



# JAVA DAVE ENVIRONMENT

TDD, JUnit Testing, DHTML, DOM, JavaScript syntax &  
Maintainable JS

Java workshop training, August, 2017.

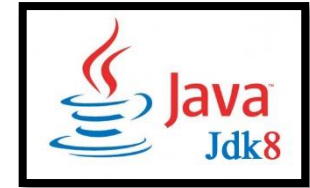
**Dr. Kishore Biswas (Forrest/柯修)**

PhD. Artificial Intelligence—Natural Language Processing.

CanWay IT Training ®.  
CSUMSA LL LLSIUDG ®

# CONTENTS

- 1) TDD & Writing testable codes.
- 2) JUnit Testing (Continued...)
- 3) DHTML and DOM
- 4) JavaScript syntax & Maintainable JS
- 5) Code practice

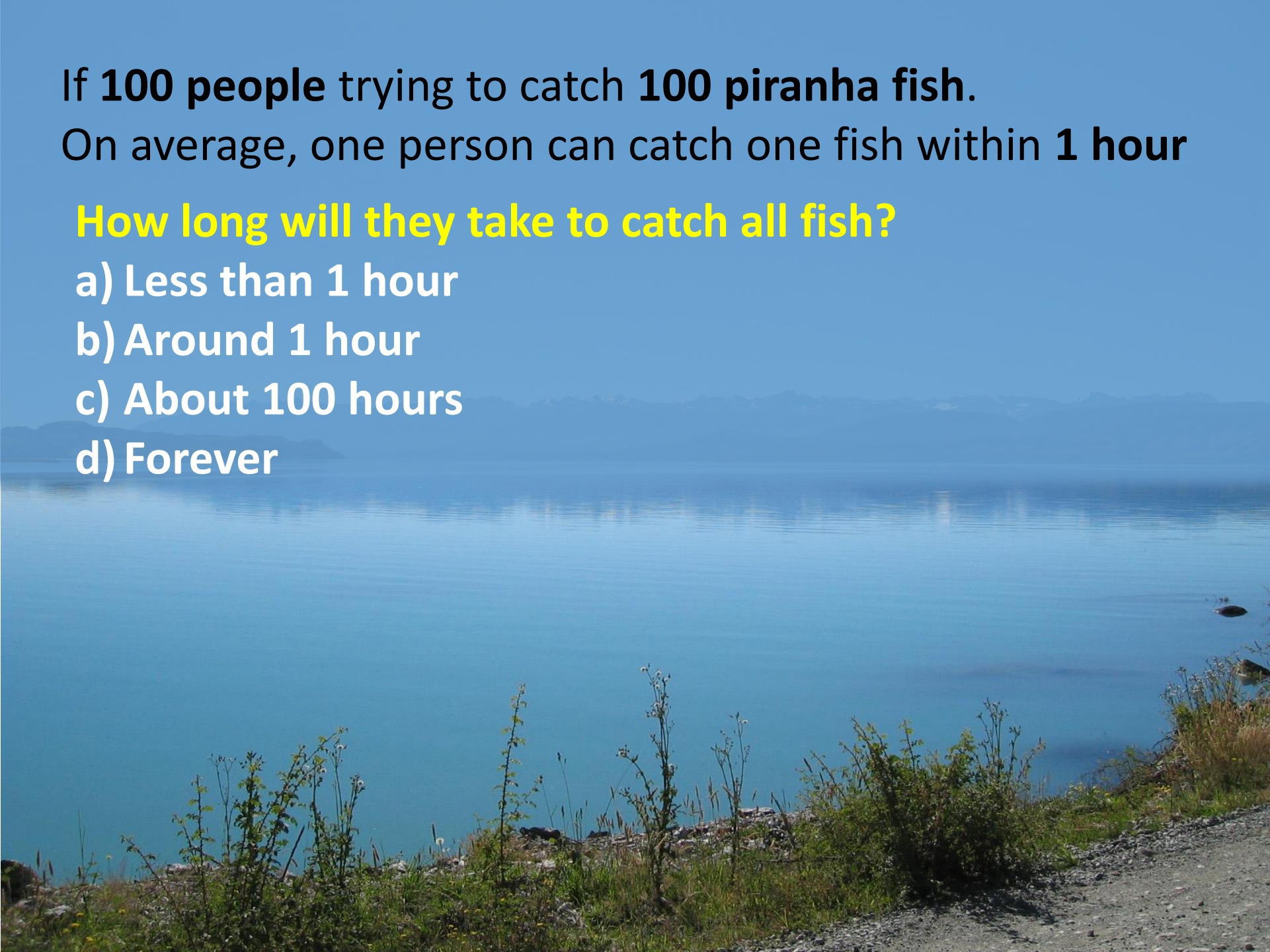


# 1) TDD & WRITING TESTABLE CODES.

If **100 people** trying to catch **100 piranha fish**.  
On average, one person can catch one fish within **1 hour**

**How long will they take to catch all fish?**

- a) Less than 1 hour
- b) Around 1 hour
- c) About 100 hours
- d) Forever



If **100 people** trying to catch **100 piranha fish**.  
On average, one person can catch one fish within **1 hour**

**How long will they take to catch all fish?**

- a) Less than 1 hour
- b) Around 1 hour
- c) About 100 hours
- d) Forever



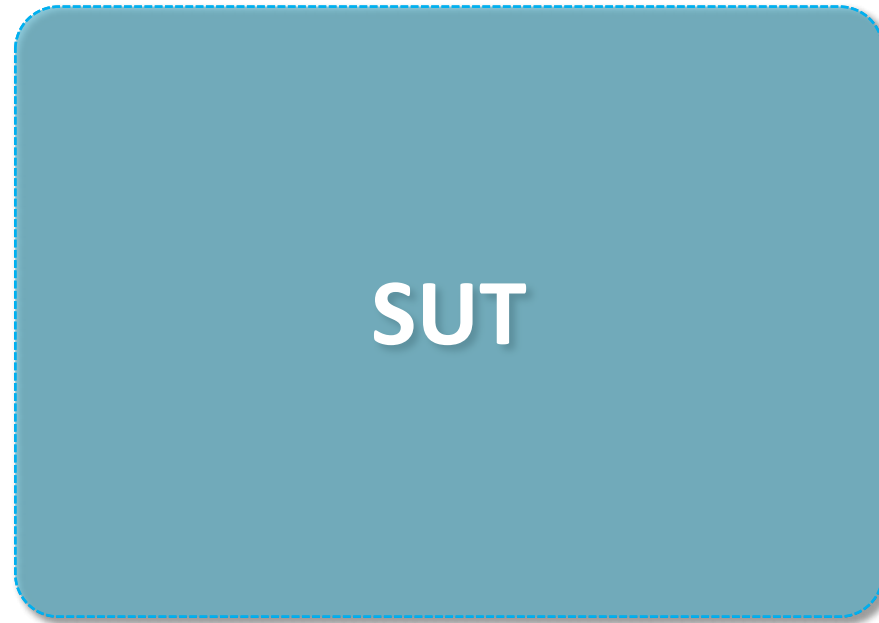
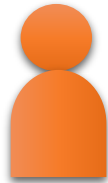
If **100 people** trying to catch **100 piranha fish**.  
On average, one person can catch one fish within **1 hour**

