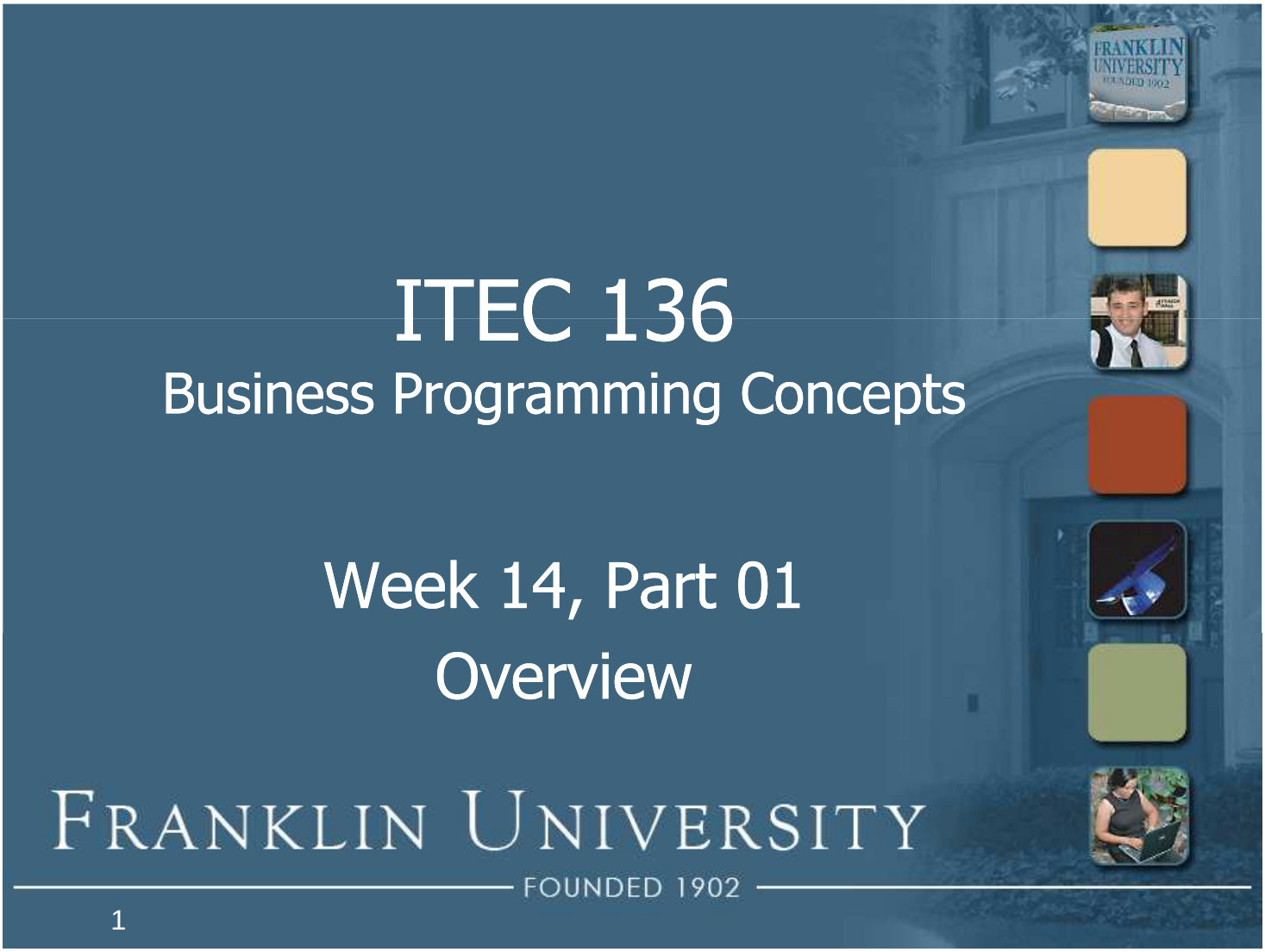


Week 14 Overview

- Week 13 review
  - What is an object? (three parts)
    - State (properties)
    - Identity (location in memory)
    - Behavior (methods)

Week 14 Overview

- Week 13 review
  - Custom JS objects
    - Constructors
    - `this` reference
    - The `prototype` property of functions
  - Benefits of object-orientation
    - Coupling and cohesion (among others)

Week 14 Overview

- Week 13 review
  - 3 of the 5 pillars of OOP
    - Abstraction
    - Encapsulation
    - Composition (code reuse, 2 kinds)

Week 14 Overview

- Week 13 review
  - Exception handling
    - Detection and correction of errors are at different places in code.
    - Exceptions communicate between those places and alter the flow of execution
      - `throw` keyword
      - `try/catch/finally` blocks

Week 14 Overview

- Outcomes
  - Select necessary and sufficient test cases.
  - Use a debugger to examine a running program.
  - Correct runtime errors through a debugger.

