/dev/202108/05/code_from_slides$ java TestArea
Exception in thread "main" java.lang.IllegalArgumentException: Invalid length
or width!
        at TestArea.area(TestArea.java:4)
        at TestArea.main(TestArea.java:18)
ricegf@antares:~/dev/202108/05/code_from_slides@
```

Uh oh – Test #4 isn't catching the exception!

```
public class Area {
  public s
    if (le
    return
  }
}
public cla
  public s
    // TEST VECTOR #1: Normal Sides (or just use assert)
    if (Area.area(14, 10) != 140)
        System.err.println("FAIL: 10x14 not 140 but " + Area.area(14,10));

    // TEST VECTOR #2: Identical Length Sides
    if (Area.area(10, 10) != 100)
        System.err.println("FAIL: 10x10 not 100 but " + Area.area(10,10));

    // TEST VECTOR #3: Zero Length Side
    if (Area.area(0, 10) != 0) System.err.println("FAIL: 0x10 not 0 but " + Area.area(0,10));

    // TEST VECTOR #4: Negative Length Side
    try {
        int i = Area.area(-1, -2);
        System.err.println("FAIL: Negative side not exception but " + Area.area(-1, -2));
    } catch (Exception e) { // discard the expected exception
    }
```

Debugging a Failing Program
Debugging Options

- Option #1: Mental Execution
  - Pretend that you are the computer running your program
  - Does its output match your expectations? If not, why not?
- Option #2: Add Output
  - Send intermediate data to out ←
  - Or use a logging framework ← Sometimes helpful



Newbie Alert!

- Option #3: Invoke the Debugger ← This is usually your best choice!
  - Rebuild your program with additional “hooks”
    - Our default Ant build.xml file already includes these!
  - Run it inside a program *specifically designed to help*
    - Examine (and even change) variables while it runs
    - Stop when something “interesting” happens

Debugger

- A debugger is a program used to examine and understand a running program
  - Execute and stop at a specific statement, change in a field, or on an exception
  - Single step through program statements (at the current call stack, into a method, or until a return from a method)
  - List (trace) all method calls as they occur
  - Examine (and change!) fields and local variables
- Each IDE has its own debugger
- We'll briefly cover jdb, the official command line debugger
  - But learn (and USE) your IDE's debugger, if applicable!

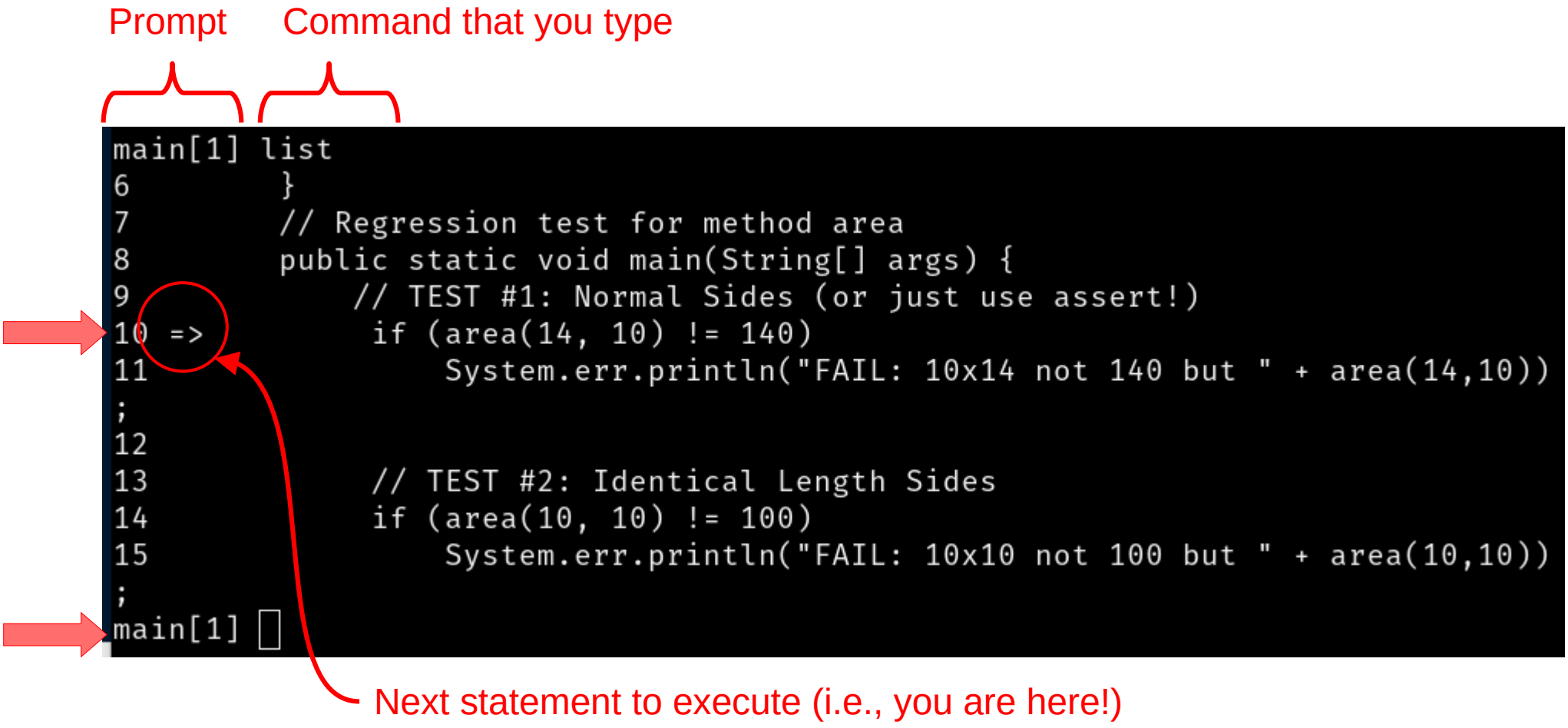
Prepare to Debug!

- Rebuild your code with the -g option (adds info to class file)
- Load class using **`jdb [classname]`** (analogous to **`java [classname]`**)
- (Usually) Tell jdb to stop at 1st statement: **`stop at [classname].main`**
- **`run`** the program (it stops at the first statement if requested)