**How long will they take to catch all fish?**

- a) Less than 1 hour
- b) Around 1 hour
- c) About 100 hours
- d) Forever



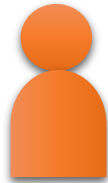
# SYSTEM TESTING



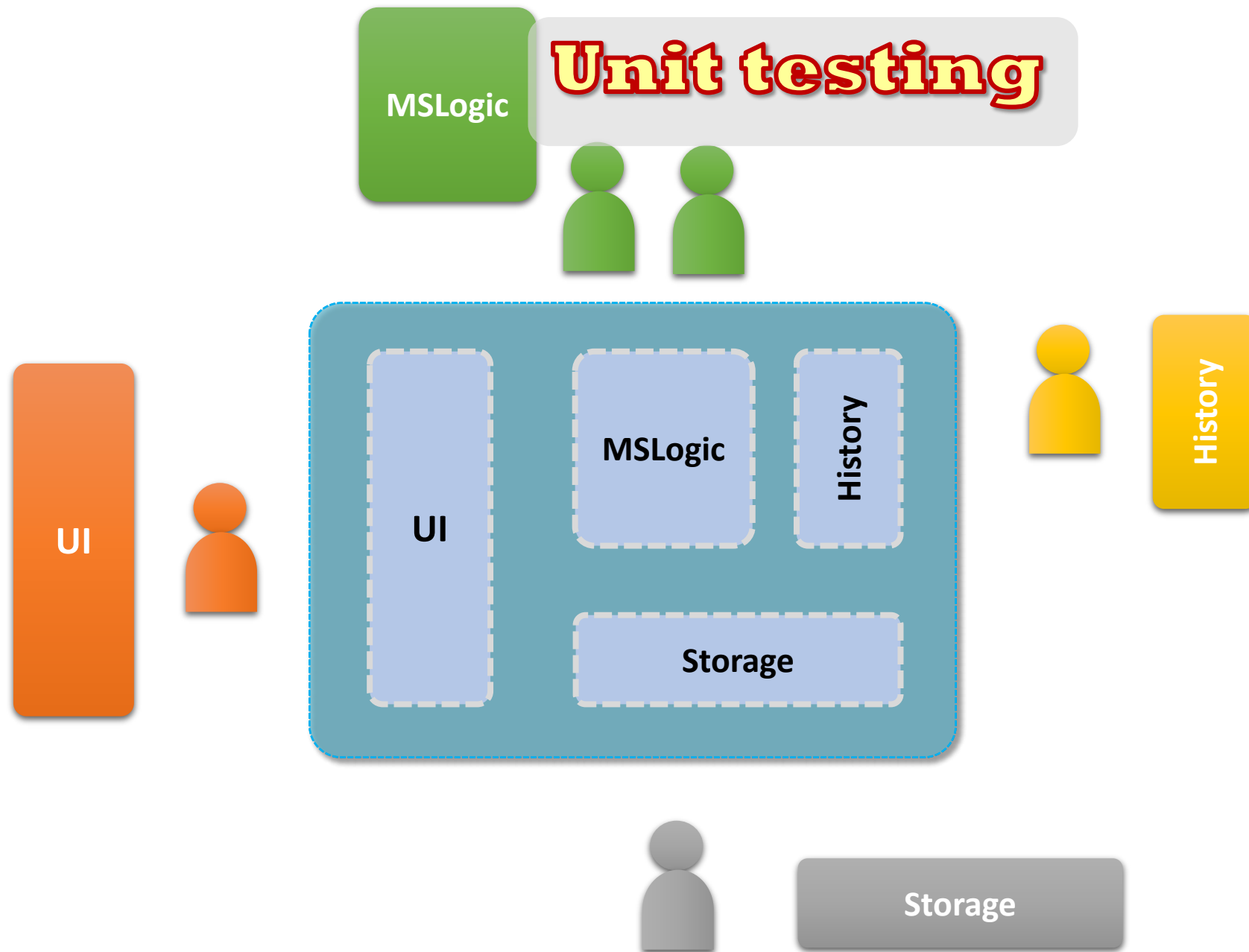
SUT

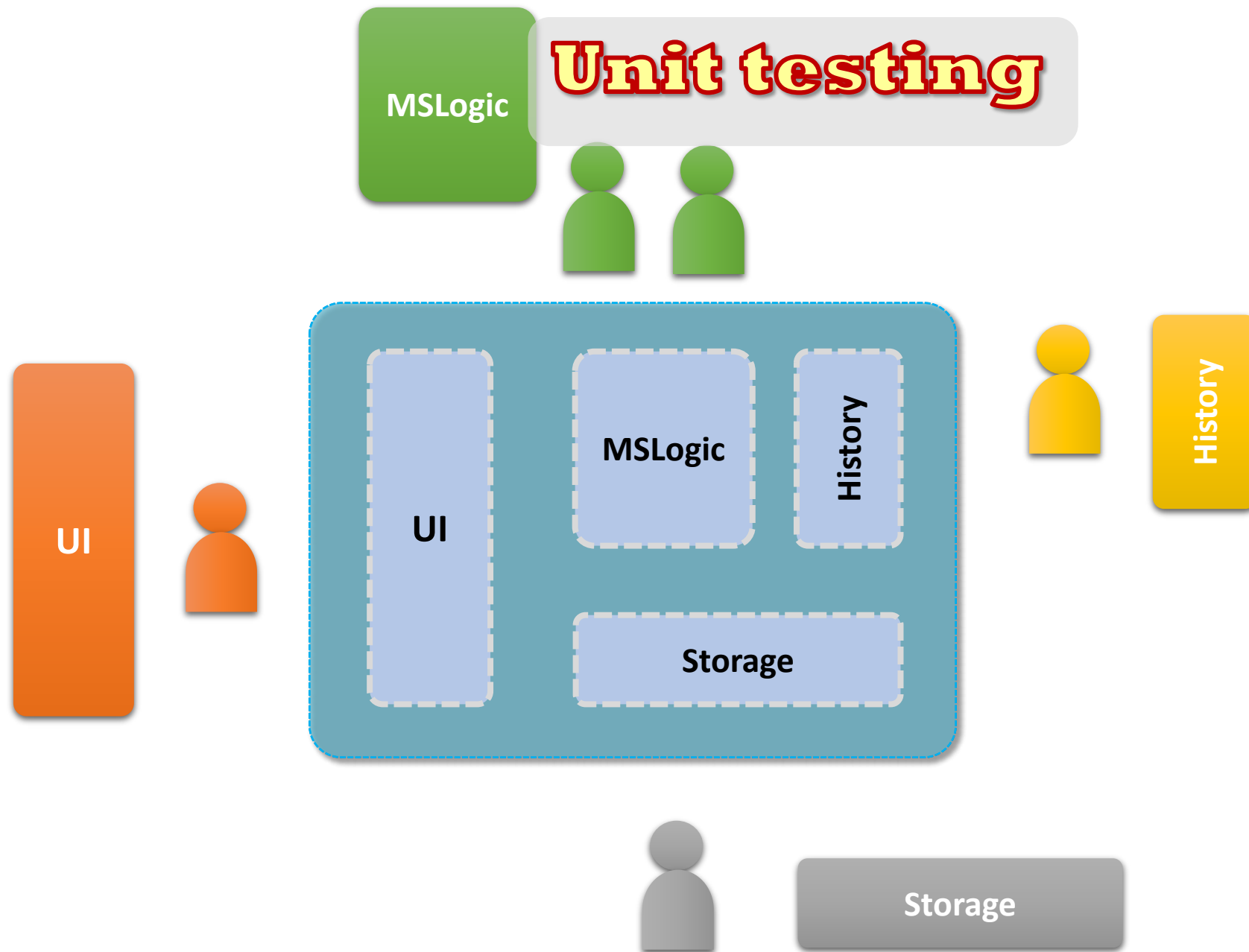


# SYSTEM TESTING

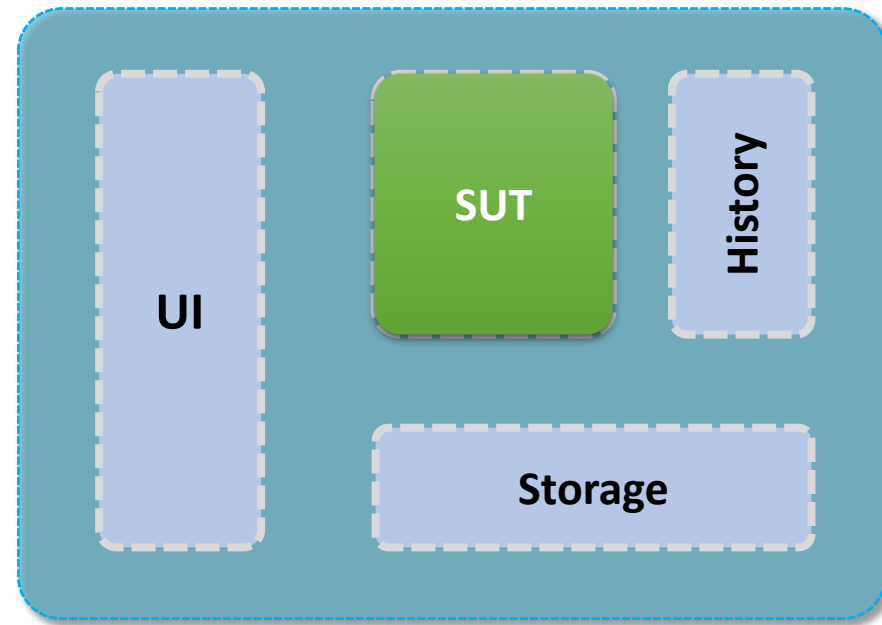


SUT

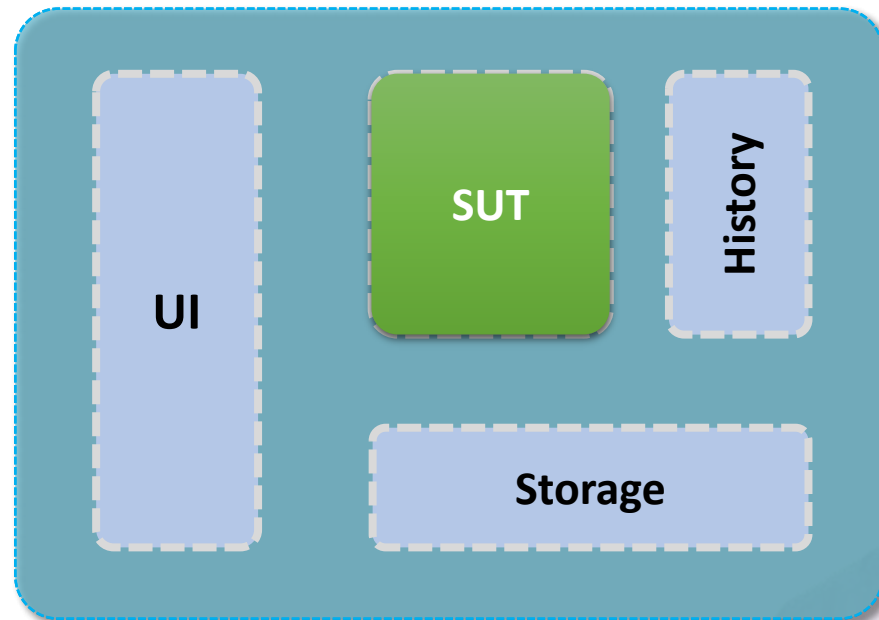




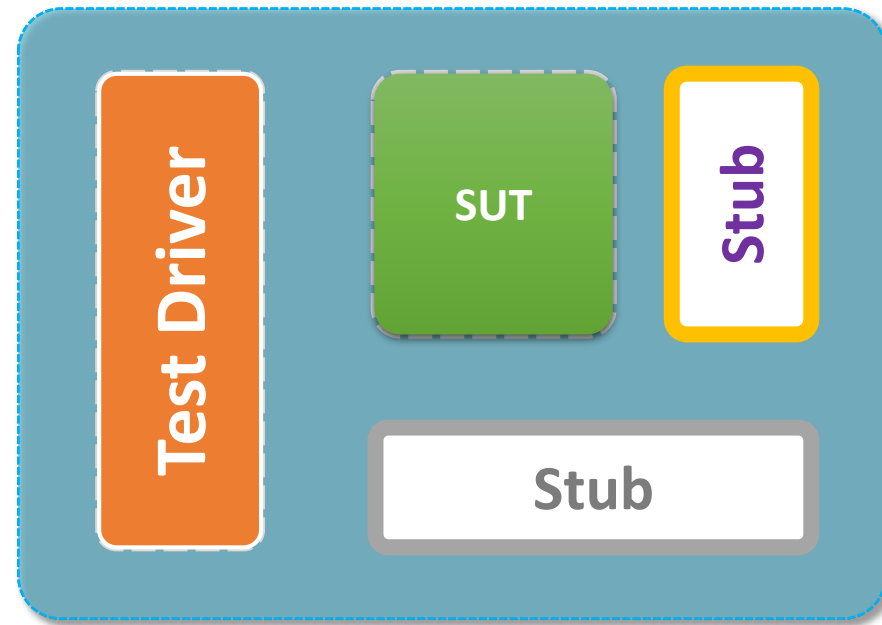
# Unit testing



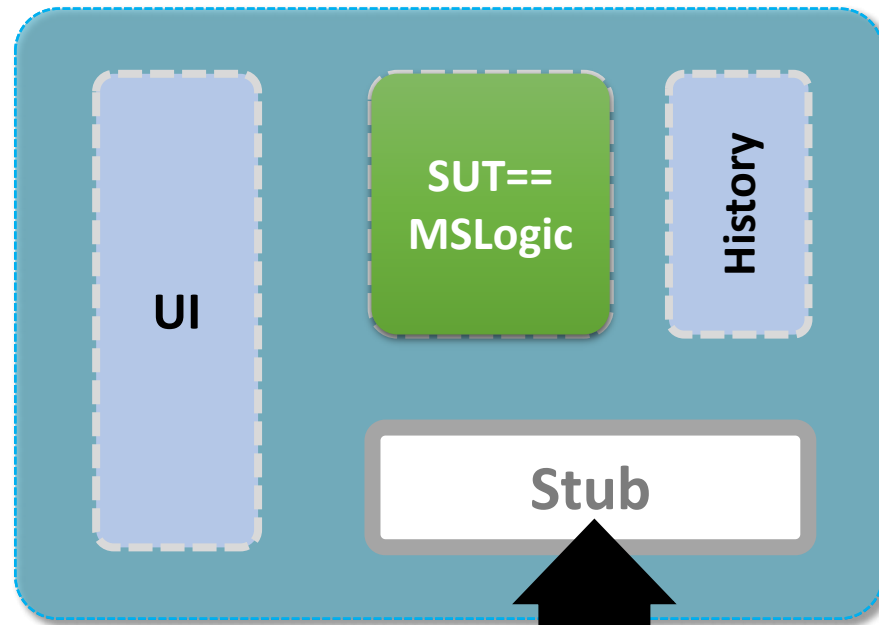
# Unit testing



# Unit testing

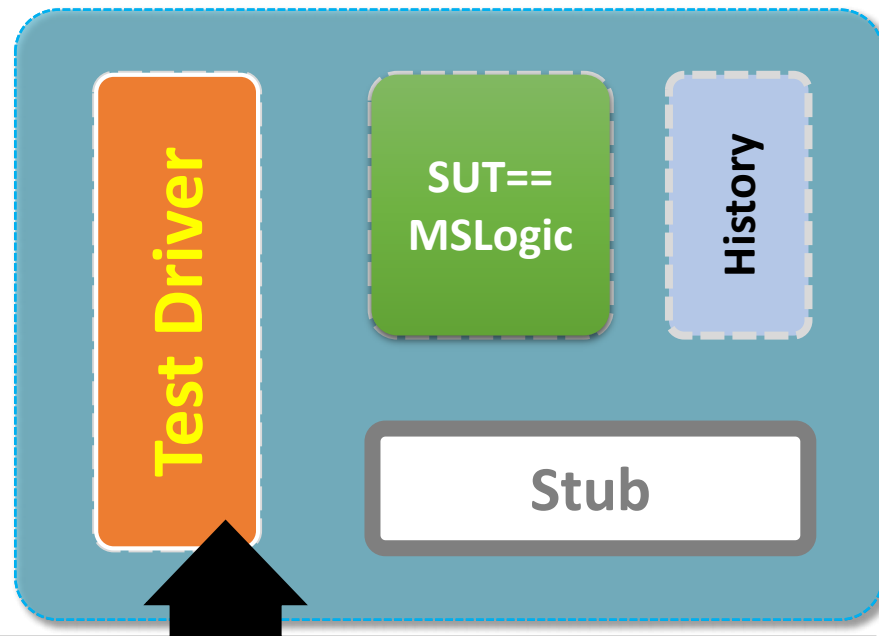


# Unit testing



```
String retrieveFromDatabase (int key){  
    if(key==1) return "Item for key 1";  
    if(key==2) return "Item for key 2";  
    ...  
}
```

# Unit testing

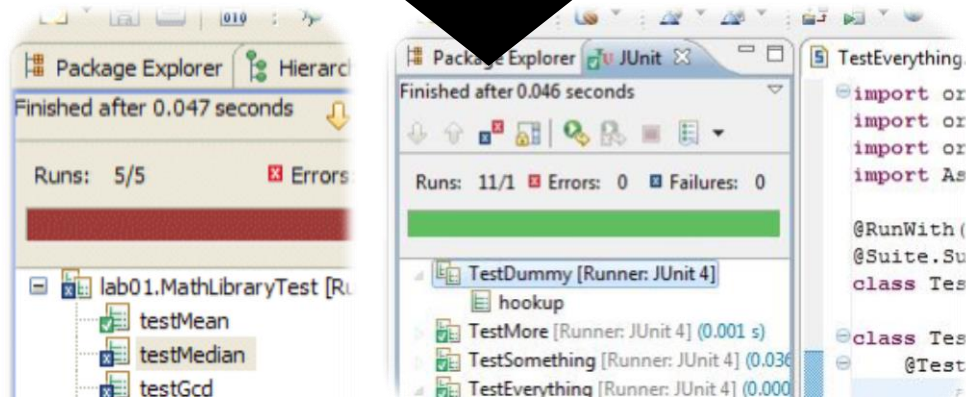


```
String item = msLogic.getItem(1);  
if(!item.equals("Item1") print("Case 1 failed");  
...
```

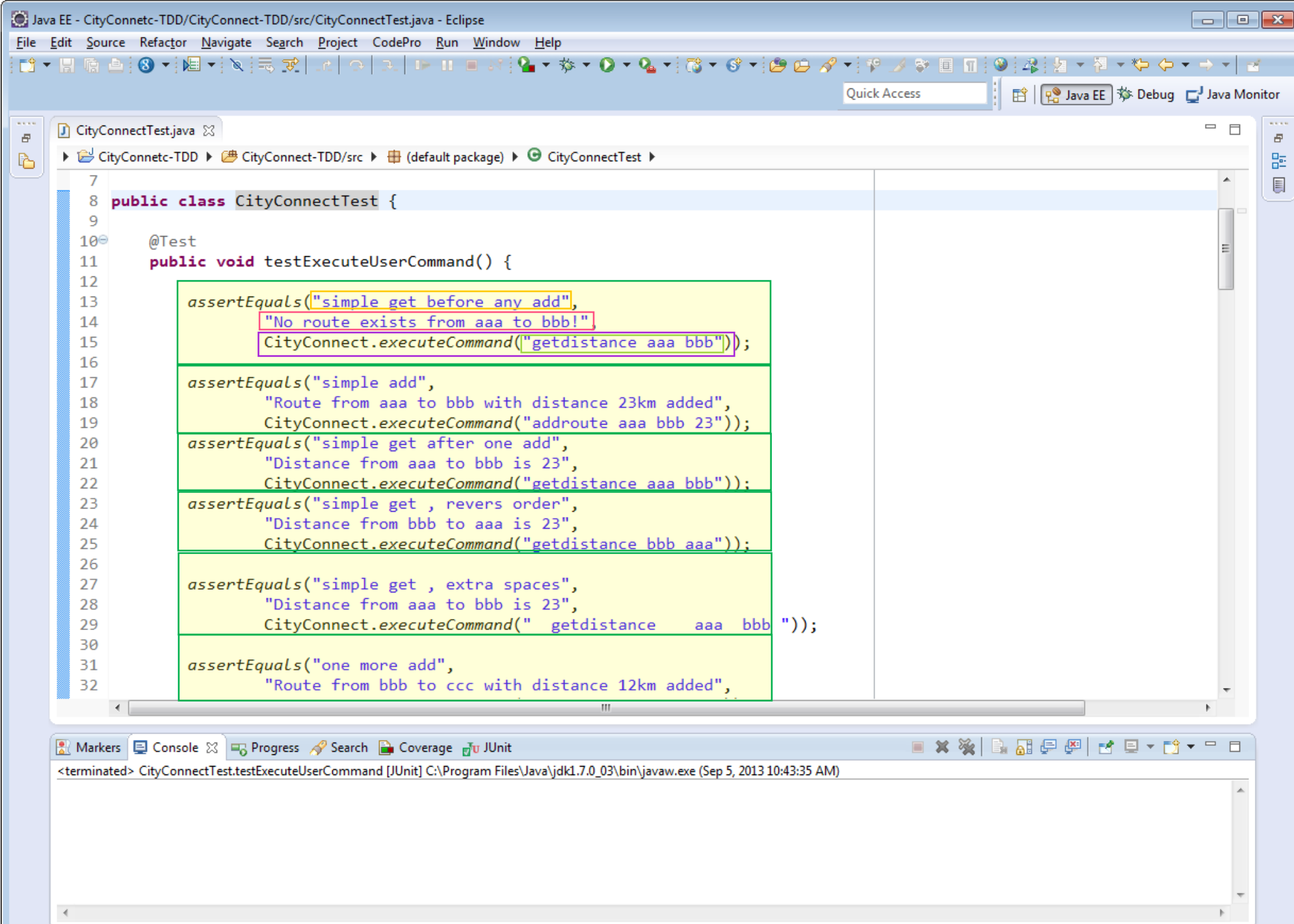
```
String item = msLogic.getItem(1);  
if(!item.equals("Item1") print("Case 1 failed");  
...
```

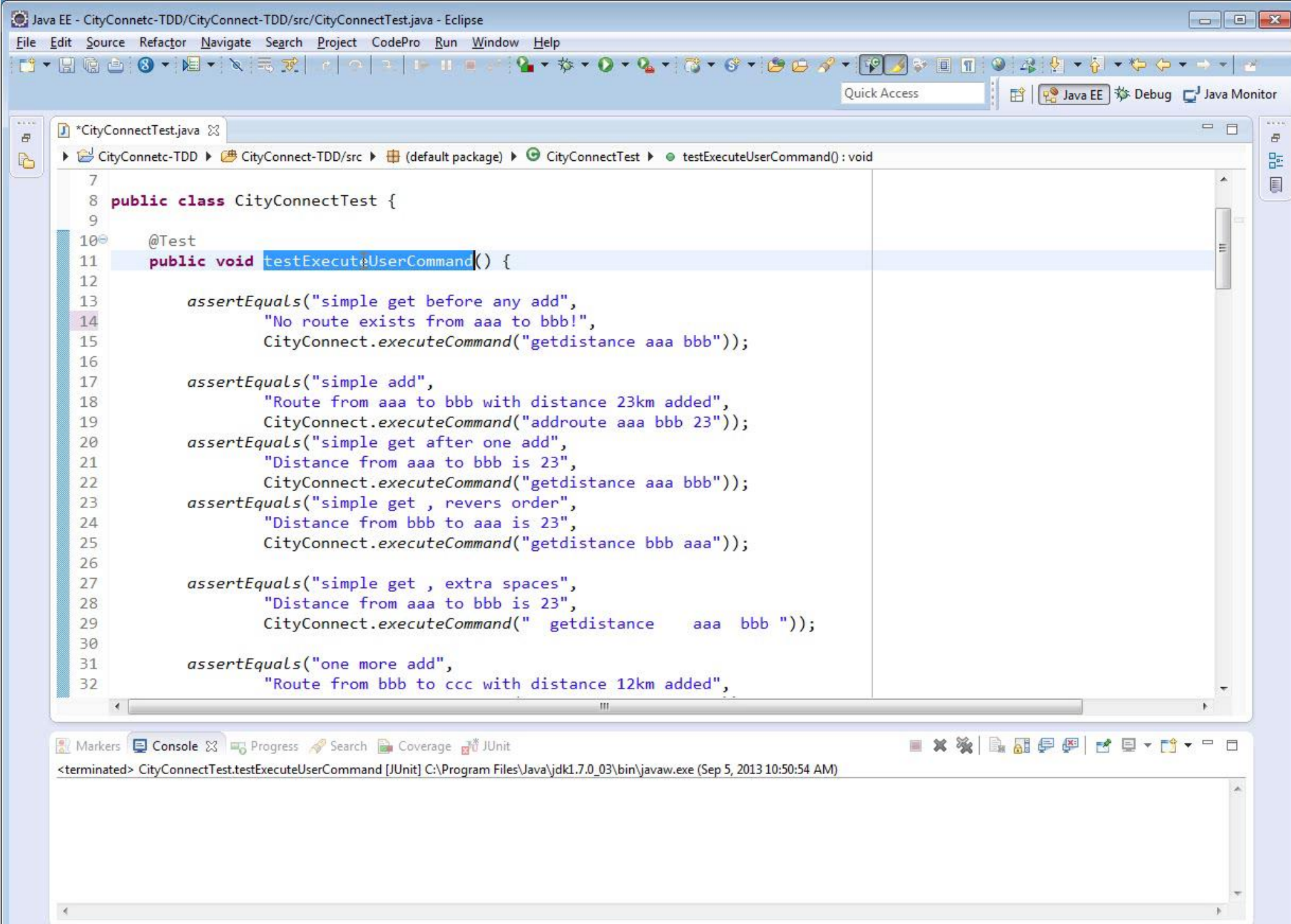
## Testing frameworks

msLogic.getItem(1) "Item1"

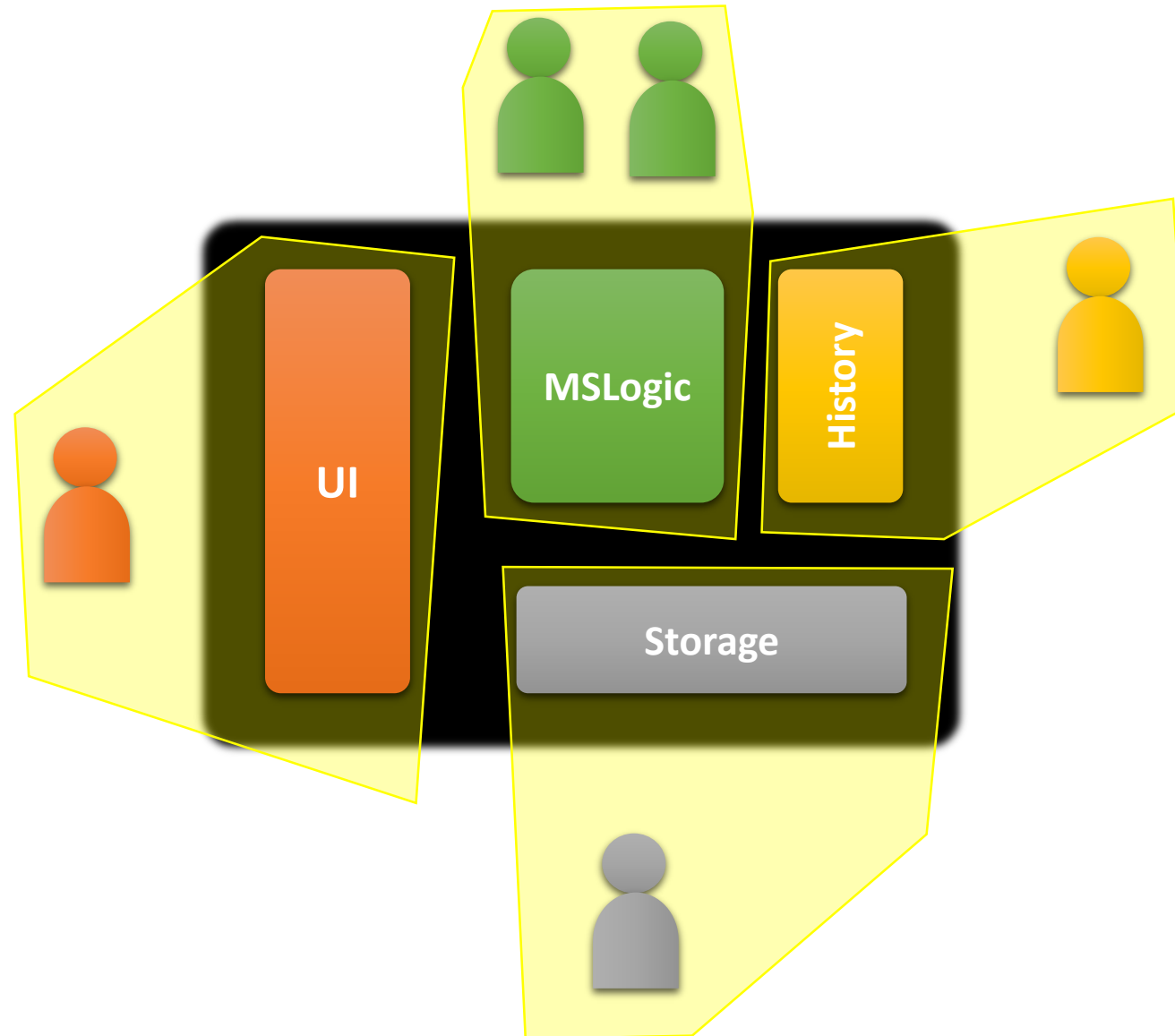


JUnit (Java)  
Visual Studio Native Tests (C++)