Week 14 Overview

- Outcomes
  - Select necessary and sufficient test cases.
  - Use a debugger to examine a running program.
  - Correct runtime errors through a debugger.

FRANKLIN UNIVERSITY

ITEC 136

Business Programming Concepts

Week 14, Part 02

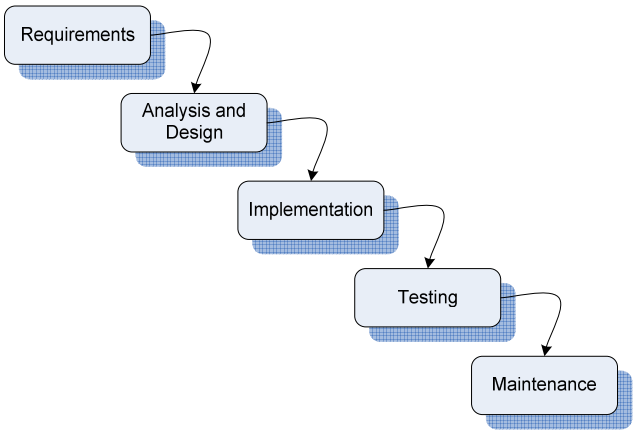
Testing Concepts

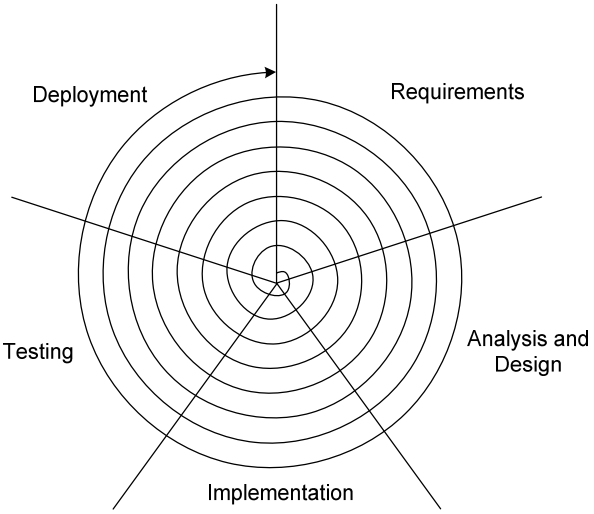
FRANKLIN UNIVERSITY

FOUNDED 1902

Testing Concepts

- Testing in the SDLC





Testing Concepts

- Validation vs. verification
  - Validation: A comparison of the system behavior against what the user actually needs. "Have we build the right software?"

Testing Concepts

- Validation vs. verification
  - Validation: A comparison of the system behavior against what the user actually needs. "Have we build the right software?"

Primarily the work of business analysts and designers. Critical to real-world software success.

Testing Concepts

- Validation vs. verification
  - Validation: A comparison of the system behavior against what the user actually needs. "Have we build the right software?"
  - Verification: A comparison of system behavior against the specification. "Have we built the software right?"

Testing Concepts

- Validation vs. verification
  - Validation: A comparison of the system behavior against what the user actually needs. "Have we build the right software?"
  - Verification: A comparison of system behavior against the specification. "Have we built the software right?"

Testing Concepts

- Validation vs. verification
  - Validation: A c em behavior agai ally needs. "Have software?"
  - Verification: A comparison of system behavior against the specification. "Have we built the software right?"

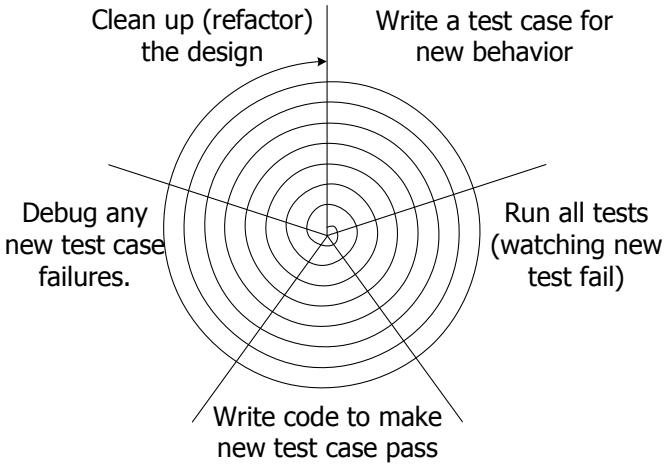
Primarily the work of software and quality assurance engineers. What most people think of when you say "testing." Our focus this week.

Testing Concepts

- Types of testing
  - **Unit testing**: A function or object designed to test the behavior of a another function or object.
    - Operates in isolation of other objects.
    - Provide known inputs to check against known outputs.
    - Group together into a test suite.

Testing Concepts

- Test driven development
  - If testing is good, then why not do it continuously?



Testing Concepts

- Types of testing
  - **Integration testing**: verifies the behavior of larger groupings of modules.
    - Done after unit testing of each component part, yet before system testing.
    - Exposes interface, design problems

Testing Concepts

- Types of testing
  - **System testing**: testing of the entire system assembled from major modules.
    - Done after integration testing
    - Load, security, stress, performance, reliability, etc.