```
ricegf@antares:~/dev/202108/05/code_from_slides$ javac -g TestArea.java
ricegf@antares:~/dev/202108/05/code_from_slides$ jdb TestArea
Initializing jdb ...
> stop at TestArea.main
Deferring breakpoint TestArea.main.
It will be set after the class is loaded.
> run
run TestArea
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint TestArea.main

Breakpoint hit: "thread=main", TestArea.main(), line=10 bci=0
10              if (area(14, 10) != 140)

main[1] list
```

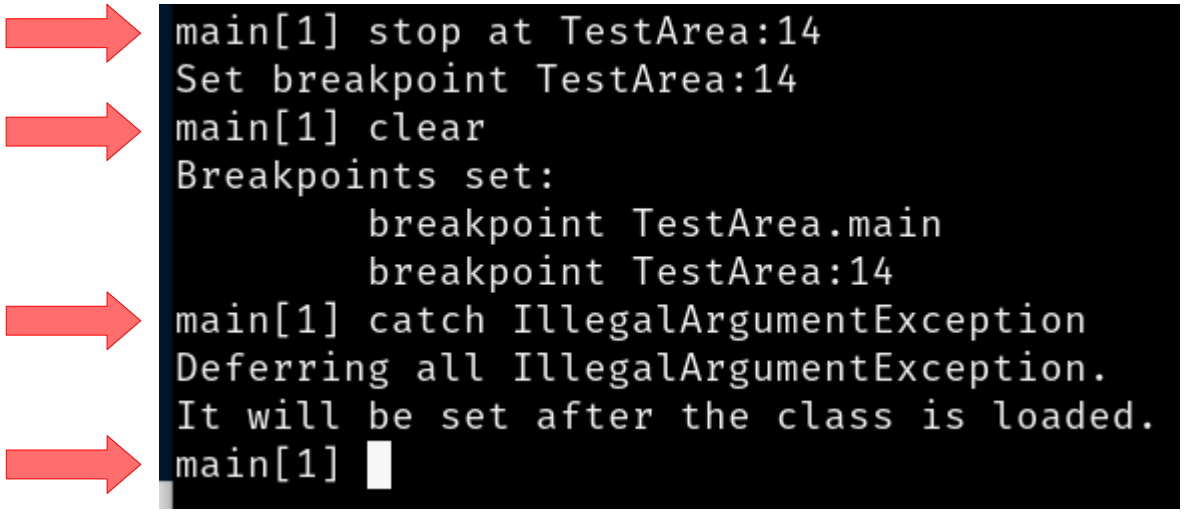
Prepare to Debug!

- **`list`** the statements



Set and List Breakpoints

- **`stop`** can also specify a line number in a class
  - **`clear`** with no parameters lists all breakpoints
  - **`clear`** followed by a breakpoint removes it
- **`catch`** with an exception type (or all) stops when thrown
  - **`ignore`** lists all catchpoints, **`ignore catchpoint`** clears



View Data

- **`cont`** continues from the previous break
- **`step`** executes a statement, stopping at 1st line of called method (if any)
  - **`step up`** executes until the method returns, **`next`** executes to next statement
- **`locals`** lists all local variables
  - **`methods`** and **`fields`** list current / specified class methods / fields, respectively