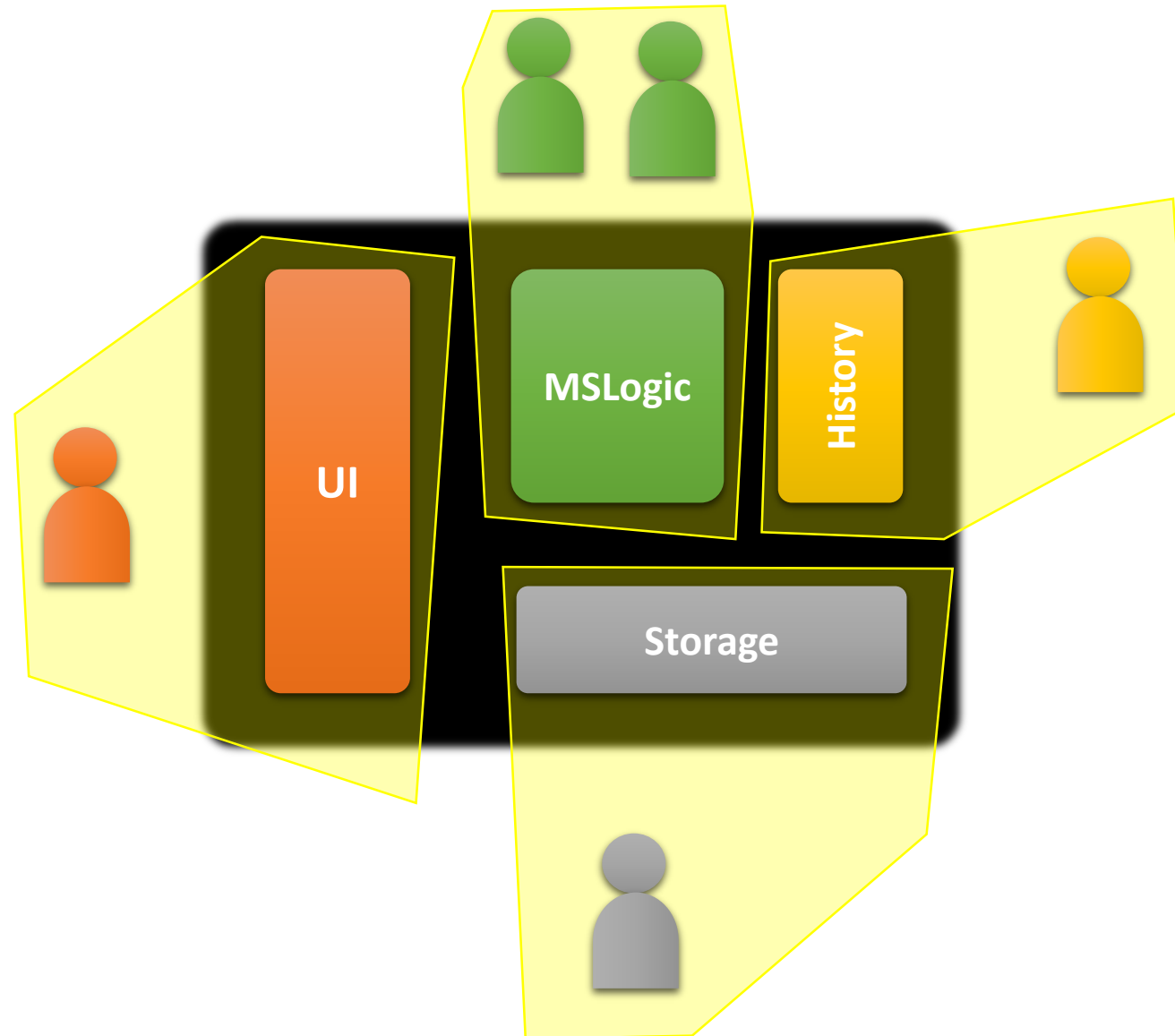




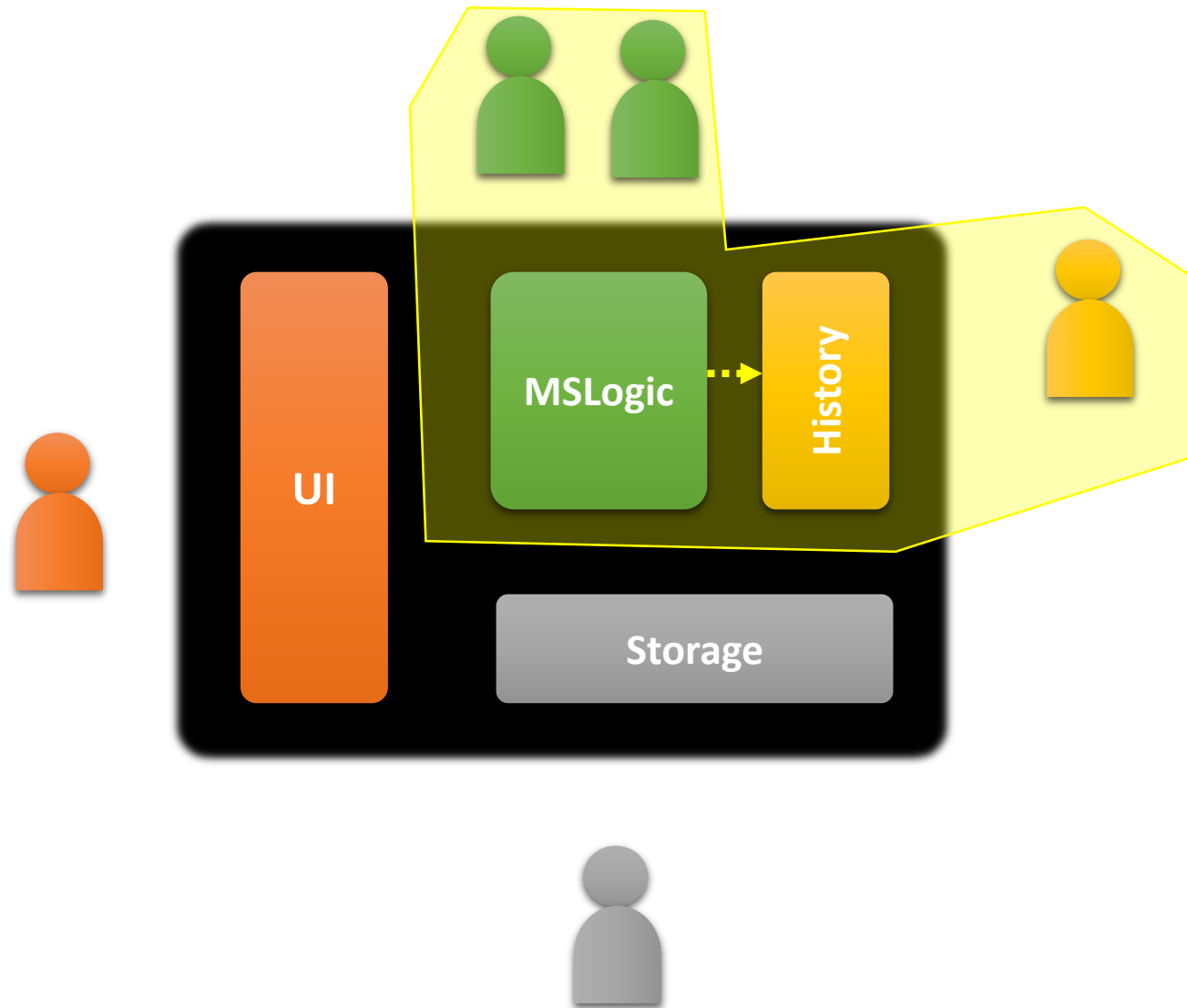
# Unit testing



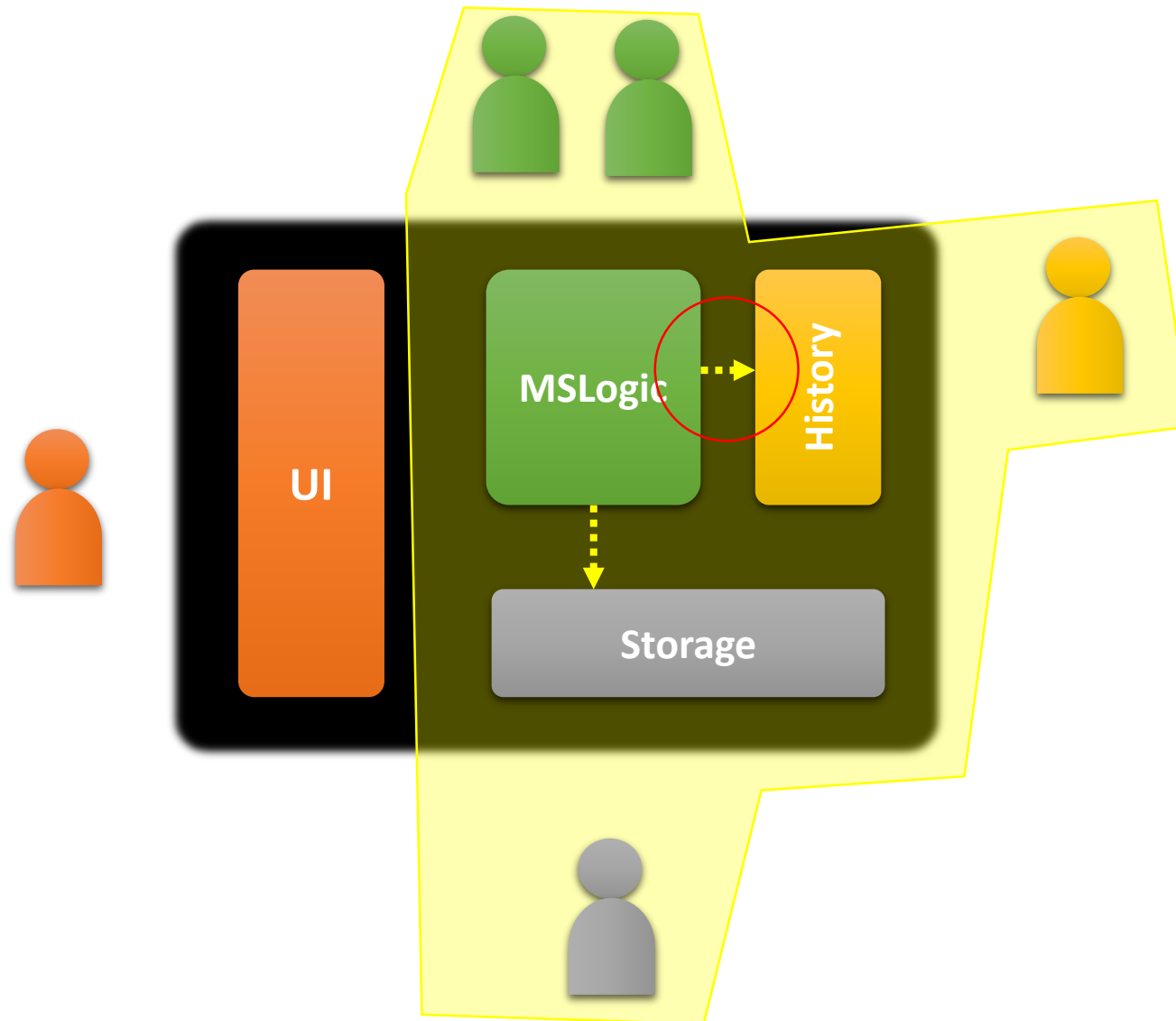
# Unit testing



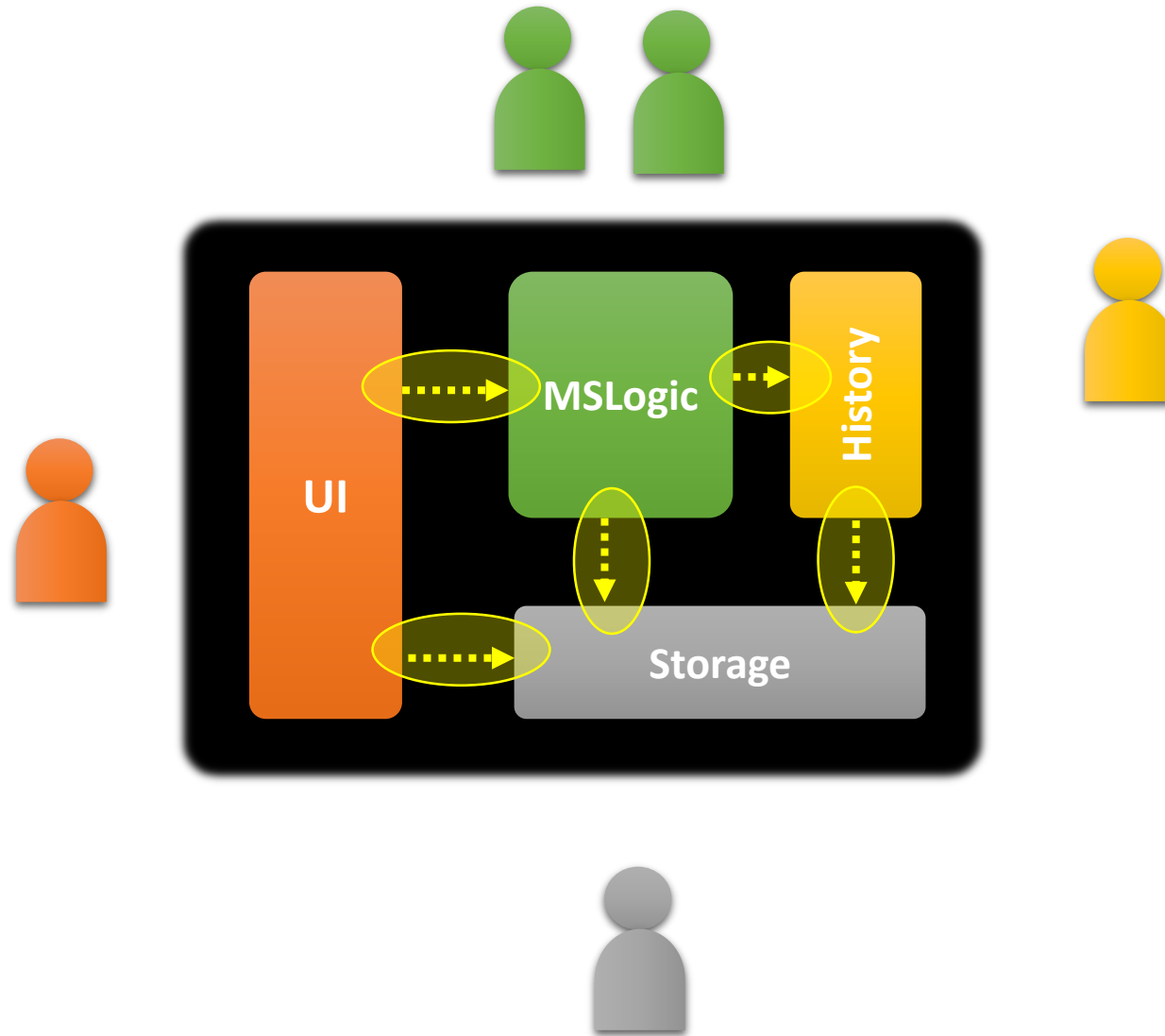
# Integration testing



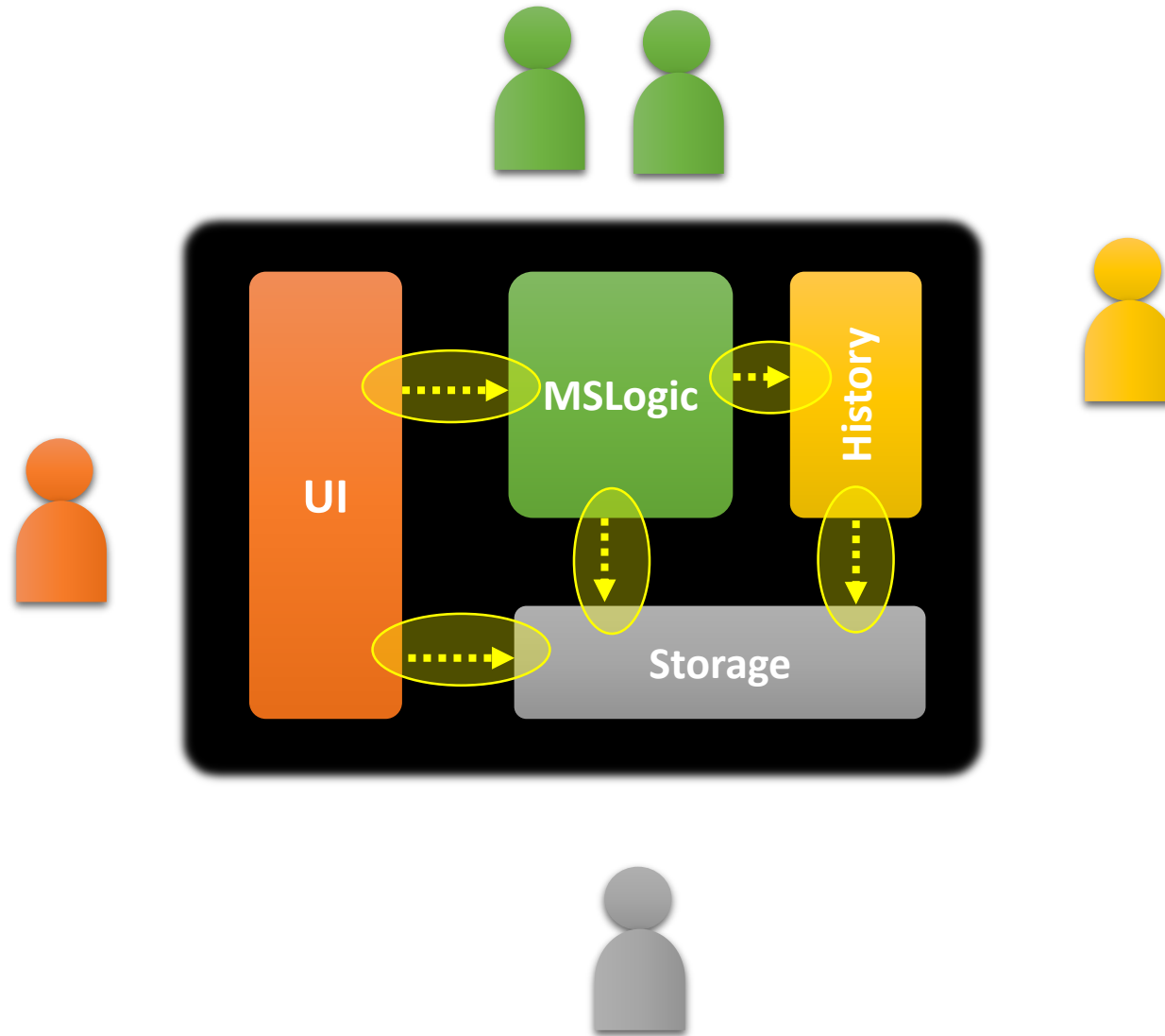
# Integration testing



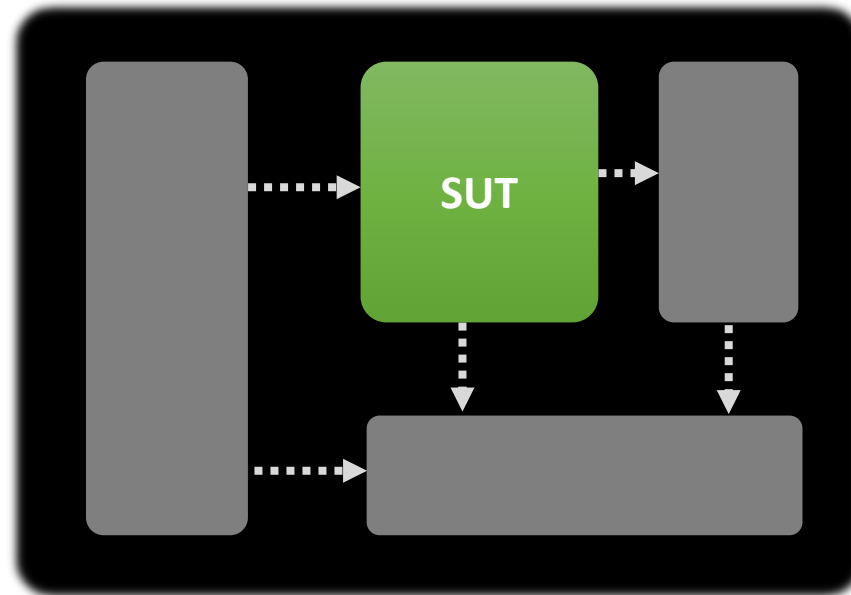
# Integration testing



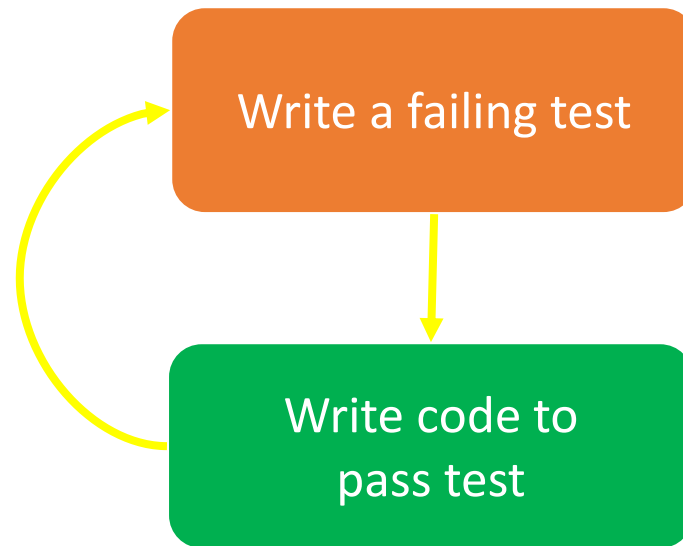
# Integration testing



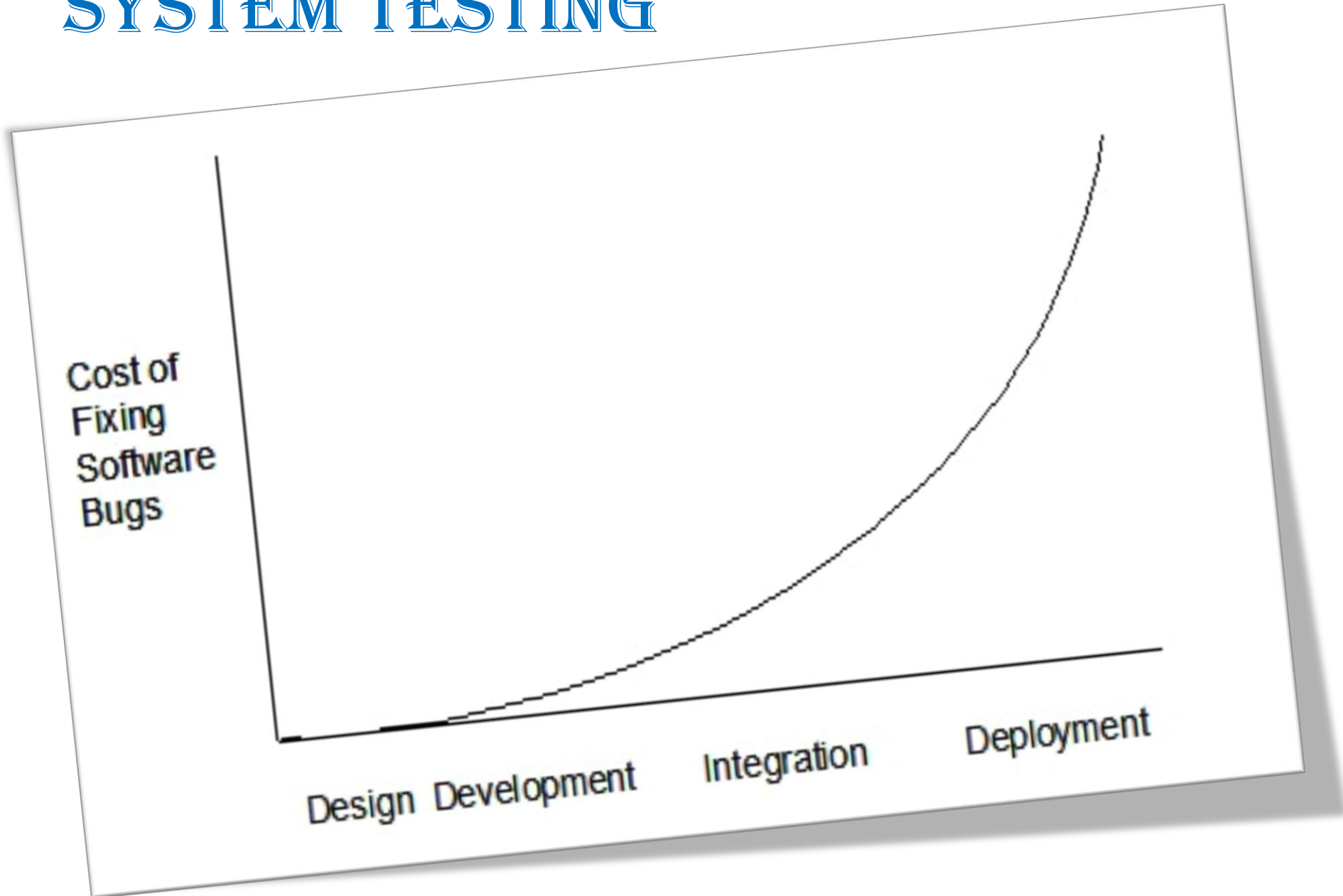
# Test-Driven Development (TDD)