Testing Concepts

- Types of testing
  - **Acceptance testing**: performed by subject matter experts just prior to release. Last chance for bug finding.
    - Done after system testing.
    - Sometimes called *beta* testing.
    - Binary decision (go/no go for release).

Testing Concepts

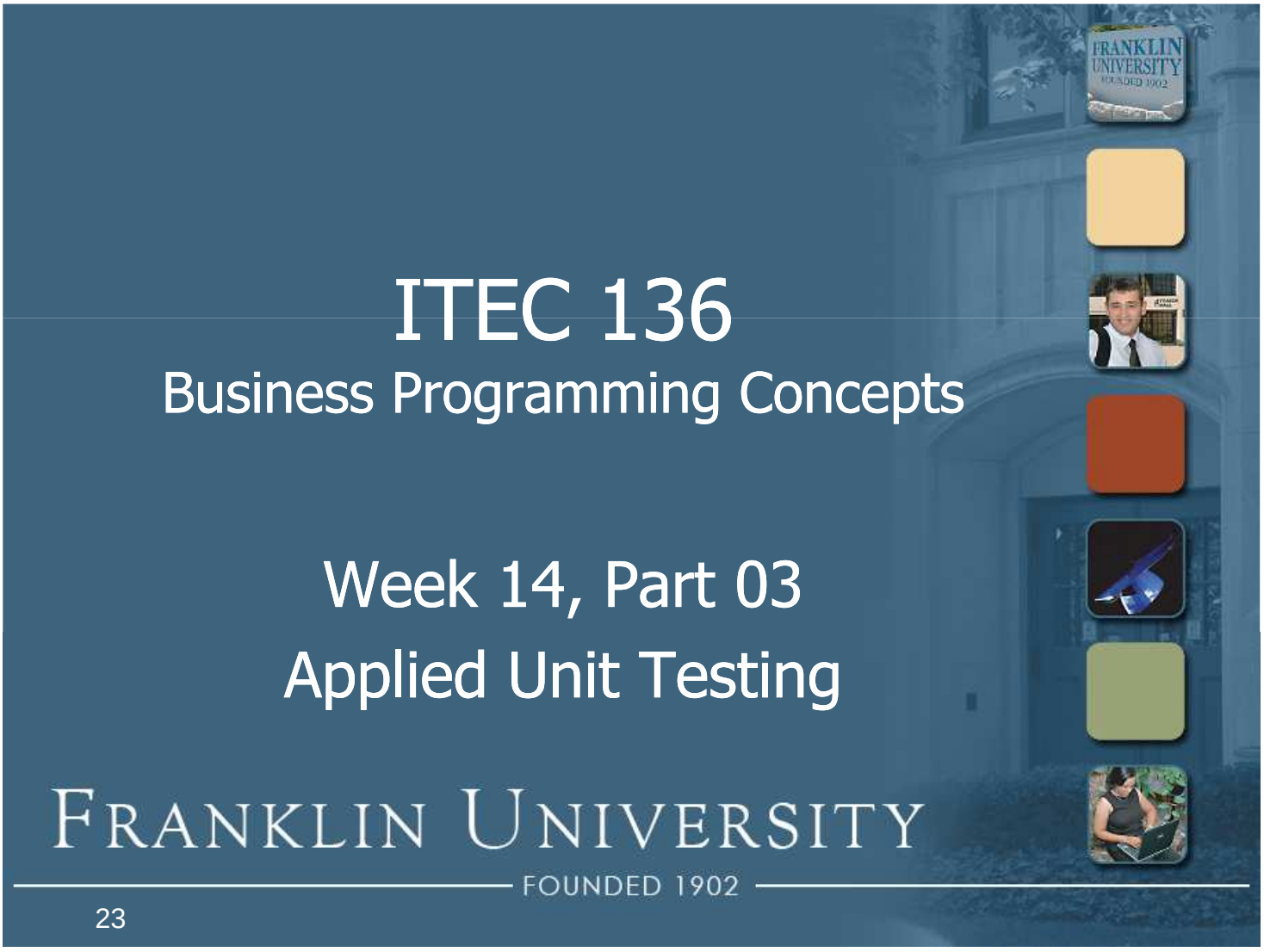
- Types of testing
  - **Regression testing**: re-running old test cases after every bug fix to ensure that the fix introduced no new bugs.
    - Prevents *cycling* of bugs.
    - Acts as a safety net.
    - Permits refactoring (redesign of existing code) while maintaining quality.

Testing Concepts

- Black- vs. white-box testing
  - **Black-box testing**: treats the item under test as a black box, providing only inputs (both valid and invalid) and checking outputs. Does not exploit internal knowledge of how the code works.

Testing Concepts

- Black- vs. white-box testing
  - **White-box testing**: testing all paths through the software using carefully crafted inputs (both valid and invalid).
    - Critical to achieve a high degree of *test coverage*, i.e. the percentage of lines of code exercised by the tests.