```
main[1] cont
>
Breakpoint hit: "thread=main", TestArea.main(), line=14 bci=31
14              if (area(10, 10) != 100)

main[1] step
>
Step completed: "thread=main", TestArea.area(), line=3 bci=0
3             if (length <= 0 || width <= 0)

main[1] locals
Method arguments:
length = 10
width = 10
Local variables:
main[1]
```

Eureka!

- And there's the problem!

```
main[1] step up
>
Step completed: "thread=main", TestArea.main(), line=14 bci=38
14              if (area(10, 10) != 100)

main[1] stop at TestArea:18
Set breakpoint TestArea:18
main[1] cont
>
Breakpoint hit: "thread=main", TestArea.main(), line=18 bci=61
18              if (area(0, 10) != 0) System.err.println("FAIL: 0x10 not 0 but "
 + area(0,10));

main[1] step
>
Step completed: "thread=main", TestArea.area(), line=3 bci=0
3             if (length <= 0 || width <= 0)

main[1] locals
Method arguments:
length = 0
width = 10
Local variables:
```

Either it's permissible to have a length or width of 0, OR our test should expect an exception here.
Again, a Requirements issue – the implementer and tester had different interpretations of an edge case.
Happens all the time!

Learn YOUR Debugger

- Here are the most common gdb commands

Run / Step

```
run [class [args]]  -- start execution of application's main class
step                -- execute current line
step up             -- execute until the current method returns to its caller
stepi               -- execute current instruction
next                -- step one line (step OVER calls)
cont                -- continue execution from breakpoint
```

Stop / Break

```
stop <at|in> <location>
                    -- set a breakpoint
                    -- if no options are given, the current list of breakpoints is printed
                    -- if "go" is specified, immediately resume after stopping
                    -- "at" and "in" have the same meaning
                    -- <location> can either be a line number or a method:
                          <class_id>:<line_number>
                          <class_id>.<method>[(argument_type,...)]
clear               -- list breakpoints
clear <class id>.<method>[(argument_type,…)]    -- clear a breakpoint in a method
clear <class id>:<line>                         -- clear a breakpoint at a line
```

Learn YOUR Debugger

- Here are the most common gdb commands

View Code / Data

```
list [line number|method]    -- print source code
classes                      -- list currently known classes
class <class id>             -- show details of named class
methods <class id>           -- list a class's methods
fields <class id>            -- list a class's fields
cocals                       -- print all local variables in current stack frame

print <expr>                 -- print value of expression
dump <expr>                  -- print all object information
set <lvalue> = <expr>        -- assign new value to field/variable/array element

up [n frames]                -- move up the call stack
down [n frames]              -- move down the call stack
```

Learn YOUR Debugger

- Here are the most common gdb commands

Monitor Data

```
catch [uncaught|caught|all] <class id>|<class pattern>
                        -- break when specified exception occurs
ignore [uncaught|caught|all] <class id>|<class pattern>
                        -- cancel 'catch' for the specified exception
watch [access|all] <class id>.<field name>
                        -- watch access/modifications to a field
unwatch [access|all] <class id>.<field name>
                        -- discontinue watching access/modifications to a field
```

List Calls / Returns