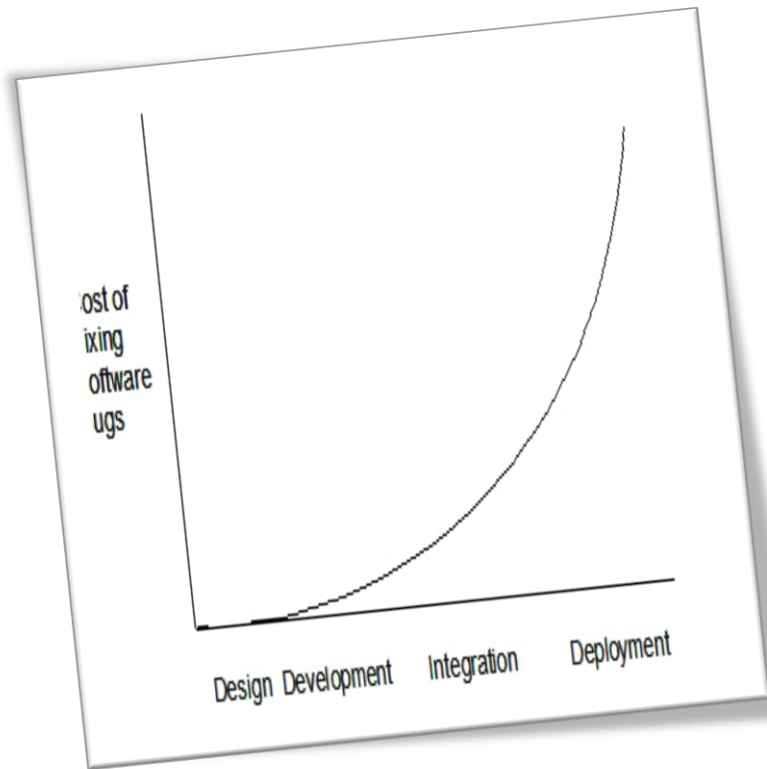
# Test-Driven Development (TDD)



# SYSTEM TESTING

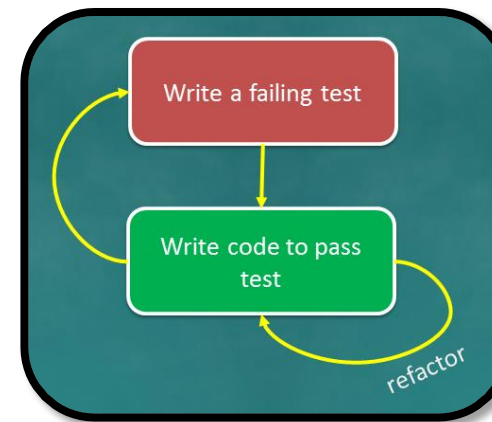
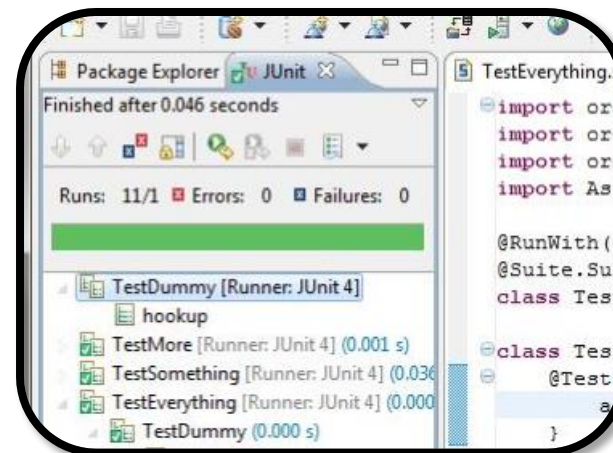
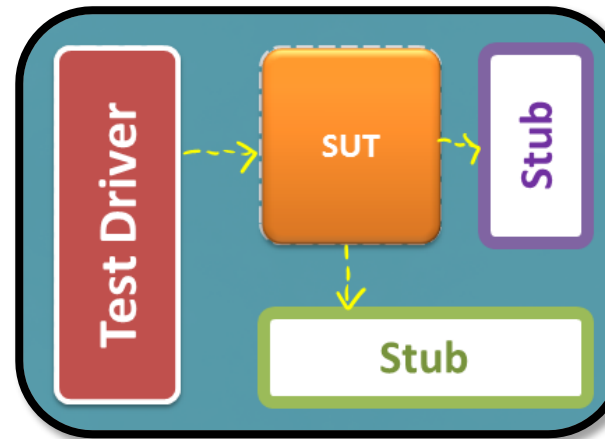
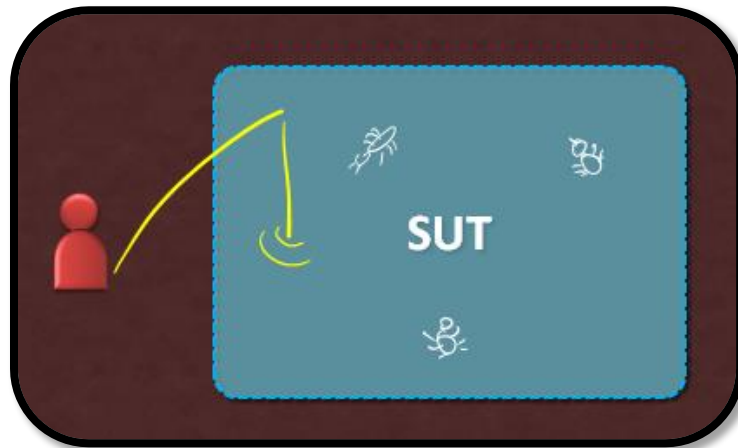


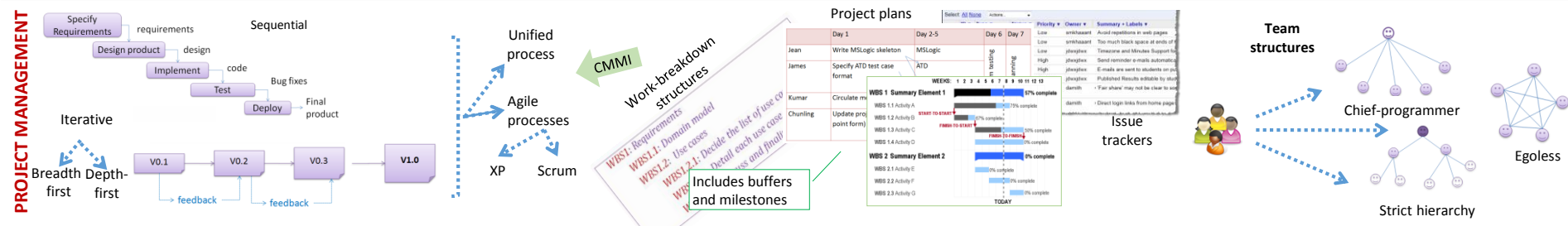
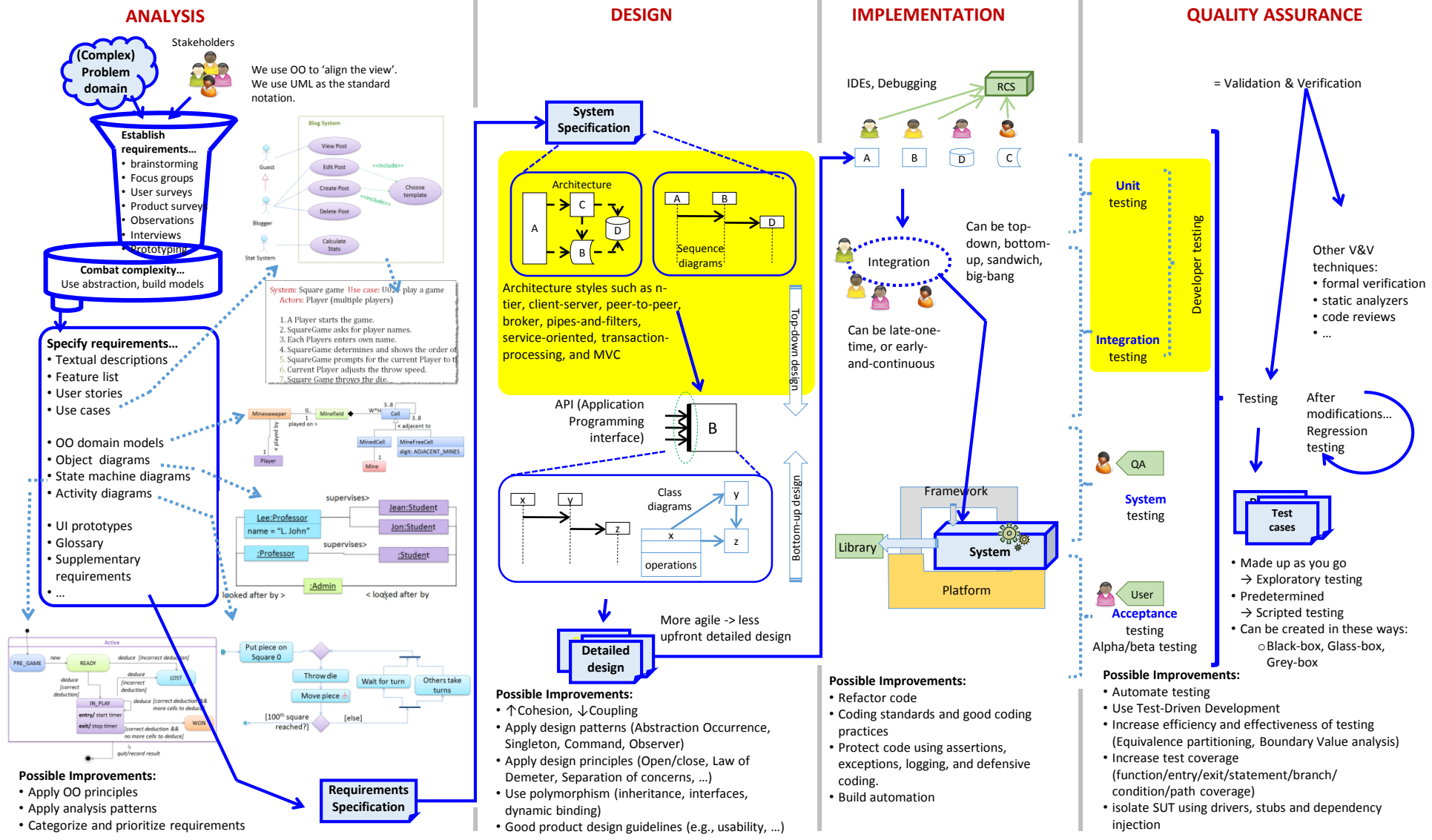
Do you want maximum publicity for your mistakes?



Added:  
desc: send budget  
deadline: Friday 23<sup>rd</sup> Sep

add send budget by Friday





# Test Driven Development (TDD)

"TDD is emerging as one of the most successful developer productivity enhancing techniques to be recently discovered. The three-step: write test, write code, refactor – is a dance many of us are enjoying"

— *Eric Vautier, David Vydra*

"TDD is the Gem of Extreme Programming"

— *Kent Beck*

# TDD: What is it?

- ◆ TDD turns traditional development around
  - Write test code before the methods
  - Do so in very small steps
  - Refuse to add any code until a test exists for it
    - Have a failing test (red bar) before writing the code to make the test pass
- ◆ You can run tests anytime
  - And you should do so quite often

# Simple, yet hard to do

- ◆ When you first try TDD, it requires great discipline because it is easy to “slip” and write methods without first writing a new test
- ◆ Pair programming helps you to stay on track
  - Williams and Kessler

# Comments, Documentation?

- ◆ Most programmers don't read documentation
  - instead they prefer to work with the code
- ◆ Programmers often look for sample code that sends messages to instances of a class
  - Well-written unit tests can provide such a working specification of your code
    - more docs are necessary
  - Unit tests effectively become a significant portion of your technical documentation

# Testing Framework

- ◆ An underlying assumption of TDD is that you have a unit-testing framework
- ◆ Good software developers use a testing framework such as JUnit
  - Many exist [http://en.wikipedia.org/wiki/List\\_of\\_unit\\_testing\\_frameworks](http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks)
- ◆ Without a unit-testing framework, TDD is virtually impossible

# TDD Helps Design

- ◆ TDD makes you think about what you want or need before you write the code
  - pick class and method names
  - decide the needed arguments
  - choose return types
  - write the class, constructors, instance variables, and methods to make what you want work

# Benefits of TDD

- ◆ Benefits include
  - Rapid feedback
  - Interface before algorithm
  - Know when you're finished
  - Change the code with confidence

# Principles of TDD

- ◆ You maintain an exhaustive suite of Programmer Tests
- ◆ No code goes into production unless it has associated tests
- ◆ You write the tests first
- ◆ Tests determine what code you need to write if you are programming by intention
  - Then you are designing as you write tests

# A unit test with 2 test methods

```
import static org.junit.Assert.*;
import org.junit.Test;

public class BankAccountTest {

    @Test
    public void testDeposit() {
        BankAccount anAcct = new BankAccount("Zac", 1000.00);
        assertEquals(1000.00, anAcct.getBalance(), 1e-12);
        anAcct.deposit(0.52);
        assertEquals(1000.52, anAcct.getBalance(), 1e-12);
    }

    @Test
    public void testWithdraw() {
        BankAccount anAcct = new BankAccount("Zac", 1000.52);
        assertEquals(1000.52, anAcct.getBalance(), 1e-12);
        anAcct.withdraw(10.52);
        assertEquals(990.00, anAcct.getBalance(), 1e-12);
    }
}
```

# Annotations

- ◆ Annotations provide data about a program that is not part of the program itself. Uses include
    - Information for the compiler
      - Annotations can be used by the compiler to detect errors or suppress warnings
- ```
@SuppressWarnings ("unused")
```
- Compiler-time and deployment-time processing
    - Software tools can process annotation information to generate code, XML files
  - Runtime processing
    - examined at runtime `@Test`

# JUnit Annotations

- ◆ Useful annotations used by JUnit
  - `@Test` marks a test method
  - `@Before` and `@After` mark methods that execute before and after every test method in the unit test
  - `@BeforeClass` marks a method that executes exactly once before any other annotated method in the unit test

```
import org.junit.*;

public class ShowFlowOfControl {

    @BeforeClass
    public static void beforeAllOthers() {
        System.out.println("First one time");
    }

    @Before
    public void beforeEachTest() {
        System.out.println("Before");
    }

    @After
    public void afterEachTest() {
        System.out.println("After");
    }

    @Test
    public void test1() {
        System.out.println("one");
    }

    @Test
    public void test2() {
        System.out.println("two");
    }
}
```

What is the  
output?

# Assertions

## ◆ Useful assertions that go into `@Test` methods

```
// Asserts that a condition is true
public static void assertTrue(boolean condition)

public static void assertEquals(int expected, int actual)

public static void assertEquals (double expected,
                                double actual,
                                double err)

// Asserts that two Object objects are equal
public static void assertEquals(Object expected,
                                Object actual)
```

- ◆ All assertions of `org.junit.Assert` can be found at <http://www.junit.org/apidocs/org/junit/Assert.html>

# So why test code?

- ◆ So we develop quality software that works
- ◆ It can get you jobs
- ◆ It's what good software developers do
- ◆ It is very important to test your code
- ◆ You can develop code in half the time
- ◆ Gives you the power and confidence to refactor
- ◆ To save lives and help avoid financial losses
- ◆ Rick says you have to

## 2) LEARNING TESTING TOOL: JUNIT & MOCKITO FOR JAVA

# BUGS AND TESTING



- **software reliability:** Probability that a software system will not cause failure under specified conditions.
  - Measured by uptime, MTTF (mean time till failure), crash data.
- **Bugs** are inevitable in any complex software system.
  - Industry estimates: 10-50 bugs per 1000 lines of code.
  - A bug can be visible or can hide in your code until much later.
- **testing:** A systematic attempt to reveal errors.
  - Failed test: an error was demonstrated.
  - Passed test: no error was found (for this particular situation).

# DIFFICULTIES OF TESTING

- Perception by some developers and managers:
  - Testing is seen as a novice's job.
  - Assigned to the least experienced team members.
  - Done as an afterthought (if at all).
    - "My code is good; it won't have bugs. I don't need to test it."
    - "I'll just find the bugs by running the client program."
- Limitations of what testing can show you:
  - It is impossible to completely test a system.
  - Testing does not always directly reveal the actual bugs in the code.
  - Testing does not prove the absence of errors in software.