Applied Unit Testing

- Unit tests must
  - Be quick and easy to write
  - Run in an automated way
  - Provide value to the programmer
- Method
  - Provide inputs
  - Validate outputs

Applied Unit Testing

- How do we test this:

```
function absoluteValue(number)
{
    if (number < 0)
        return -number;
    return number;
}
```


Applied Unit Testing

- How do we test this:
  - Simple way:

```
function testAbsoluteValueFunction()
{
    if (5 != absoluteValue(-5))
        alert("failed test 1");
    if (5 != absoluteValue(5))
        alert("failed test 2")
}
```

Applied Unit Testing

- How do we test this:
  - Simple way:

Advantages:
- Simple.
- Can be used for regression testing.

```
function testAbsolut
{
    if (5 != absoluteValue(-5))
        alert("failed test 1");
    if (5 != absoluteValue(5))
        alert("failed test 2")
}
```


Applied Unit Testing

- How do we test this:
  - Simple way:

Advantages:
- Simple.
- Can be used for regression testing.

Disadvantages:
- Not easily reused for other kinds of testing
- Too many alerts

```
function testAbsolut
{
    if (5 != absoluteVa
        alert("failed test 1
    if (5 != absoluteValue(5
        alert("failed test 2
}
```

Applied Unit Testing

- Building a unit testing *framework*
  - Principles:
    - Make unit testing easy
    - Take away all the repetitive code
    - Report errors succinctly

Applied Unit Testing

- Building a unit testing *framework*

```
var tests = new UnitTester();
tests.addTests({
  testNegative : function() {
    tests.assertEquals(5, absoluteValue(-5));
  },
  testPositive : function() {
    tests.assertEquals(5, absoluteValue(-5));
  },
  testNonNumber : function() {
    tests.assertEquals("NaN", "" + absoluteValue("x"));
  }
});
tests.runTests();
```

Applied Unit Testing

- Building a unit testing *framework*

```
var tests = new UnitTester();
tests.addTests({
  testNegative : function() {
    tests.assertEquals(5, absoluteValue(-5));
  },
  testPositive : function() {
    tests.assertEquals(5, absoluteValue(-5));
  },
  testNonNumber : function() {
    tests.assertEquals("NaN", "" + absoluteValue("x"));
  }
});
tests.runTests();
```

Expected answer

Actual answer

Applied Unit Testing

- Building a unit testing *framework*

```
function FailedTest(expected, actual) {
    this.expected = expected;
    this.actual = actual;
}

FailedTest.prototype.toString = function() {
    return "expected: " + this.expected +
        ", actual: " + this.actual;
}
```

A failed test throws a `FailedTest` exception capturing how it failed.

Applied Unit Testing

- Building a unit testing *framework*

```
function UnitTester() {
    this.allTests = new Object();
    this.failures = [];
}

UnitTester.prototype.assertEquals = function(
   expected, actual) {
    if (expected.equals && !expected.equals(actual))
        throw new FailedTest(expected, actual);
    else if (!(expected == actual))
        throw new FailedTest(expected, actual);
}
```

Compares expected and actual values.

Applied Unit Testing

- Building a unit testing *framework*

```
UnitTester.prototype.addTests = function(manyTests)
{
    for (index in manyTests) {
        this.addTest(index, manyTests[index]);
    }
}

UnitTester.prototype.addTest = function(name, test)
{
    this.allTests[name] = test;
}
```

Sets up test functions to run.

Applied Unit Testing

- Building a unit testing *framework*

```
UnitTester.prototype.runTests = function() {
  for (var index in this.allTests) {
    if (this.allTests[index] instanceof Function) {
      try {
        this.allTests[index]();
      }
      catch (exception) {
        this.failures.push(index + ": " + exception);
      }
    }
  }
  alert(this.makeResultsString());
}
```

Actually runs the test functions

Applied Unit Testing

- Building a unit testing *framework*

```
UnitTester.prototype.makeResultsString = function()
{
    var str = "";
    for (var index in this.failures)
    {
        str += this.failures[index] + "\n";
    }
    if (str == "")
        return "All tests passed.";
    return str;
}
```

Produces the results.

ITEC 136
Business Programming Concepts

Week 14, Part 04
Selecting Test Cases

FRANKLIN UNIVERSITY
FOUNDED 1902

Selecting Test Cases

- Test coverage
  - Making sure that all parts of your program are exercised by the test cases.
    - Every direction of nested if/else structures
    - Every possible case in a switch statement
    - Every possible way a loop can be run
      - Never iterating (pre-test only)
      - Iterating once
      - Iterating many times

FRANKLIN UNIVERSITY

Selecting Test Cases

- Test coverage
  - Making sure that all kinds of data are tested
    - An "expected" test case
    - "Corner" cases
    - Illegal inputs

Selecting Test Cases

- Test coverage
  - Ex: calculating square roots
    - Expected inputs: numbers from [1…n]
    - Corner cases: 0, [0…1]
    - Illegal inputs: negative numbers

Applied Unit Testing

- Try it:
  - Write test cases using the testing framework to determine if your `merge()` function (which merges two separately sorted arrays) works on many different data sets.