```
trace [go] methods
                        -- trace method entries and exits.
trace [go] method exit
                        -- trace the current method's exit, or all methods' exits
untrace [methods]       -- stop tracing method entrys and/or exits
```

Learn YOUR Debugger

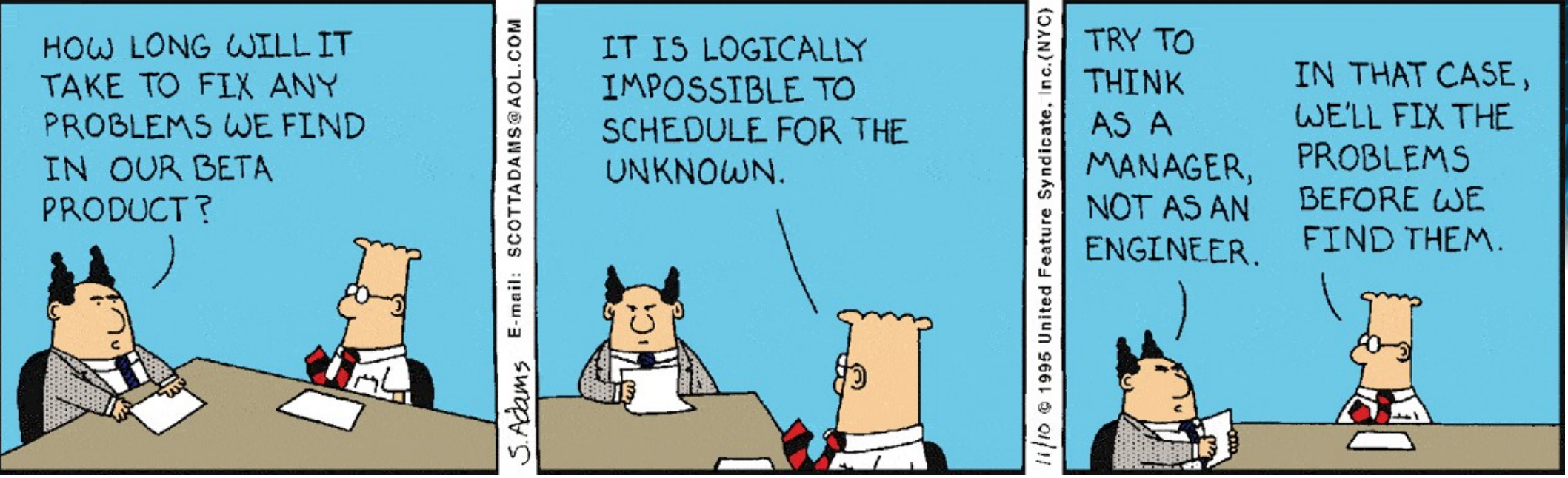
- Here are the most common gdb commands

Automation

!!	-- repeat last command
<n> <command>	-- repeat command n times (often used with **`next`**)
read <filename>	-- read and execute a command file (scripting!)
# <command>	-- discard (no-op) useful for comments in scripts
monitor <command>	-- execute command each time the program stops (e.g., **`list`**)
monitor	-- list monitors
unmonitor <monitor#>	-- delete a monitor (monitor# is from the monitor command)

Miscellaneous

help (or ?)	-- list all commands
exit (or quit)	-- exit debugger



Technical obstacles presented by bugs will undoubtedly not be the greatest challenge you face...

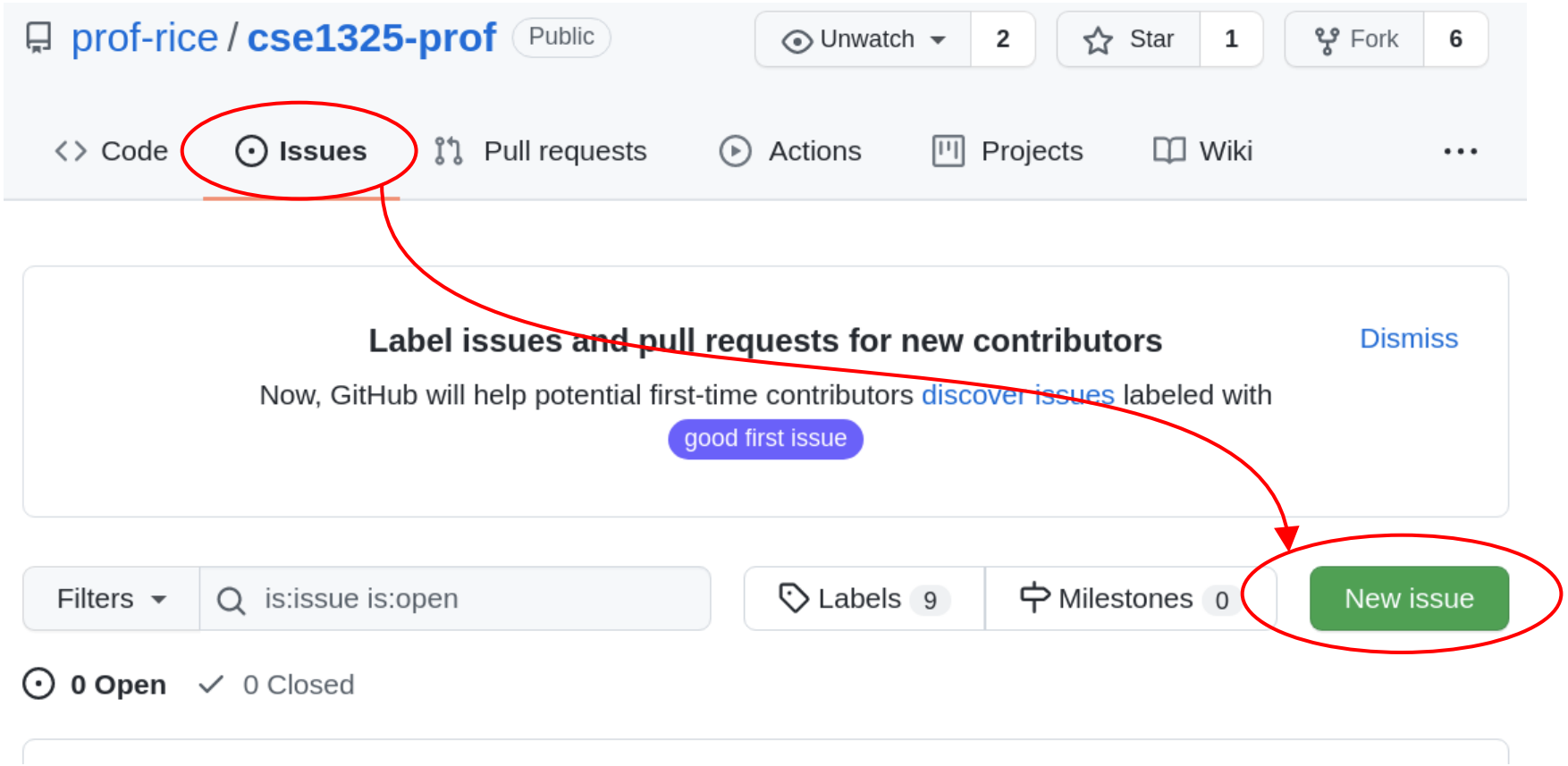
http://amureprints.com/reprints/classroom http://dilbert.com/strip/1995-11-10 http://dilbert.com/strip/1995-11-13

Bug Management

- TODO: is NOT a good way to track features (use Scrum) or bugs!
- Bug trackers offer end-to-end bug management
  - Create bug reports (even by end users!)