# UNIT TESTING

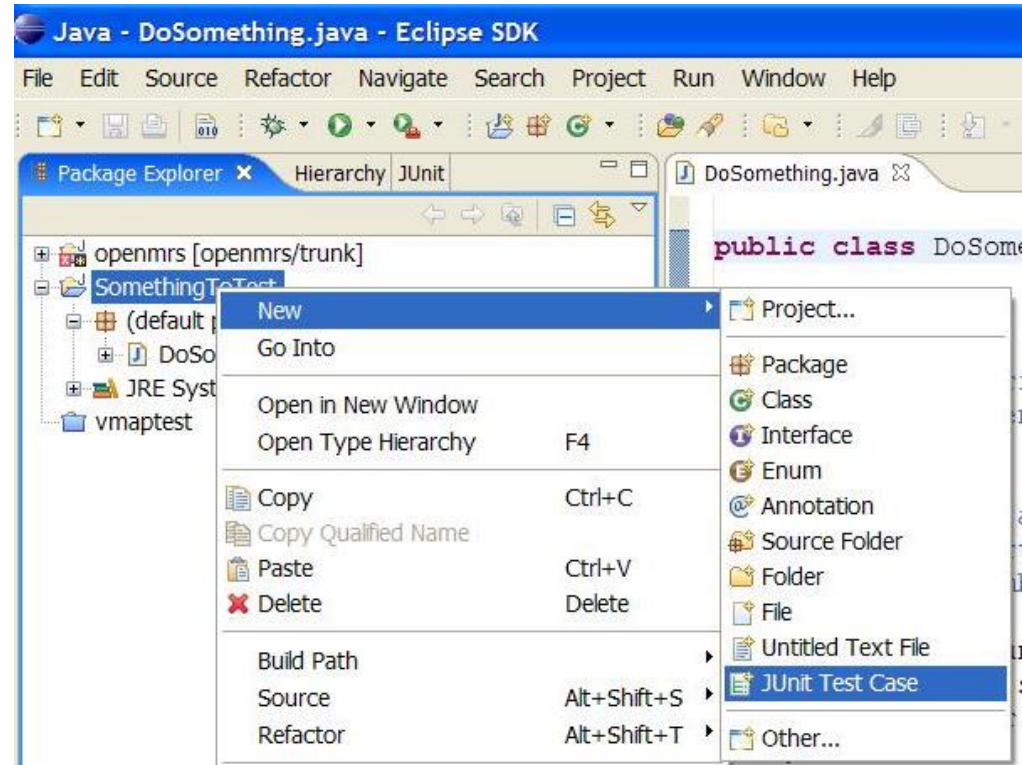


- **unit testing:** Looking for errors in a subsystem in isolation.
  - Generally a "subsystem" means a particular class or object.
  - The Java library **JUnit** helps us to easily perform unit testing.
- The basic idea:
  - For a given class `Foo`, create another class `FooTest` to test it, containing various "test case" methods to run.
  - Each method looks for particular results and passes / fails.
- JUnit provides "**assert**" commands to help us write tests.
  - The idea: Put assertion calls in your test methods to check things you expect to be true. If they aren't, the test will fail.

# JUNIT AND ECLIPSE

- To add JUnit to an Eclipse project, click:
  - **Project → Properties → Build Path → Libraries → Add Library... → JUnit → JUnit 4 → Finish**

- To create a test case:
  - right-click a file and choose **New → Test Case**
  - or click **File → New → JUnit Test Case**
- Eclipse can create stubs of method tests for you.



# A JUNIT TEST CLASS

```
import org.junit.*;
import static org.junit.Assert.*;

public class name {
    ...

    @Test
    public void name() { // a test case
method
        ...
    }
}
```

- A method with `@Test` is flagged as a JUnit test case.
  - All `@Test` methods run when JUnit runs your test class.

# JUNIT ASSERTION METHODS

|                                                            |                                                               |
|------------------------------------------------------------|---------------------------------------------------------------|
| <code>assertTrue(<b>test</b>)</code>                       | fails if the boolean test is <code>false</code>               |
| <code>assertFalse(<b>test</b>)</code>                      | fails if the boolean test is <code>true</code>                |
| <code>assertEquals(<b>expected</b>, <b>actual</b>)</code>  | fails if the values are not equal                             |
| <code>assertSame(<b>expected</b>, <b>actual</b>)</code>    | fails if the values are not the same (by <code>==</code> )    |
| <code>assertNotSame(<b>expected</b>, <b>actual</b>)</code> | fails if the values <i>are</i> the same (by <code>==</code> ) |
| <code>assertNull(<b>value</b>)</code>                      | fails if the given value is <i>not</i> <code>null</code>      |
| <code>assertNotNull(<b>value</b>)</code>                   | fails if the given value is <code>null</code>                 |
| <code>fail()</code>                                        | causes current test to immediately fail                       |

- Each method can also be passed a string to display if it fails:
  - e.g. `assertEquals("message", expected, actual)`
- Why is there no `pass` method?

# ARRAYINTLIST JUNIT TEST

```
import org.junit.*;
import static org.junit.Assert.*;

public class TestArrayIntList {
    @Test
    public void testAddGet1() {
        ArrayIntList list = new ArrayIntList();
        list.add(42);
        list.add(-3);
        list.add(15);
        assertEquals(42, list.get(0));
        assertEquals(-3, list.get(1));
        assertEquals(15, list.get(2));
    }

    @Test
    public void testIsEmpty() {
        ArrayIntList list = new ArrayIntList();
        assertTrue(list.isEmpty());
        list.add(123);
        assertFalse(list.isEmpty());
        list.remove(0);
        assertTrue(list.isEmpty());
    }
    ...
}
```

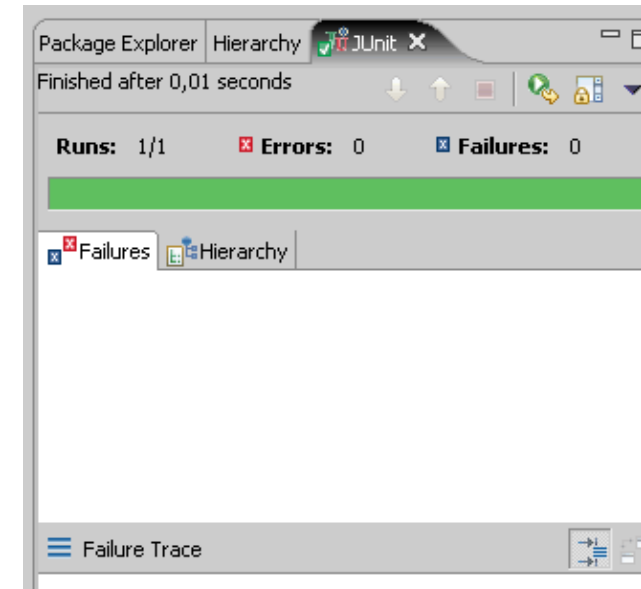
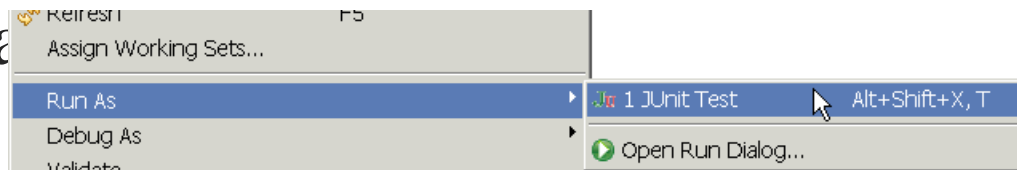
# RUNNING A TEST

- Right click it in the Eclipse Package Explorer at left; choose:

**Run As → JUnit Test**

- The JUnit bar will show **green** if all tests pass, **red** if any fail.

- The Failure Trace shows which tests fail



# JUNIT EXERCISE

Given a Date class with the following methods:

- `public Date(int year, int month, int day)`
- `public Date()` *// today*
- `public int getDay(), getMonth(), getYear()`
- `public void addDays(int days)` *// advances by days*
- `public int daysInMonth()`
- `public String dayOfWeek()` *// e.g. "Sunday"*
- `public boolean equals(Object o)`
- `public boolean isLeapYear()`
- `public void nextDay()` *// advances by 1 day*
- `public String toString()`

- Come up with unit tests to check the following:
  - That no Date object can ever get into an invalid state.
  - That the addDays method works properly.
    - It should be efficient enough to add 1,000,000 days in a call.

# WHAT'S WRONG WITH THIS?

```
public class DateTest {  
    @Test  
    public void test1() {  
        Date d = new Date(2050, 2, 15);  
        d.addDays(4);  
        assertEquals(d.getYear(), 2050);  
        assertEquals(d.getMonth(), 2);  
        assertEquals(d.getDay(), 19);  
    }  
  
    @Test  
    public void test2() {  
        Date d = new Date(2050, 2, 15);  
        d.addDays(14);  
        assertEquals(d.getYear(), 2050);  
        assertEquals(d.getMonth(), 3);  
        assertEquals(d.getDay(), 1);  
    }  
}
```

# WELL-STRUCTURED ASSERTIONS

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        assertEquals(2050, d.getYear()); // expected
        assertEquals(2, d.getMonth()); // value should
        assertEquals(19, d.getDay()); // be at LEFT
    }

    @Test
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        assertEquals("year after +14 days", 2050, d.getYear());
        assertEquals("month after +14 days", 3, d.getMonth());
        assertEquals("day after +14 days", 1, d.getDay());
    } // test cases should usually have messages explaining
} // what is being checked, for better failure output
```

# EXPECTED ANSWER OBJECTS

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        Date expected = new Date(2050, 2, 19);
        assertEquals(expected, d);    // use an expected answer
                                     // object to minimize tests
    }

                                     // (Date must have toString
                                     // and equals methods)
    @Test
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, d);
    }
}
```

# NAMING TEST CASES

```
public class DateTest {  
    @Test  
    public void test_addDays_withinSameMonth_1() {  
        Date actual = new Date(2050, 2, 15);  
        actual.addDays(4);  
        Date expected = new Date(2050, 2, 19);  
        assertEquals("date after +4 days", expected, actual);  
    }  
    // give test case methods really long descriptive names  
  
    @Test  
    public void test_addDays_wrapToNextMonth_2() {  
        Date actual = new Date(2050, 2, 15);  
        actual.addDays(14);  
        Date expected = new Date(2050, 3, 1);  
        assertEquals("date after +14 days", expected, actual);  
    }  
    // give descriptive names to expected/actual values  
}
```

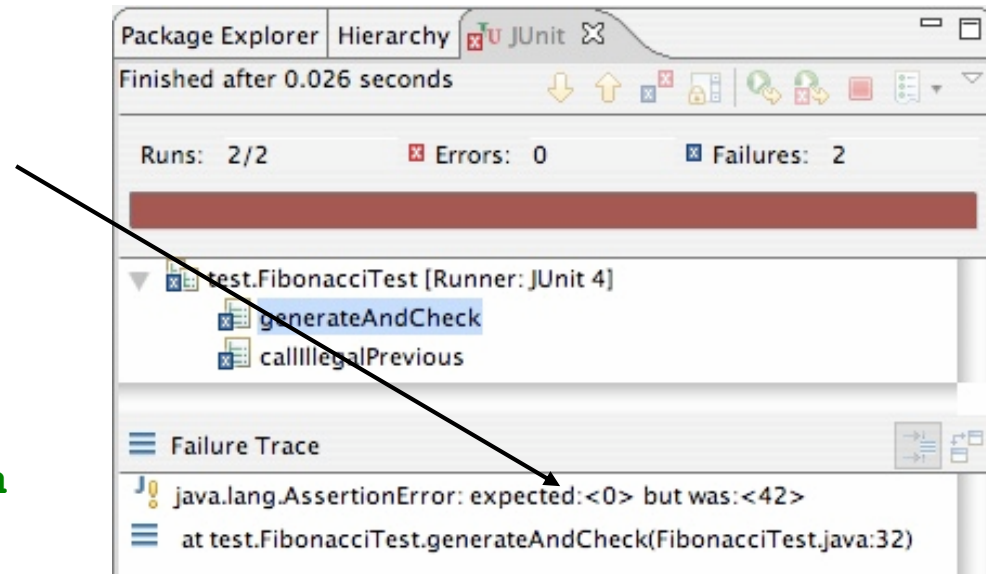
# WHAT'S WRONG WITH THIS?

```
public class DateTest {  
    @Test  
    public void test_addDays_addJustOneDay_1() {  
        Date actual = new Date(2050, 2, 15);  
        actual.addDays(1);  
        Date expected = new Date(2050, 2, 16);  
        assertEquals(  
            "should have gotten " + expected + "\n" +  
            " but instead got " + actual + "\n",  
            expected, actual);  
    }  
    ...  
}
```

# GOOD ASSERTION MESSAGES

```
public class DateTest {  
    @Test  
    public void test_addDays_addJustOneDay_1() {  
        Date actual = new Date(2050, 2, 15);  
        actual.addDays(1);  
        Date expected = new Date(2050, 2, 16);  
        assertEquals("adding one day to 2050/2/15",  
            expected, actual);  
    }  
    ...  
}
```

```
// JUnit will already show  
// the expected and actual  
// values in its output;  
//  
// don't need to repeat them  
// in the assertion message
```



# TESTS WITH A TIMEOUT

```
@Test(timeout = 5000)  
public void name() { ... }
```

- The above method will be considered a failure if it doesn't finish running within 5000 ms

```
private static final int TIMEOUT = 2000;  
...
```

```
@Test(timeout = TIMEOUT)  
public void name() { ... }
```

- Times out / fails after 2000 ms

# PERVASIVE TIMEOUTS

```
public class DateTest {  
    @Test(timeout = DEFAULT_TIMEOUT)  
    public void test_addDays_withinSameMonth_1() {  
        Date d = new Date(2050, 2, 15);  
        d.addDays(4);  
        Date expected = new Date(2050, 2, 19);  
        assertEquals("date after +4 days", expected, d);  
    }  
  
    @Test(timeout = DEFAULT_TIMEOUT)  
    public void test_addDays_wrapToNextMonth_2() {  
        Date d = new Date(2050, 2, 15);  
        d.addDays(14);  
        Date expected = new Date(2050, 3, 1);  
        assertEquals("date after +14 days", expected, d);  
    }  
  
    // almost every test should have a timeout so it can't  
    // lead to an infinite loop; good to set a default, too  
    private static final int DEFAULT_TIMEOUT = 2000;  
}
```

# TESTING FOR EXCEPTIONS

```
@Test(expected = ExceptionType.class)
public void name() {
    ...
}
```

- Will pass if it *does* throw the given exception.
  - If the exception is *not* thrown, the test fails.
  - Use this to test for expected errors.

```
@Test(expected = ArrayIndexOutOfBoundsException.class)
public void testBadIndex() {
    ArrayList list = new ArrayList();
    list.get(4);    // should fail
}
```

# SETUP AND TEARDOWN

**@Before**

```
public void name() { ... }
```

**@After**

```
public void name() { ... }
```

- methods to run before/after each test case method is called

**@BeforeClass**

```
public static void name() { ... }
```

**@AfterClass**

```
public static void name() { ... }
```

- methods to run once before/after the entire test class runs

# TIPS FOR TESTING

- You cannot test every possible input, parameter value, etc.
  - So you must think of a limited set of tests likely to expose bugs.
- Think about boundary cases
  - positive; zero; negative numbers
  - right at the edge of an array or collection's size
- Think about empty cases and error cases
  - 0, -1, null; an empty list or array
- test behavior in combination
  - maybe `add` usually works, but fails after you call `remove`
  - make multiple calls; maybe `size` fails the second time only

# WHAT'S WRONG WITH THIS?

```
public class DateTest {  
    // test every day of the year  
    @Test(timeout = 10000)  
    public void tortureTest() {  
        Date date = new Date(2050, 1, 1);  
        int month = 1;  
        int day = 1;  
        for (int i = 1; i < 365; i++) {  
            date.addDays(1);  
            if (day < DAYS_PER_MONTH[month]) {day++;}  
            else {month++; day=1;}  
            assertEquals(new Date(2050, month, day), date);  
        }  
    }  
  
    private static final int[] DAYS_PER_MONTH = {  
        0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31  
    }; // Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec  
}
```

# TRUSTWORTHY TESTS

- Test one thing at a time per test method.
  - 10 small tests are much better than 1 test 10x as large.
- Each test method should have few (likely 1) assert statements.
  - If you assert many things, the first that fails stops the test.
  - You won't know whether a later assertion would have failed.
- Tests should avoid logic.
  - minimize `if/else`, `loops`, `switch`, etc.
  - avoid `try/catch`
    - If it's supposed to throw, use `expected= ...` if not, let JUnit catch it.
- Torture tests are okay, but only *in addition to* simple tests.

# JUNIT EXERCISE

Given our Date class seen previously:

- `public Date(int year, int month, int day)`
  - `public Date()` *// today*
  - `public int getDay(), getMonth(), getYear()`
  - `public void addDays(int days)` *// advances by days*
  - `public int daysInMonth()`
  - `public String dayOfWeek()` *// e.g. "Sunday"*
  - `public boolean equals(Object o)`
  - `public boolean isLeapYear()`
  - `public void nextDay()` *// advances by 1 day*
  - `public String toString()`
- Come up with unit tests to check the following:
    - That no Date object can ever get into an invalid state.
    - That the addDays method works properly.
      - It should be efficient enough to add 1,000,000 days in a call.

# SQUASHING REDUNDANCY

```
public class DateTest {
    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_withinSameMonth_1() {
        addHelper(2050, 2, 15, +4, 2050, 2, 19);
    }

    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_wrapToNextMonth_2() {
        addHelper(2050, 2, 15, +14, 2050, 3, 1);
    }

    // use lots of helpers to make actual tests extremely short
    private void addHelper(int y1, int m1, int d1, int add,
                           int y2, int m2, int d2) {
        Date act = new Date(y, m, d);
        actual.addDays(add);
        Date exp = new Date(y2, m2, d2);
        assertEquals("after +" + add + " days", exp, act);
    }

    // can also use "parameterized tests" in some frameworks
    ...
}
```

# FLEXIBLE HELPERS

```
public class DateTest {
    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_multipleCalls_wrapToNextMonth2x() {
        Date d = addHelper(2050, 2, 15, +14, 2050, 3, 1);
        addHelper(d, +32, 2050, 4, 2);
        addHelper(d, +98, 2050, 7, 9);
    }

    // Helpers can box you in; hard to test many calls/combine.
    // Create variations that allow better flexibility
    private Date addHelper(int y1, int m1, int d1, int add,
                           int y2, int m2, int d2) {
        Date date = new Date(y, m, d);
        addHelper(date, add, y2, m2, d2);
        return d;
    }

    private void addHelper(Date date, int add,
                           int y2, int m2, int d2) {
        date.addDays(add);
        Date expect = new Date(y2, m2, d2);
        assertEquals("date after +" + add + " days", expect, d);
    }

    ...
}
```

# REGRESSION TESTING

- **regression:** When a feature that used to work, no longer works.
  - Likely to happen when code changes and grows over time.
  - A new feature/fix can cause a new bug or reintroduce an old bug.
- **regression testing:** Re-executing prior unit tests after a change.
  - Often done by scripts during automated testing.
  - Used to ensure that old fixed bugs are still fixed.
  - Gives your app a minimum level of working functionality.
- Many products have a set of mandatory check-in tests that must pass before code can be added to a source code repository.

# TEST-DRIVEN DEVELOPMENT

- Unit tests can be written after, during, or even *before* coding.
  - **test-driven development:** Write tests, *then* write code to pass them.
- Imagine that we'd like to add a method `subtractWeeks` to our `Date` class, that shifts this `Date` backward in time by the given number of weeks.
- Write code to test this method *before* it has been written.
  - Then once we do implement the method, we'll know if it works.