Applied Unit Testing

- Try it:
  - Write test cases using the testing framework to determine if a sorting algorithm actually sorts arrays.

Applied Unit Testing

- Try it:
  - Write a function that builds a histogram table at 10% intervals (i.e. given an array of data in the range [0, 100] output an array with 11 "buckets" containing the count of elements that fall in those buckets).
  - Write tests to verify that it works.

FRANKLIN UNIVERSITY

ITEC 136

Business Programming Concepts

Week 14, Part 05

Debugging Tools

FRANKLIN UNIVERSITY

FOUNDED 1902

Debugging Tools

- Old school approach
  - Logging: debugging statements placed strategically in program code.

```
function log(div, message) {
    document.getElementById(div).innerHTML +=
       message + "<br />";
}
// then later...
if (debug == true) {
    log("debug", "Got to here");
}
```

Debugging Tools

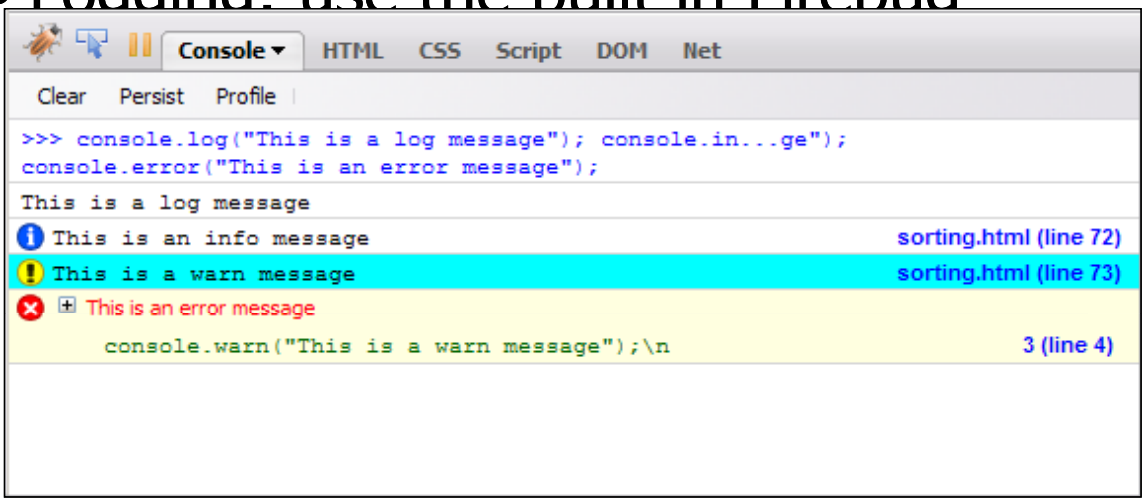
- New school approach
  - Logging: use the built in Firebug logging console!

```
console.log("This is a log message");
console.info("This is an info message");
console.warn("This is a warn message");
console.error("This is an error message");
```

FRANKLIN UNIVERSITY

Debugging Tools

- New school approach
  - Logging: use the built in Firebug



Debugging Tools

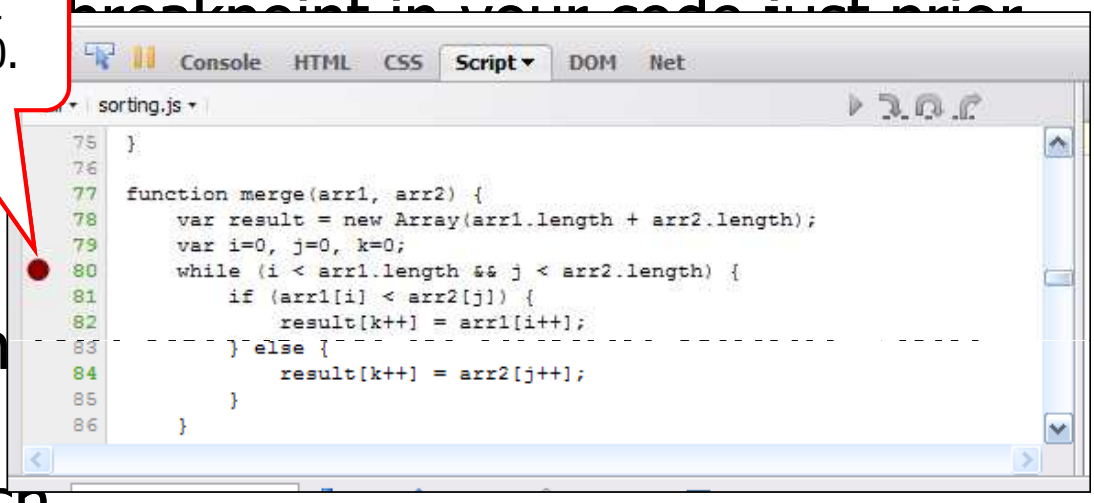
- Debuggers
  - Programs that allow you to examine the state of another running program.
    - Built in to the IDE in which you program
      - Stop your program at a particular point (*breakpoint*)
      - Inspect the contents of a variable (*inspect* or *watch*)
      - Step through a program as it executes (*step into*, *step over*, *step out*).