  - Triage bug reports: **Verified** (needs to be fixed), **Could Not Duplicate** (developers can't verify that bug), **Duplicate** (already reported, close with link to original), **Won't Fix** (it's a feature, not a bug)
  - Prioritize bug reports by Severity: **Showstopper**, **Critical**, **Severe**, **Important**, **Informational**
    - Showstopper and (usually) Critical bugs must be fixed prior to release
  - Report progress on and close out bug reports with notification
- A verified bug becomes a task on your Sprint Backlog
  - It's priority is determined in part by its severity
  - Fixing a bug means not implementing a new feature
- Innumerable bug trackers exist
  - Atlassian JIRA, Bugzilla, IBM ClearQuest, Excel (ahem), etc.

GitHub Tracks Bugs

- GitHub itself has a built-in bug tracker



Applications and Class Libraries Branch and Fork

- Branching is used to manage change
  - Implement bug fixes
  - Develop a new version
  - Explore concepts and possible new features
- Forking creates a new product
  - Complementary products with similar features
  - Competitive products when the original developer fails to meet customer expectations
    - Free and Open Source greatly enable this!

A second distinct development path for a product is called a **Branch** if followed by the **same** company or project, and a **Fork** if followed by a **different** company or project.

Basic Branching in git

- You create a local branch using ‘-b’