# TESTS AND DATA STRUCTURES

- Need to pass lots of arrays? Use array literals

```
public void exampleMethod(int[] values) { ... }  
...  
exampleMethod(new int[] {1, 2, 3, 4});  
exampleMethod(new int[] {5, 6, 7});
```

- Need a quick ArrayList? Try Arrays.asList

```
List<Integer> list = Arrays.asList(7, 4, -2, 3, 9, 18);
```

- Need a quick set, queue, etc.? Many collections can take a list

```
Set<Integer> list = new HashSet<Integer>(  
    Arrays.asList(7, 4, -2, 9));
```

# WHAT'S WRONG WITH THIS?

```
public class DateTest {  
    // shared Date object to test with (saves memory!!1)  
    private static Date DATE;  
  
    @Test(timeout = DEFAULT_TIMEOUT)  
    public void addDays_sameMonth() {  
        DATE = new Date(2050, 2, 15);           // first test;  
        addhelper(DATE, +4, 2050, 2, 19); // DATE = 2/15 here  
    }  
  
    @Test(timeout = DEFAULT_TIMEOUT)  
    public void addDays_nextMonthWrap() { // second test;  
        addhelper(DATE, +10, 2050, 3, 1); // DATE = 2/19 here  
    }  
  
    @Test(timeout = DEFAULT_TIMEOUT)  
    public void addDays_multipleCalls() { // third test;  
        addDays_sameMonth(); // go back to 2/19;  
        addhelper(DATE, +1, 2050, 2, 20); // test two calls  
        addhelper(DATE, +1, 2050, 2, 21);  
    }  
    ...  
}
```

# TEST CASE "SMELLS"

- Tests should be self-contained and not care about each other.
- "Smells" (bad things to avoid) in tests:
  - *Constrained test order* : Test A must run before Test B.  
(usually a misguided attempt to test order/flow)
  - *Tests call each other* : Test A calls Test B's method  
(calling a shared helper is OK, though)
  - *Mutable shared state* : Tests A/B both use a shared object.  
(If A breaks it, what happens to B?)



# TEST SUITES

- **test suite:** One class that runs many JUnit tests.
  - An easy way to run all of your app's tests at once.

```
import org.junit.runner.*;  
import org.junit.runners.*;
```

```
@RunWith(Suite.class)  
@Suite.SuiteClasses({  
    TestCaseName.class,  
    TestCaseName.class,  
    ...  
    TestCaseName.class,  
})  
public class name {}
```

# TEST SUITE EXAMPLE

```
import org.junit.runner.*;
import org.junit.runners.*;

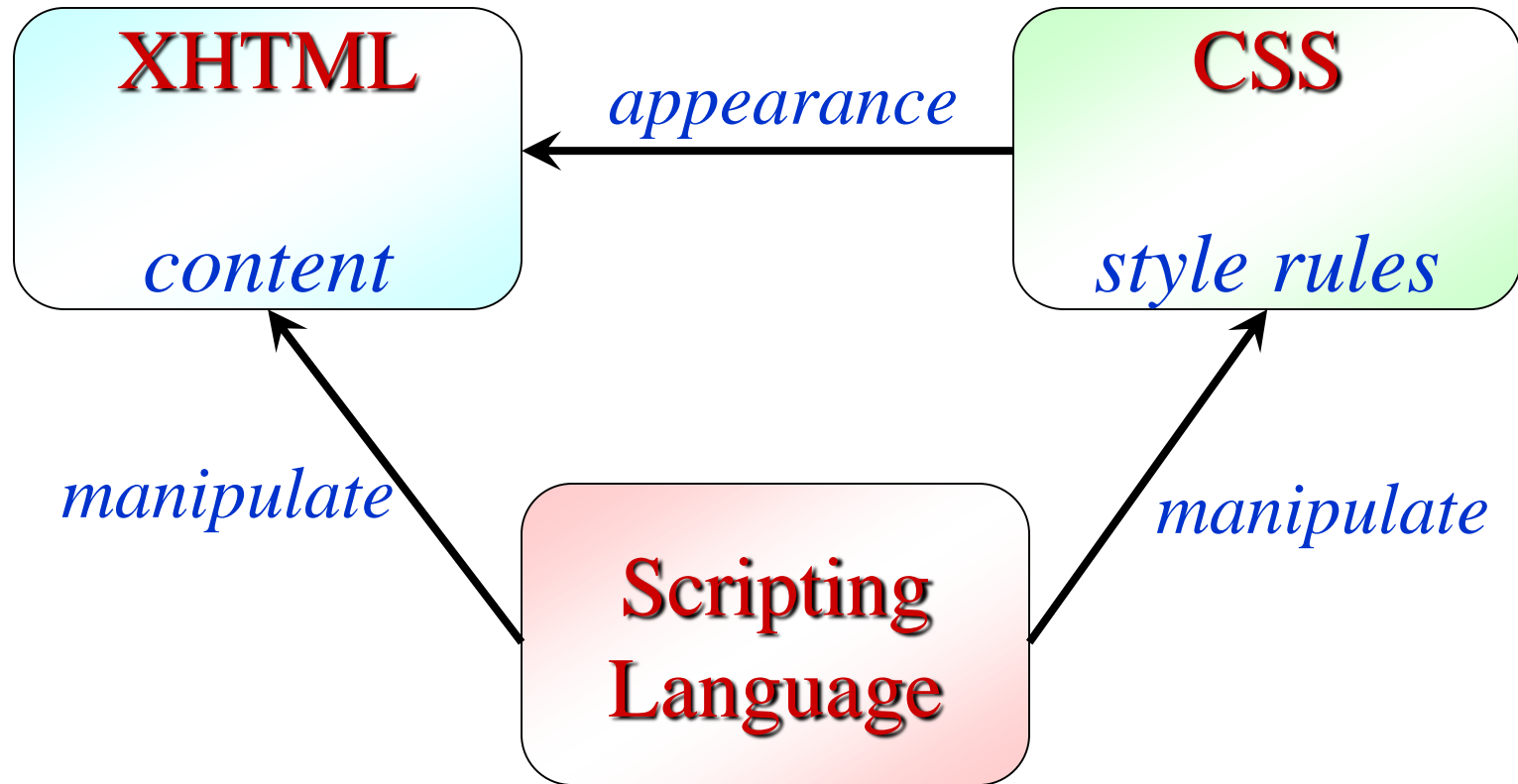
@RunWith(Suite.class)
@Suite.SuiteClasses({
    WeekdayTest.class,
    TimeTest.class,
    CourseTest.class,
    ScheduleTest.class,
    CourseComparatorsTest.class
})
public class HW2Tests {}
```

# JUNIT SUMMARY

- Tests need *failure atomicity* (ability to know exactly what failed).
  - Each test should have a clear, long, descriptive name.
  - Assertions should always have clear messages to know what failed.
  - Write many small tests, not one big test.
    - Each test should have roughly just 1 assertion at its end.
- Always use a `timeout` parameter to every test.
- Test for expected errors / exceptions.
- Choose a descriptive assert method, not always `assertTrue`.
- Choose representative test cases from equivalent input classes.
- Avoid complex logic in test methods if possible.
- Use helpers, `@Before` to reduce redundancy between tests.

### 3) DHTML AND DOM

## Dynamic HTML



# DHTML

- A combination of technologies used to create animated documents
- Not a W3C standard!
  - Originally, a marketing term used by Netscape and Microsoft
- Using *scripts*, we manipulate *HTML* content and *style* properties in reaction to *events*

# HTML DOM

“A platform- and language-neutral interface that allows programs and **scripts** to dynamically access and **update** the **content** and **structure** of HTML and XHTML documents.”

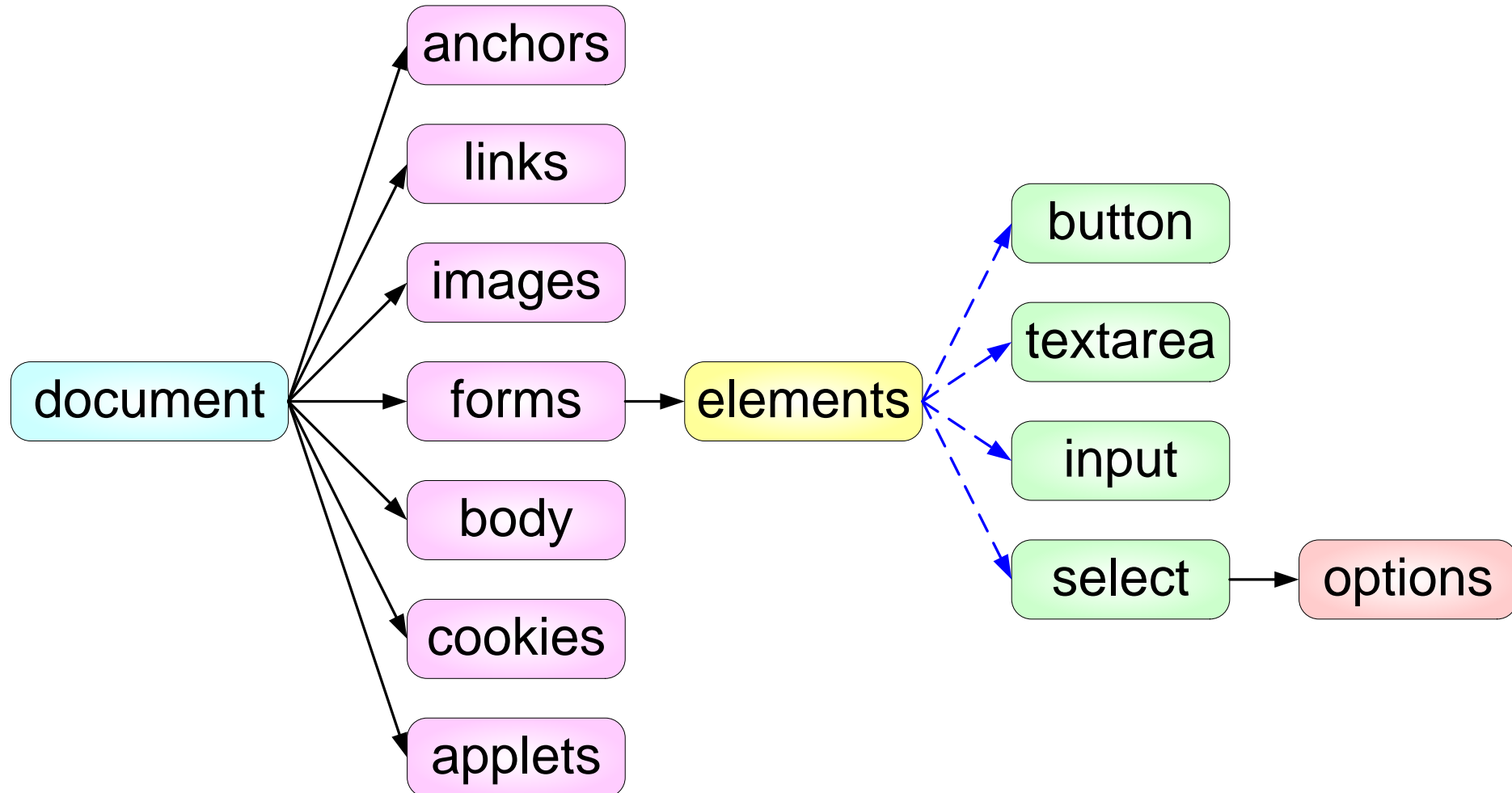
# DOM AND JAVASCRIPT

- Combined with **JavaScript**, every element in the HTML document is represented by an *object*
- Elements can be manipulated using the **properties** and **methods** of the corresponding objects
- Changes in the element properties are immediately reflected by the browser

# ACCESSING HTML ELEMENTS

- All HTML elements (objects) are accessed through the **document** object
- **document** itself is automatically created
- Several ways to access a specific element
  - **paths** in the DOM tree
  - retrieval by **tag**
  - retrieval by **ID**

# HTML DOM TREE



# ACCESSING ELEMENTS BY PATHS

```
function execute() {  
    var img = document.images[0]; img.src="lighton.gif"  
    var inx = document.forms[0].elements[0]; inx.value="xx"  
    var iny = document.forms["form1"].elements["y"]; iny.value="yy"  
}
```

*head*

```
<p></p>  
<form id="form1" method="get" action="nosuch"><p>  
    <input type="text" name="x"/>  
    <input type="text" name="y"/>  
    <input type="reset"/></p>  
</form>
```

*body*

# ACCESSING ELEMENTS BY TAGS

```
function execute() {  
    var spans = document.getElementsByTagName("span");  
    spans[0].style.color="red";  
    spans[1].style.color="blue";  
    spans[1].style.fontVariant="small-caps";  
}
```

*head*

```
<p>This <span>example</span> is lovely.</p>  
<p>But <span>this one</span> is even more!</p>
```

*body*

# ACCESSING ELEMENTS BY ID

```
function execute() {  
  var theDiv = document.getElementById("div1");  
  if (theDiv.style.visibility=="hidden")  
    {theDiv.style.visibility="visible" }  
  else {theDiv.style.visibility="hidden" }  
}
```

*head*

```
<div id="div1"> This text can be hidden!</div>
```

*body*

This technique is more *stable*  
w.r.t. document changes (*why?*)

# ELEMENT PROPERTIES

- Elements of different types have different sets of properties and methods
- See [www.w3schools.com/html/dom/](http://www.w3schools.com/html/dom/) for a detailed list of element properties and methods
- Most elements have the **style** member
- **style** is an object that represents the style-sheet rules applied over the element

# EVENT EXAMPLE

```
<html>
<head>
  <title>Simple Events</title>
  <script type="text/javascript">
    function focusInput() {
      var theInput = document.getElementsByTagName("input")[0]
      theInput.style.background="yellow" }
    function blurInput() {
      theInput = document.getElementsByTagName("input")[0]
      theInput.style.background="white" }
  </script>
</head>
```

# EVENT EXAMPLE (CONT)

```
<body>
  <p>
    
  </p>
  <p>
    <input type="text" onfocus="focusInput()"
      onblur="blurInput()" />
  </p>
</body>
</html>
```

# EVENT MODEL

- *Events* usually occur due to users actions
  - For example, pressing the keyboard, changing a text field, moving the mouse over an element, etc.
- An event is represented by an *event object* that is created upon the event occurrence
- Every event has an associated *target element*
  - For example, the image over which the mouse clicks

## EVENT MODEL (CONT)

- Elements can have **registered** *event listeners* which are associated with certain types of events
- *When an event takes place, the listeners that are registered for this event are invoked*
- Typically, a listener is described by a scripting code (e.g., JavaScript)
  - This code is executed upon listener invocation

# INLINE LISTENER REGISTRATION

- The simplest (and most common) way to register a listener is by an attribute assignment:

*on***type** = "*JavaScript code*"

- For example:

**<img** src="img.gif" **onmouseover**="alert('!')" />

- The JavaScript code has access to the following objects:
  - **this** - the element (e.g., the image defined above)
  - **event** - the event object

# SOME EVENT TYPES

*load*

*click*

*reset*

*unload*

*dblclick*

*select*

*abort*

*mousedown*

*submit*

*mousemove*

*keydown*

*mouseup*

*change*

*keypress*

*mouseover*

*blur*

*keyup*

*focus*

# ANOTHER EXAMPLE

```
<html>
```

```
<head><title>Event Object Example</title>
```

```
<script type="text/javascript">
```

```
function execute(e) {  
    alert(" x: " + e.clientX + ", y: " + e.clientY +  
        " mouse button: " + e.button); }  

```

```
</script></head>
```

```
<body onmousedown="execute(event)"  
    style="cursor: pointer;  
    position: absolute; width: 100%; height: 100%"> </body>
```

```
</html>
```

# FORM VALIDATION

- In the `form` element, `onsubmit="code"` defines a listener with a special functionality
- When the form is supposed to be submitted, `code` is executed **before** submission
- The `code` can return a `Boolean` value
  - e.g., `onsubmit="return function()"`
- If `code` returns `false`, submission is cancelled!

# FORM VALIDATION - SIMPLE EXAMPLE

```
<html>
  <head><title>Form Validation</title>
    <script type="text/javascript">
      function validate() {
        var theX = document.forms[0].x.value;
        var theY = document.forms[0].y.value;
        if(theX != theY) { alert("x != y!!"); return false; }
        else { return true; }
      }
    </script>
  </head>
```

# FORM VALIDATION - SIMPLE EXAMPLE (CONT)

```
<body>
  <form id="email-form" action="myurl" method="get"
    onsubmit="return validate()">
    <p>
      x: <input type="text" name="x" />
      y: <input type="text" name="y" />
      <input type="submit" />
    </p>
  </form>
</body>
</html>
```

# FORM VALIDATION - ANOTHER EXAMPLE

```
<head><title>Form Validation</title>
<script type="text/javascript">
  function validateEmail(form) {
    var emailRegExp = /^\\w+\\@\\w+\\.\\w+$/;
    var theEmail = form.email.value;
    if(theEmail.match(emailRegExp)) { return true; }
    alert(theEmail + " is not a valid email!");
    return false;
  }
</script>
</head>
```

# FORM VALIDATION - ANOTHER EXAMPLE (CONT)

```
<body>
  <form id="email-form" action="myurl" method="get"
    onsubmit="return validateEmail()">
    <p>
      Name: <input type="text" name="Name:" /> <br/>
      Email: <input type="text" name="email" />
      <input type="submit" />
    </p>
  </form>
</body>
```

# FORM SUBMISSION

- A form can be submitted without the special submission button
- Use the function `form.submit()` to submit a specific form from JavaScript code

# MOUSE-CLICK EVENTS

- To register a listener for the **click** event, use can use the **onclick** attribute of the element
  - Apply the style rule **cursor:pointer** to the element in order to get the pointer effect
- Alternatively, you can link to a JavaScript code:
  - `<a href="javascript:code">Click here</a>`



# EVENT FLOW

```
<script type="text/javascript">  
    function f1() { alert("1") }  
    function f2() { alert("2") }  
    function f3() { alert("3") }  
</script>
```

```
<body>  
    <div onclick="f1()">  
        <p onclick="f2()">  
              
        </p>  
    </div>  
</body>
```

# EVENT FLOW (CONT)

- When we click on the image, which of the functions should be executed?
  - **Answer:** all of them!
- In what order?
- Two different models:
  - **Microsoft** (impl. in **IE**)
  - **W3C** (impl. in **Mozilla, Opera 7, Konqueror**)

# MICROSOFT MODEL

- **Event Bubbling**: events propagate through the elements in bottom-up order
  - i.e., from the most specific to the most general
- Whenever an element is visited, its listeners are triggered
- In our example: `img`  $\rightarrow$  `p`  $\rightarrow$  `div`

# W3C MODEL

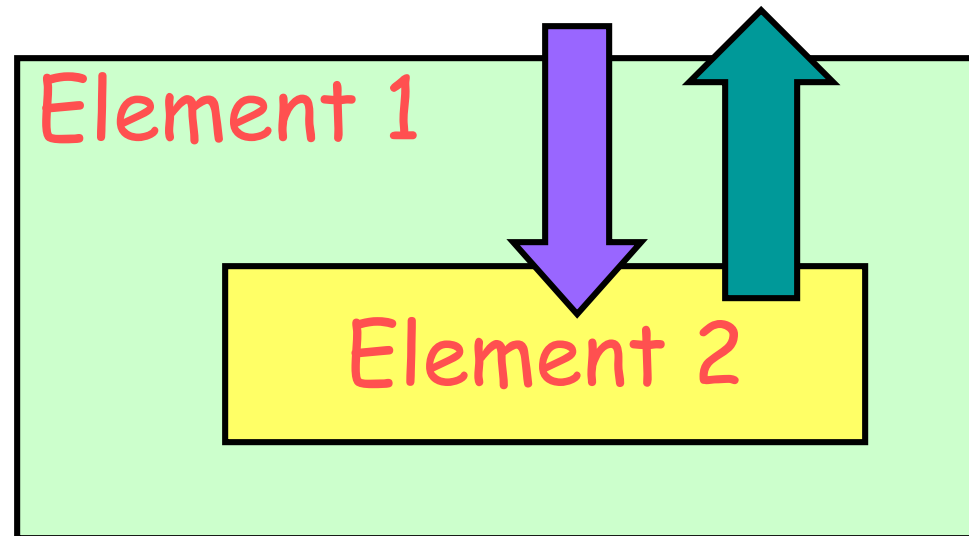
- In the W3C model, there are 2 traversals:

**1. Event capturing:** top-down

- e.g.,  $\text{div} \rightarrow \text{p} \rightarrow \text{img}$

**2. Event bubbling:** bottom-up

- e.g.,  $\text{img} \rightarrow \text{p} \rightarrow \text{div}$



# EVENT FLOW (CONT)

- A listener can be registered in either the capturing or the bubbling phase
- By default, listeners register in the bubbling phase
  - So, what will be the result of the example code?
- An element may choose to stop the flow at any listener execution, by calling `event.stopPropagation()`
  - In IE: `event.cancelBubble = true`