Debugging Tools

- Typical debugging session
  - Set a breakpoint in your code just prior to where you think a problem is occurring (i.e. on the line just ahead of the one in an exception's stack trace).
  - Run the program in debug mode, and the program will stop just ahead of the crash

Debugging Tools

- Typical debugging session
  - A breakpoint set at line 80.
  - Run the program in debug mode, and the program will stop just ahead of the crash

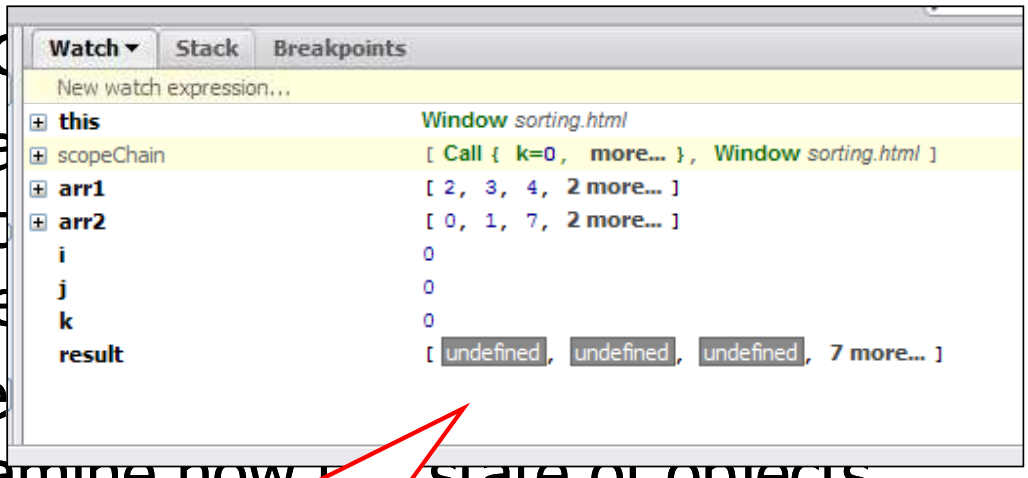


Debugging Tools

- Typical debugging session
  - Examine the variables at that point to determine what may have gone wrong. Use the watch or inspect features.
  - Step forward through the program to examine how the state of objects change as the program is executing line-by-line.

Debugging Tools

- Typic
  - Exa
    det
    Use
  - Ste
    exa



ram is executing

The watch window displays the names and values of the local variables in the function

Debugging Tools

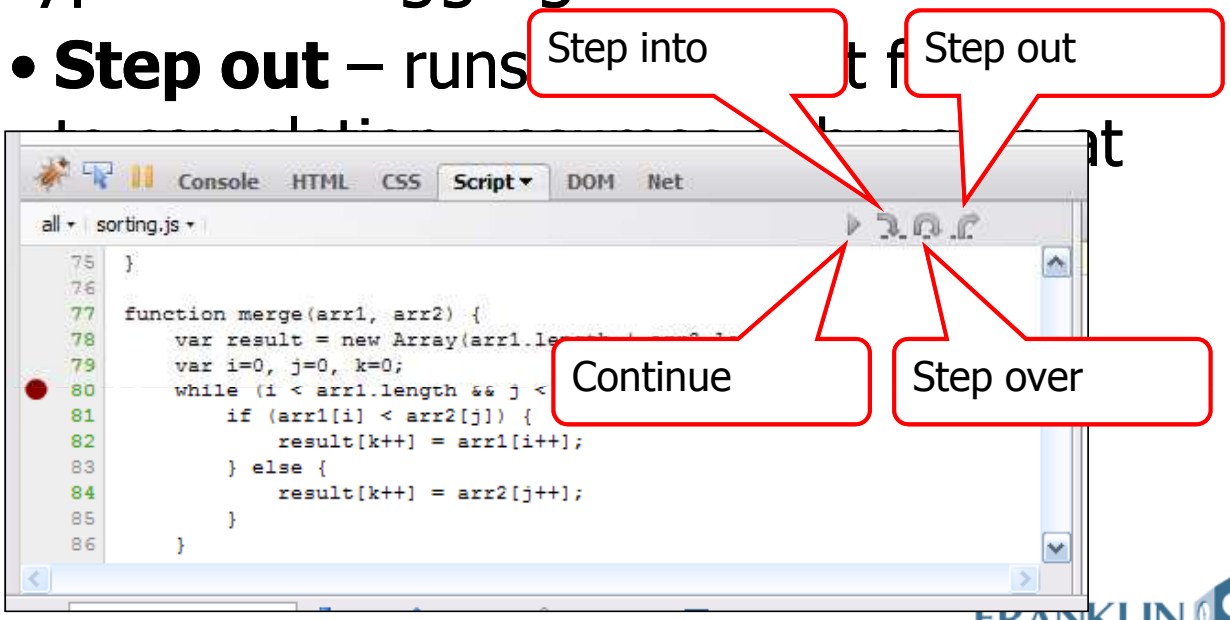
- Typical debugging session
  - **Step into** – if on a line with a function call, starts debugging that function, otherwise just executes the next line
  - **Step over** – if on a line with a function call, calls the function (but doesn't debug it), otherwise just executes the next line.