```
git checkout –b bug_013  # Fix bug #13
```

- While in (branch) bug_013, commits are made to that branch, not (branch) main
- You can switch back to “main” using

```
git checkout master
```

- Then back to bug_013 using

```
git checkout bug_013
```

- You can merge your bug fix back into main

```
git checkout main
git merge bug_013
```

Learn (a Lot!) More

https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

Examples of Forks

- Netscape → Firefox
- Debian → Ubuntu → Mint
- Unix System V → BSD Unix → Mac OS X
- MySQL → MariaDB
- OpenOffice → LibreOffice


The End

Appendix A: Planning for Errors

Regular Expressions (regex)

- Java provides “regex”, excellent for data validation
  - Lets you define the “look” of valid data
  - The match method will let you know if the data “looks” valid

```
import java.util.Scanner;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Ints {
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    Pattern pattern = Pattern.compile("^-?\\d+$");
    System.out.println("Enter some integers:");
    while(in.hasNextLine()) {
        Matcher matcher = pattern.matcher(in.nextLine());
        if(matcher.find())
            System.out.println("That's an int!");
        else
            System.out.println("### INVALID INPUT ###");
    }
  }
}
```

```
ricegf@antares:~/dev/202108/0
ricegf@antares:~/dev/202108/0
Enter some integers:
3.14
### INVALID INPUT ###
314
That's an int!
347859304385720348 75
That's an int!
twelve
### INVALID INPUT ###
0x10
### INVALID INPUT ###
-365
That's an int!
19h
### INVALID INPUT ###
ricegf@antares:~/dev/202108/0
```

Ints.java

(Very) Basic Regex Rules

- Most characters match themselves
- A special character matches one of a group
  - \s matches any whitespace except newline
  - \w matches a “word” character – A-Z, a-z, 0-9, or _
  - \d matches a “digit” – 0-9
  - \S, \W, and \D match the opposite of their lowercase version
- Repeaters compactly express multiple characters
  - \d? represents either 0 or 1 digits
  - \d* represents 0 or more digits
  - \d+ represents 1 or more digits
- Parentheses work as expected
  - (ha)+ represents ha, haha, hahahahaha, etc.

```
regex integer{"-?\\d+"};
```

Matches itself
Matches “-” 0 or 1 times
Matches a digit
Matches the digit 1 or more times

What Regex Would Validate These?

- “Professor Rice” or “Professor George Rice”

- A C++ name, e.g., “_counter3”
- A negative integer, e.g., “-108”
- Adding two positive integers, e.g., “42+38”

- Exactly 3 words, e.g., “Go Mavs Go”

- One or more words, e.g., “Hello” or “Bye Bye Baby Bye Bye”

What Regex Would Validate These?

- “Professor Rice” or “Professor George Rice”
  Professor(George)? Rice
- A C++ name, e.g., “_counter3” \w+
- A negative integer, e.g., “-108” -\d+
- Adding two positive integers, e.g., “42+38”
  \d+\+\d+
- Exactly 3 words, e.g., “Go Mavs Go”
  \w+\s\w+\s\w+
- One or more words, e.g., “Hello” or “Bye Bye Baby Bye Bye”
  \w+(\s\w+)*

Regex: Learning More

- We won’t focus on regex this semester
  - Even though they are *incredibly* useful
  - Moreso when writing dialog boxes next section!
- Tutorials are available for further understanding
  - Jenkov: http://tutorials.jenkov.com/java-regex/index.html
  - Oracle: https://docs.oracle.com/javase/tutorial/essential/regex/