## AN EXAMPLE

- What will happen if we replace **f2** with the following?

```
function f2(e) {  
    alert("2");  
    if(e.stopPropagation) e.stopPropagation();  
    if(e.cancelBubble!= undefined) e.cancelBubble=true;  
}
```

# DYNAMIC LISTENER REGISTRATION

- A listener can be dynamically registered by using JavaScript code
- Microsoft:

*element.ontype = functionName*

element.attachEvent("on~~type~~", *functionName*)

- Note that the function is given as an **object**
- The function is called without arguments
- The event can be accessed using **window.event**

# DYNAMIC LISTENER REGISTRATION (CONT)

- W3C:

*element.ontype = functionName*

*element.addEventListener("type", functionName, isCapture)*

- The function is called with *event* as an argument
- If *isCapture* is *true*, the listener is registered for the capturing phase

# STRUCTURE MANIPULATION

- In structure manipulation, we
  - **add/remove/replace** HTML elements
  - change the text under elements
- Two approaches:
  - **DOM tree** manipulation (W3C specification)
  - Setting the **innerHTML** attribute (not a specification)

# DOM TREE MANIPULATION

- In this approach, we explicitly
  - **create** new nodes
  - **add** created nodes to the DOM tree
  - **remove** old nodes
- To create new nodes, use these methods of **document**:
  - **document.createElement**("tag")
  - **document.createTextNode**("text")
  - **document.createAttribute**("attname")

# DOM TREE MANIPULATION (CONT)

- To add and remove children of a specific element, use the following methods:
  - *element.appendChild(newChild)*
  - *element.insertBefore(newChild, child)*
  - *element.removeChild(child)*
  - *element.replaceChild(newChild, oldChild)*

# AN EXAMPLE

```
<html>
<head><script type="text/javascript">...</script></head>
<body>
  <p>First Paragraph.</p>
  <div id="d1"><p id="p1">Second paragraph.</p></div>
  <p>
    <input type="button" value="replace"
      onclick="replace(this)" />
  </p>
</body>
</html>
```

# AN EXAMPLE (CONT)

```
function replace(button) {  
    d = document.getElementById("d1");  
    p = document.getElementById("p1");  
  
    h = document.createElement("h1");  
    text = document.createTextNode("This is a header.");  
    h.appendChild(text);  
  
    d.replaceChild(h,p);  
  
    button.disabled=true;  
}
```

# THE `innerHTML` PROPERTY

- The attribute `innerHTML` attribute of an element is the HTML code embedded inside that element
- Hence, you can replace existing content by setting this attribute:
  - `element.innerHTML = "new HTML code"`
- Not recognized by W3C specifications, but supported by Web browsers

# PREVIOUS EXAMPLE WITH `innerHTML`

```
function replace(button) {  
    d = document.getElementById("d1");  
    d.innerHTML = "<h1>This is a header</h1>";  
    button.disabled=true;  
}
```

# THE window OBJECT

- Built-in object called `window`
- Represents the browser window of the `document`
- Several window objects may co-exist
  - Separate windows/tabs
  - Separate frames
- Default object – need not specify `window` to access its properties and methods
  - `window.alert()` and `alert()` are the same

# DIALOG BOXES

- `alert("warning!!!");`
- `confirm("are you sure?");`
  - returned value is **Boolean**
- `prompt("enter your name");`
  - returned value is either a **string** or a **null object**



Rendering stops until box closure!

# AN EXAMPLE

```
<script type="text/javascript">
```

```
    alert("You are about to start");  
    document.write("Started<br/>");
```

```
    conf = confirm("Should I continue?");
```

```
    if(conf) {
```

```
        name = prompt("Enrer your name")
```

```
        document.write("Your name is " + name + "<br/>");
```

```
    }
```

```
</script>
```

# THE location OBJECT

- The object `window.location` represents the current URL of the window
- For example:
  - `location.href` - the current URL (can be changed!)
  - `location.hostname`
  - `location.pathname`
- Also has methods:
  - `location.reload()`,
  - `location.replace('URL')`

# OPENING NEW WINDOWS

- `window.open("URL")` - opens URL in a new window
  - you can specify other properties, like size, whether it is resizable, etc.
  - returns the new `window` object

# ACCESSING WINDOW FRAMES

- `window.top` - the topmost window
- `window.frames` - a collection of the frames in the window
- For example, in a specific frame, use `window.top.frames[i]` to get to another frame

# THE navigator OBJECT

- The object `window.navigator` contains information about the browser
- For example:
  - `navigator.appName` - the name of the browser
  - `navigator.appVersion` - the version of the browser
  - `navigator.cookieEnabled`
  - `navigator.platform` - the OS name

# THE history OBJECT

- The object `window.history` enables navigation according to the navigation history
- For example:
  - `history.back()` - same as clicking the `back` button
  - `history.forward()` - same as clicking the `forward` button
  - `history.go(i)` - go forward *i* times
    - If *i* is negative, go back *-i* times

## 4) JAVASCRIPT SYNTAX

# JAVASCRIPT HISTORY AND VERSIONS

- **JavaScript** was introduced as part of the Netscape 2.0 browser
- Microsoft soon released its own version called **JScript**
- ECMA developed a standard language known as **ECMAScript**
- ECMAScript Edition 3 is widely supported and is what we will call “JavaScript”

# JAVASCRIPT INTRODUCTION

- Let's write a “Hello World!” JavaScript program
- **Problem:** the JavaScript language itself has **no input/output** statements(!)
- **Solution:** Most browsers provide *de facto* standard I/O methods
  - `alert`: pops up **alert box** containing text
  - `prompt`: pops up window where user can enter text

# JAVASCRIPT INTRODUCTION

- File JSHelloWorld.js:
- HTML document executing this code:

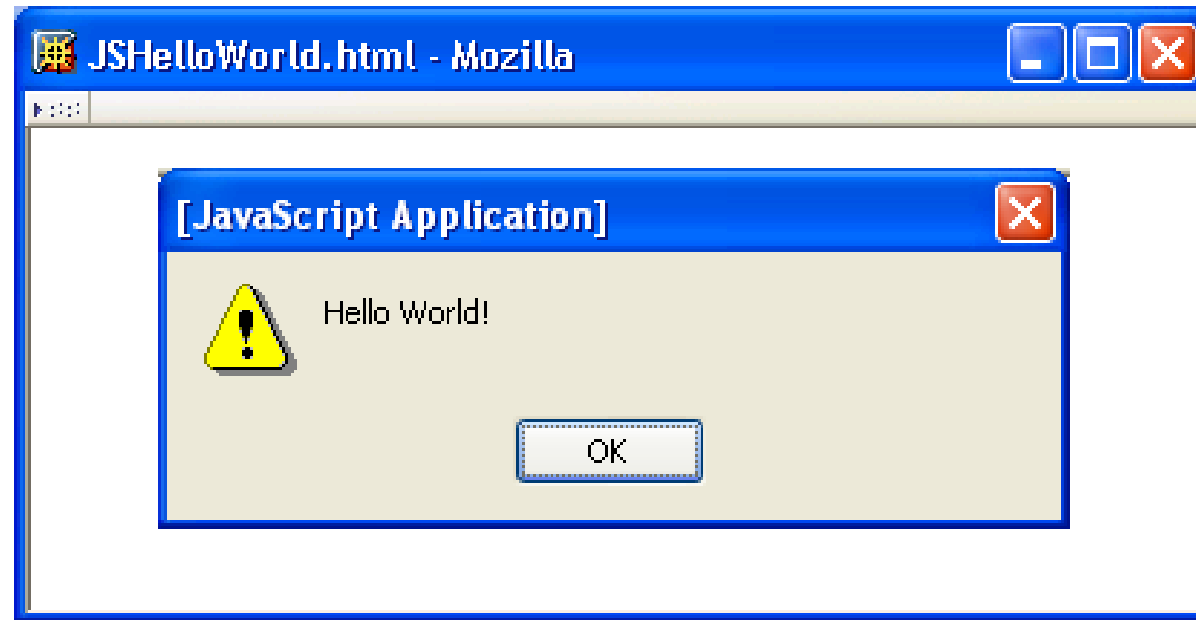
```
window.alert("Hello World!");
```

```
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>
      JSHelloWorld.html
    </title>
    <script type="text/javascript" src="JSHelloWorld.js">
    </script>
  </head>
  <body>
  </body>
</html>
```

} script element used  
to load and execute  
JavaScript code

# JAVASCRIPT INTRODUCTION

- Web page and alert box generated by JSHelloWorld.html document and JSHelloWorld.js code:



# JAVASCRIPT INTRODUCTION

- Prompt window example:

```
var inString = window.prompt("Enter JavaScript code to be tested:",  
                             "");
```



# JAVASCRIPT PROPERTIES

- Note that JavaScript code did not need to be compiled
  - JavaScript is an **interpreted** language
  - Portion of browser software that reads and executes JavaScript is an **interpreter**
- Interpreted vs. compiled languages:
  - Advantage: **simplicity**
  - Disadvantage: **efficiency**

# JAVASCRIPT PROPERTIES

- JavaScript is a **scripting language**: designed to be executed within a larger software environment
- JavaScript can be run within a variety of environments:
  - Web browsers (our focus in next chapter)
  - Web servers
  - Application containers (general-purpose programming)

# JAVASCRIPT PROPERTIES

- Components of a JavaScript implementation:
  - **Scripting engine**: interpreter plus required ECMAScript functionality (core library)
  - **Hosting environment**: functionality specific to environment
    - Example: browsers provide `alert` and `prompt`
    - All hosting environment functionality provided via objects

# JAVASCRIPT PROPERTIES

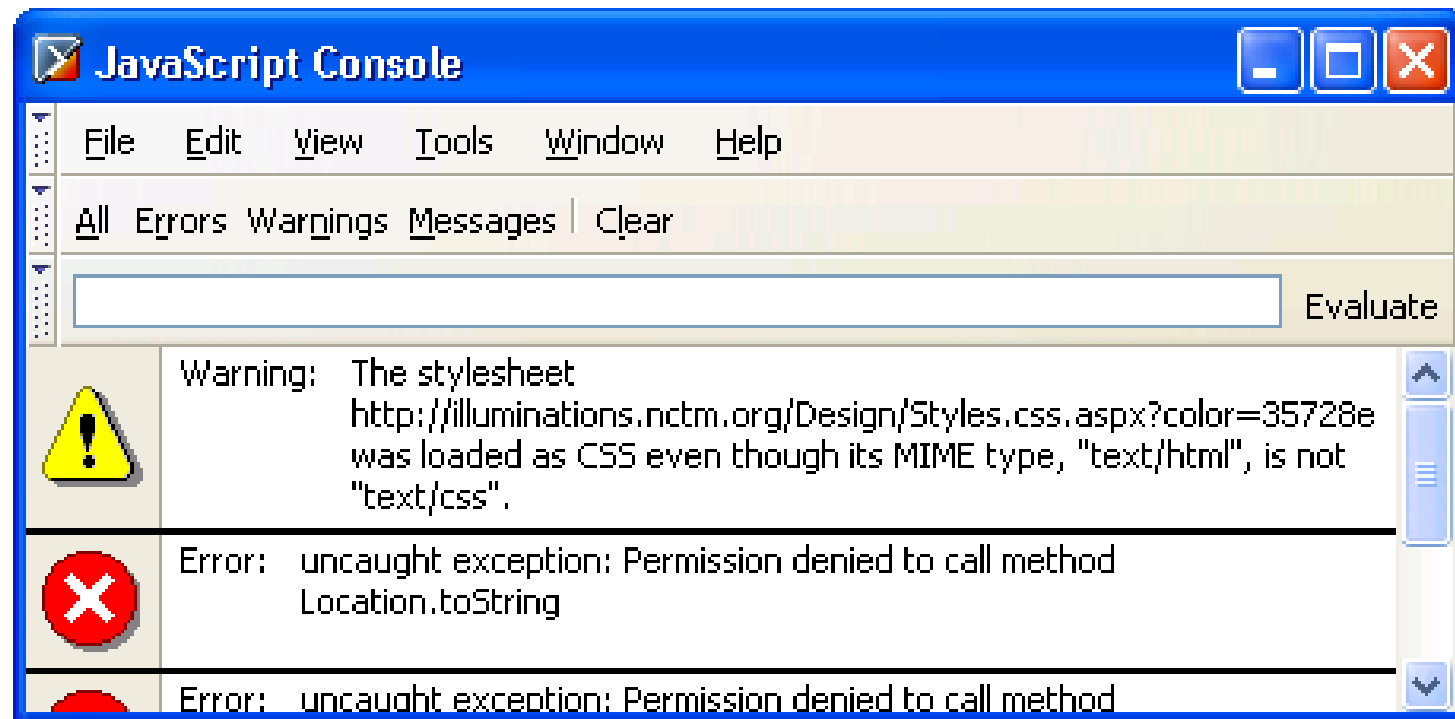
- All data in JavaScript is an object or a property of an object
- Types of JavaScript objects
  - **Native**: provided by scripting engine
    - If automatically constructed before program execution, known as a **built-in** object (ex: window)
  - **Host**: provided by host environment
    - `alert` and `prompt` are host objects

# DEVELOPING JAVASCRIPT SOFTWARE

- **Writing** JavaScript code
  - Any text editor (*e.g.*, Notepad, Emacs)
  - Specialized software (*e.g.*, MS Visual InterDev)
- **Executing** JavaScript
  - Load into browser (need HTML document)
  - Browser detects **syntax** and **run-time** errors
    - Mozilla: JavaScript console lists errors
    - IE6: Exclamation icon and pop-up window

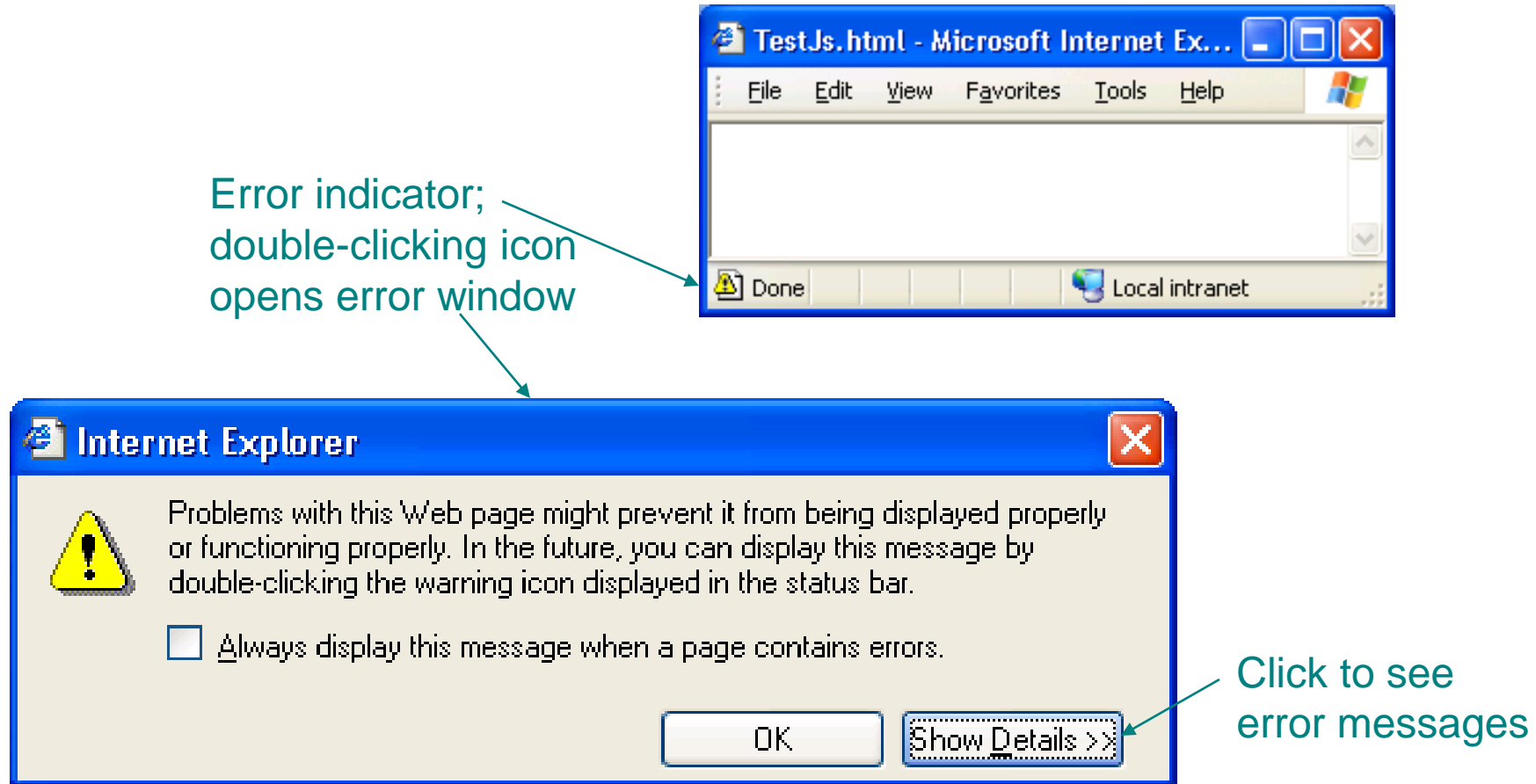
# DEVELOPING JAVASCRIPT SOFTWARE

- Mozilla JavaScript console (Tools | Web Development | JavaScript Console):



# DEVELOPING JAVASCRIPT SOFTWARE

- IE6 error window:



# DEVELOPING JAVASCRIPT SOFTWARE

- Firefox (2.0 and up): the JavaScript console has been renamed “Error Console” (Tools|Error Console) and shows JavaScript errors, CSS errors etc...
- Enhancements available as extensions (e.g. Console<sup>2</sup>, firebug)
- Chrome (4) has excellent dev support (developer|JavaScript Console)
- IE8: Tools|Developer tools

# DEVELOPING JAVASCRIPT SOFTWARE

- Debugging

- Apply generic techniques: desk check, add debug output (alert's)
- Use specialized JavaScript debuggers: later

- Re-executing

- Overwrite .js file
- Reload (Mozilla)/Refresh (IE) HTML document that loads the file

# BASIC JAVASCRIPT SYNTAX

```
// HighLow.js

var thinkingOf; // Number the computer has chosen (1 through 1000)
var guess;      // User's latest guess

// Initialize the computer's number
thinkingOf = Math.ceil(Math.random()*1000);

// Play until user guesses the number
guess = window.prompt("I'm thinking of a number between 1 and 1000." +
    " What is it?", "");
```

# BASIC JAVASCRIPT SYNTAX

Notice that there is no main() function/method

```
// HighLow.js

var thinkingOf; // Number the computer has chosen (1 through 1000)
var guess;      // User's latest guess

// Initialize the computer's number
thinkingOf = Math.ceil(Math.random()*1000);

// Play until user guesses the number
guess = window.prompt("I'm thinking of a number between 1 and 1000." +
                      "  What is it?", "");
```

# BASIC JAVASCRIPT SYNTAX

// HighLow.js

Comments like Java/C++ (/\* \*/ also allowed)

```
var thinkingOf; // Number the computer has chosen (1 through 1000)
var guess;      // User's latest guess

// Initialize the computer's number
thinkingOf = Math.ceil(Math.random()*1000);

// Play until user guesses the number
guess = window.prompt("I'm thinking of a number between 1 and 1000." +
    " What is it?", "");
```

# BASIC JAVASCRIPT SYNTAX

Variable declarations:

- Not required
- Data type not specified

```
// HighLow.js
```

```
var thinkingOf; // Number the computer has chosen (1 through 1000)  
var guess;      // User's latest guess
```

```
// Initialize the computer's number  
thinkingOf = Math.ceil(Math.random()*1000);
```

```
// Play until user guesses the number  
guess = window.prompt("I'm thinking of a number between 1 and 1000." +  
    " What is it?", "");
```

# BASIC JAVASCRIPT SYNTAX

```
// HighLow.js
```

```
var thinkingOf; // Number the computer has chosen (1 through 1000)
```

```
var guess; // User's latest guess
```