Debugging Tools

- Typical debugging session
  - **Step out** – runs the current function to completion, resumes debugging at the point at which the function was called.
  - **Continue** – runs to the next breakpoint.

Debugging Tools

- Typical debugging session
  - **Step out** – runs t f
    t



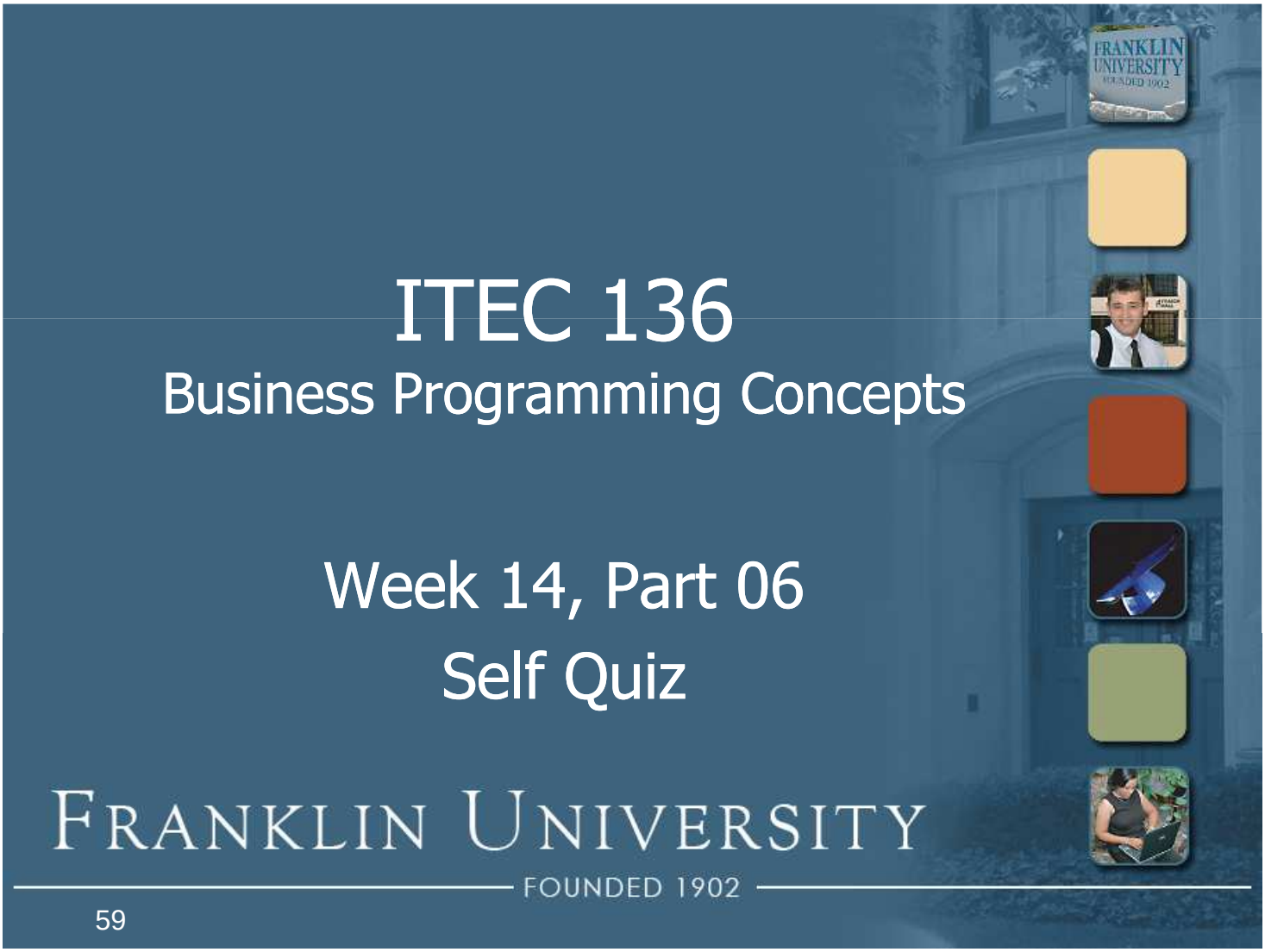
Debugging Tools

- Firebug demonstration
  - http://encytemedia.com/blog/articles/2006/05/12/an-in-depth-look-at-the-future-of-javascript-debugging-with-firebug
  - http://www.digitalmediaminute.com/screencast/firebug-js/

Questions?

Next Week

- Last class! ☺
- Final exam! ☹



Self Quiz

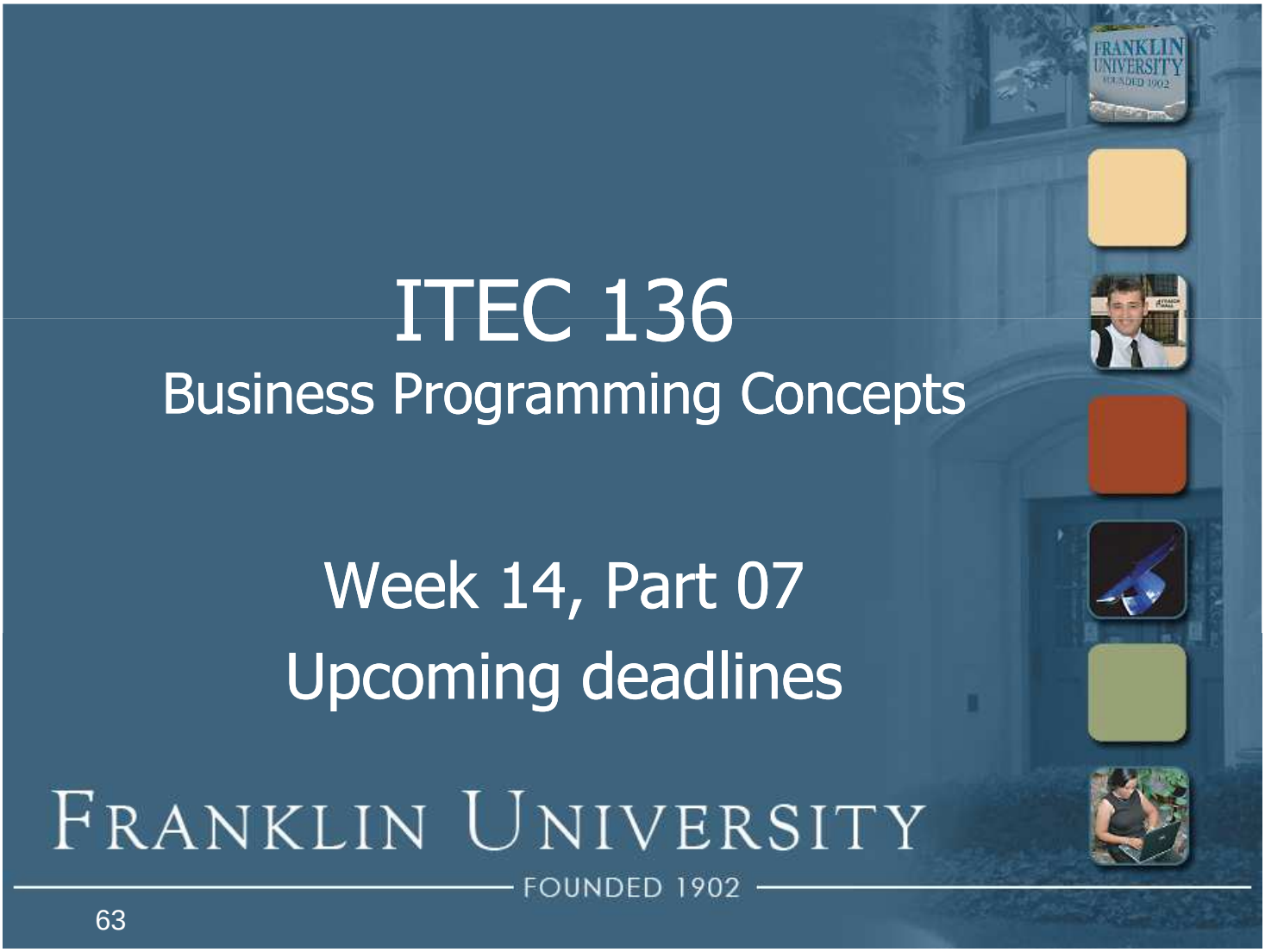
- What is the difference between validation and verification?
- Name the five different types of testing in the SDLC.
- Compare and contrast black-box and white-box testing.

Self Quiz

- What is the advantage of a unit testing framework over some ad-hoc approach?
- Write a function "`isSorted`" that returns true if the array given as a parameter is sorted, and false otherwise.

Self Quiz

- Write a thorough set of test cases for `isSorted`.
- Describe the process of debugging using a debugger.
- Describe the process of debugging using a logging facility.



Upcoming Deadlines

- Due April 13
  - Homework 12 (optional)
  - Lab 4
  - Reflection paper
  - Final exam