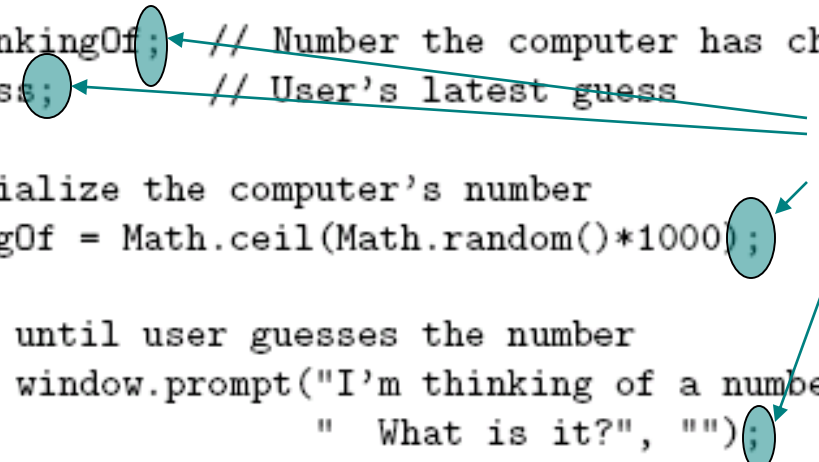
```
// Initialize the computer's number
```

```
thinkingOf = Math.ceil(Math.random()*1000);
```

```
// Play until user guesses the number
```

```
guess = window.prompt("I'm thinking of a number between 1 and 1000." +  
    " What is it?", "");
```

Semi-colons are usually not required, but always allowed at statement end



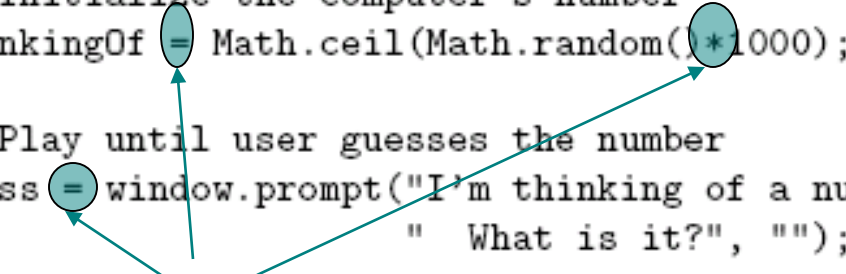
# BASIC JAVASCRIPT SYNTAX

```
// HighLow.js

var thinkingOf; // Number the computer has chosen (1 through 1000)
var guess;      // User's latest guess

// Initialize the computer's number
thinkingOf = Math.ceil(Math.random()*1000);

// Play until user guesses the number
guess = window.prompt("I'm thinking of a number between 1 and 1000." +
    " What is it?", "");
```



Arithmetic operators same as Java/C++

# BASIC JAVASCRIPT SYNTAX

```
// HighLow.js

var thinkingOf; // Number the computer has chosen (1 through 1000)
var guess;      // User's latest guess

// Initialize the computer's number
thinkingOf = Math.ceil(Math.random()*1000);

// Play until user guesses the number
guess = window.prompt("I'm thinking of a number between 1 and 1000."
    " What is it?", "");
```



String concatenation operator  
as well as addition

# BASIC JAVASCRIPT SYNTAX

```
// HighLow.js
```

```
var thinkingOf; // Number the computer has chosen (1 through 1000)
var guess;      // User's latest guess
```

Arguments can be any expressions

```
// Initialize the computer's number
thinkingOf = Math.ceil(Math.random()*1000);
```

```
// Play until user guesses the number
guess = window.prompt("I'm thinking of a number between 1 and 1000." +
    " What is it?", "");
```

Argument lists are comma-separated

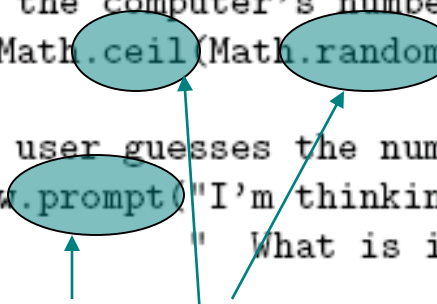
# BASIC JAVASCRIPT SYNTAX

```
// HighLow.js

var thinkingOf; // Number the computer has chosen (1 through 1000)
var guess;      // User's latest guess

// Initialize the computer's number
thinkingOf = Math.ceil(Math.random()*1000);

// Play until user guesses the number
guess = window.prompt("I'm thinking of a number between 1 and 1000." +
    " What is it?", "");
```



Object dot notation for method calls as in Java/C++

# BASIC JAVASCRIPT SYNTAX

```
while (guess != thinkingOf)
{

    // Evaluate the user's guess
    if (guess < thinkingOf) {
        guess = window.prompt("Your guess of " + guess +
                               " was too low.  Guess again.", "");
    }
    else {
        guess = window.prompt("Your guess of " + guess +
                               " was too high.  Guess again.", "");
    }
}

// Game over; congratulate the user
window.alert(guess + " is correct!");
```

# BASIC JAVASCRIPT SYNTAX

```
while (guess != thinkingOf)
{
    // Evaluate the user's guess
    if (guess < thinkingOf) {
        guess = window.prompt("Your guess of " + guess +
                               " was too low.  Guess again.", "");
    }
    else {
        guess = window.prompt("Your guess of " + guess +
                               " was too high.  Guess again.", "");
    }
}

// Game over; congratulate the user
window.alert(guess + " is correct!");
```

Many control constructs and use of  
{ } identical to Java/C++

# BASIC JAVASCRIPT SYNTAX

```
while (guess != thinkingOf)
{
```

```
    // Evaluate the user's guess
```

```
    if (guess < thinkingOf) {
```

```
        guess = window.prompt("Your guess of " + guess +  
                               " was too low.  Guess again.", "");
```

```
    }
```

```
    else {
```

```
        guess = window.prompt("Your guess of " + guess +  
                               " was too high.  Guess again.", "");
```

```
    }
```

```
}
```

```
// Game over; congratulate the user
```

```
window.alert(guess + " is correct!");
```

Most relational operators syntactically  
same as Java/C++

# BASIC JAVASCRIPT SYNTAX

```
while (guess != thinkingOf)
{
    // Evaluate the user's guess
    if (guess < thinkingOf) {
        guess = window.prompt("Your guess of " + guess +
                               " was too low.  Guess again.", "");
    }
    else {
        guess = window.prompt("Your guess of " + guess +
                               " was too high.  Guess again.", "");
    }
}

// Game over; congratulate the user
window.alert(guess + " is correct!");
```

Automatic type conversion:  
guess is String,  
thinkingOf is Number

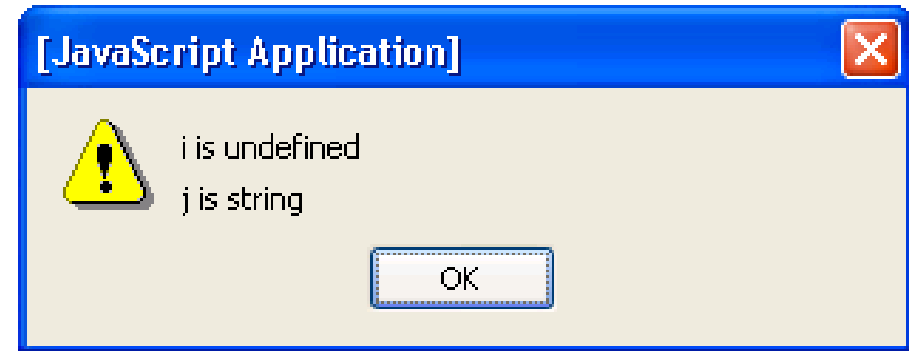
# VARIABLES AND DATA TYPES

- Type of a variable is **dynamic**: depends on the type of data it contains
- JavaScript has six data types:
  - Number
  - String
  - Boolean (values `true` and `false`)
  - Object
  - Null (only value of this type is `null`)
  - Undefined (value of newly created variable)
- **Primitive** data types: all but Object

# VARIABLES AND DATA TYPES

- `typeof` operator returns string related to data type
  - Syntax: `typeof expression`
- Example:

```
// TypeOf.js
var i;
var j;
j = "Not a number";
alert("i is " + (typeof i) + "\n" +
      "j is " + (typeof j));
```



# VARIABLES AND DATA TYPES

TABLE 4.1: Values returned by `typeof` for various operands.

Operand Value	String <code>typeof</code> Returns
<code>null</code>	<code>"object"</code>
<code>Boolean</code>	<code>"boolean"</code>
<code>Number</code>	<code>"number"</code>
<code>String</code>	<code>"string"</code>
native Object representing function	<code>"function"</code>
native Object not representing function	<code>"object"</code>
declared variable with no value	<code>"undefined"</code>
undeclared variable	<code>"undefined"</code>
nonexistent property of an Object	<code>"undefined"</code>

# VARIABLES AND DATA TYPES

- Common automatic type conversions:
  - Compare String and Number: String value converted to Number
  - Condition of `if` or `while` converted to Boolean
  - Array accessor (*e.g.*, 3 in `records[3]`) converted to String

# VARIABLES AND DATA TYPES

TABLE 4.2: Data type conversions to Boolean.

Original Value	Value as Boolean
undefined	false
null	false
0	false
NaN	false
"" (empty string)	false
any other value	true

# VARIABLES AND DATA TYPES

TABLE 4.3: Data type conversions to String.

Original Value	Value as String
undefined	"undefined"
null	"null"
true, false	"true", "false"
NaN	"NaN"
Infinity, -Infinity	"Infinity", "-Infinity"
other Number up to $\approx 20$ digits	integer or decimal representation
Number over $\approx 20$ digits	scientific notation
Object	call to <code>toString()</code> method on the object

# VARIABLES AND DATA TYPES

TABLE 4.3: Data type conversions to String.

Original Value	Value as String
undefined	"undefined"
null	"null"
true, false	"true", "false"
NaN	"NaN"
Infinity, -Infinity	"Infinity", "-Infinity"
other Number up to $\approx 20$ digits	integer or decimal representation
Number over $\approx 20$ digits	scientific notation
Object	call to <code>toString()</code> method on the object

Special Number values ("Not a Number" and number too large to represent)

# VARIABLES AND DATA TYPES

TABLE 4.4: Data type conversions to Number.

Original Value	Value as Number
undefined	NaN
null, false, "" (empty string)	0
true	1
String representing number	represented number
other String	NaN
Object	call to <code>valueOf()</code> method on the object

# VARIABLES AND DATA TYPES

- Syntax rules for names (**identifiers**):
  - Must begin with letter or underscore ( \_ )
  - Must contain only letters, underscores, and digits (or certain other characters)
  - Must not be a reserved word

# VARIABLES AND DATA TYPES


abstract	boolean	break	byte	case	catch
char	class	const	continue	debugger	default
delete	do	double	else	enum	export
extends	false	final	finally	float	for
function	goto	if	implements	import	in
instanceof	int	interface	long	native	new
null	package	private	protected	public	return
short	static	super	switch	synchronized	
this	throw	throws	transient	true	try
typeof	var	void	volatile	while	with

FIGURE 4.6: JavaScript reserved words.

# VARIABLES AND DATA TYPES

- A variable will automatically be created if a value is assigned to an undeclared identifier:
- **Recommendation:** declare all variables
  - Facilitates maintenance
  - Avoids certain exceptions

var is not  
required



```
testing = "Does this work?";  
window.alert(testing);
```

# JAVASCRIPT STATEMENTS

- **Expression statement:** any statement that consists entirely of an expression
  - **Expression:** code that represents a value
- **Block statement:** one or more statements enclosed in { } braces
- **Keyword statement:** state `i = 5;`  
`j++;` beginning with a keyword, *e.g.*, `var` or `if`

# JAVASCRIPT STATEMENTS

- `var` syntax:

```
var i, msg="hi", o=null;
```

- Java-like keyword statements:

Comma-separated declaration list with optional initializers

TABLE 4.5: JavaScript keyword statements.

Statement Name	Syntax
if-then	<code>if (expr) stmt</code>
if-then-else	<code>if (expr) stmt else stmt</code>
do	<code>do stmt while (expr)</code>
while	<code>while (expr) stmt</code>
for	<code>for (part1 ; part2 ; part3) stmt</code>
continue	<code>continue</code>
break	<code>break</code>
return-void	<code>return</code>
return-value	<code>return expr</code>
switch	<code>switch (expr) { cases }</code>
try	<code>try try-block catch-part</code>
throw	<code>throw expr</code>

# JAVASCRIPT STATEMENTS

JavaScript  
keyword  
statements  
are very similar  
to Java with  
small exceptions

```
// Can use 'var' to define a loop variable inside a 'for'
for (var i=1; i<=3; i++) {

    switch (i) {

        // 'case' value can be any expression and data type,
        // not just constant int as in Java. Automatic
        // type conversion is performed if needed.
        case 1.0 + 2:
            window.alert("i = " + i);
            break;
        default:
            try {
                throw("A JavaScript exception can be anything");
                window.alert("This is not executed.");
            }
            // Do not supply exception data type in 'catch'
            catch (e) {
                window.alert("Caught: " + e);
            }
            break;
    }
}
```

# JAVASCRIPT STATEMENTS

```
// Can use 'var' to define a loop variable inside a 'for'  
for (var i=1; i<=3; i++) {
```

```
    switch (i) {
```

```
        // 'case' value can be any expression and data type,  
        // not just constant int as in Java. Automatic  
        // type conversion is performed if needed.
```

```
        case 1.0 + 2:
```

```
            window.alert("i = " + i);  
            break;
```

```
        default:
```

```
            try {
```

```
                throw("A JavaScript exception can be anything");  
                window.alert("This is not executed.");
```

```
            }
```

```
            // Do not supply exception data type in 'catch'
```

```
            catch (e) {
```

```
                window.alert("Caught: " + e);
```

```
            }
```

```
            break;
```

```
        }
```

```
    }
```

# JAVASCRIPT STATEMENTS

```
// Can use 'var' to define a loop variable inside a 'for'  
for (var i=1; i<=3; i++) {
```

```
    switch (i) {
```

```
        // 'case' value can be any expression and data type,  
        // not just constant int as in Java. Automatic  
        // type conversion is performed if needed.
```

```
        case 1.0 + 2:
```

```
            window.alert("i = " + i);  
            break;
```

```
        default:
```

```
            try {  
                throw("A JavaScript exception can be anything");  
                window.alert("This is not executed.");  
            }
```

```
            // Do not supply exception data type in 'catch'
```

```
            catch (e) {  
                window.alert("Caught: " + e);  
            }  
            break;
```

```
    }
```

```
}
```

# JAVASCRIPT STATEMENTS

```
// Can use 'var' to define a loop variable inside a 'for'
for (var i=1; i<=3; i++) {

    switch (i) {

        // 'case' value can be any expression and data type,
        // not just constant int as in Java. Automatic
        // type conversion is performed if needed.
        case 1.0 + 2:
            window.alert("i = " + i);
            break;
        default:
            try {
                throw("A JavaScript exception can be anything");
                window.alert("This is not executed.");
            }
            // Do not supply exception data type in 'catch'
            catch (e) {
                window.alert("Caught: " + e);
            }
            break;
    }
}
```

# JAVASCRIPT OPERATORS

- **Operators** are used to create **compound expressions** from simpler expressions
- Operators can be classified according to the number of **operands** involved:
  - **Unary**: one operand (*e.g.*, `typeof i`)
    - **Prefix** or **postfix** (*e.g.*, `++i` or `i++`)
  - **Binary**: two operands (*e.g.*, `x + y`)
  - **Ternary**: three operands (**conditional operator**)

```
(debugLevel>2 ? details : "")
```

# JAVASCRIPT OPERATORS

TABLE 4.6: Precedence (high to low) for selected JavaScript operators.

Operator Category	Operators
Object Creation	<code>new</code>
Postfix Unary	<code>++, --</code>
Prefix Unary	<code>delete, typeof, ++, --, +, -, ~, !</code>
Multiplicative	<code>*, /, %</code>
Additive	<code>+, -</code>
Shift	<code>&lt;&lt;, &gt;&gt;, &gt;&gt;&gt;</code>
Relational	<code>&lt;, &gt;, &lt;=, &gt;=</code>
(In)equality	<code>==, !=, ===, !==</code>
Bitwise AND	<code>&amp;</code>
Bitwise XOR	<code>^</code>
Bitwise OR	<code> </code>
Logical AND	<code>&amp;&amp;</code>
Logical OR	<code>  </code>
Conditional and Assignment	<code>?:, =, *=, /=, %=, +=, -=, &lt;&lt;=, &gt;&gt;=, &gt;&gt;&gt;=, &amp;=, ^=,  =</code>

# JAVASCRIPT OPERATORS

- Associativity:
  - Assignment, conditional, and prefix unary operators are **right associative**: equal-precedence operators are evaluated right-to-left:
  - Other operators are **left associative**: equal-precedence operators are evaluated left-to-right

`a *= b += c`  $\longleftrightarrow$  `a *= (b += c)`

# JAVASCRIPT OPERATORS: AUTOMATIC TYPE CONVERSION

- Binary operators `+`, `-`, `*`, `/`, `%` convert both operands to Number
  - Exception: If **one** of operands of `+` is **String** then the other is converted to String
- Relational operators `<`, `>`, `<=`, `>=` convert both operands to Number
  - Exception: If **both** operands are **String**, no conversion is performed and **lexicographic string comparison** is performed

# JAVASCRIPT OPERATORS: AUTOMATIC TYPE CONVERSION

- Operators `==`, `!=` convert both operands to Number
  - Exception: If both operands are String, no conversion is performed (lex. comparison)
  - Exception: values of Undefined and Null are equal(!)
  - Exception: instance of Date built-in “class” is converted to String (and host object conversion is implementation dependent)
  - Exception: two Objects are equal only if they are references to the same object

# JAVASCRIPT OPERATORS: AUTOMATIC TYPE CONVERSION

- Operators `===`, `!==` are **strict**:
  - Two operands are `===` only if they are of the **same type** and have the **same value**
  - “Same value” for objects means that the operands are references to the same object
- **Unary** `+`, `-` convert their operand to Number
- Logical `&&`, `||`, `!` convert their operands to Boolean

# JAVASCRIPT NUMBERS

- **Syntactic** representations of Number
  - Integer (42) and decimal (42.0)
  - Scientific notation (-12.4e12)
  - Hexadecimal (0xfa0)
- **Internal** representation
  - Approximately 16 digits of precision
  - Approximate range of magnitudes
    - Smallest:  $10^{-323}$
    - Largest:  $10^{308}$  (Infinity if literal is larger)

# JAVASCRIPT STRINGS

- String literals can be **single- or double-quoted**
- Common **escape characters** within Strings
  - **\n** newline
  - **\"** escaped double quote (also **\'** for single)
  - **\\** escaped backslash
  - **\uxxxx** arbitrary Unicode 16-bit code point (x's are four hex digits)

# JAVASCRIPT FUNCTIONS

- Function declaration syntax

```
function oneTo(high) {  
    return Math.ceil(Math.random()*high);  
}
```

# JAVASCRIPT FUNCTIONS

- Function declaration syntax

Declaration  
always begins  
with keyword  
function,  
no return type

```
function oneTo(high) {  
    return Math.ceil(Math.random()*high);  
}
```

# JAVASCRIPT FUNCTIONS

- Function declaration syntax

Identifier representing  
function's *name*

```
function oneTo(high) {  
    return Math.ceil(Math.random()*high);  
}
```

# JAVASCRIPT FUNCTIONS

- Function declaration syntax

*Formal parameter list*

```
function oneTo(high) {  
    return Math.ceil(Math.random()*high);  
}
```

# JAVASCRIPT FUNCTIONS

- Function declaration syntax

```
function oneTo(high) {  
    return Math.ceil(Math.random()*high);  
}
```

One or more statements representing  
*function body*

# JAVASCRIPT FUNCTIONS

- Function call syntax

```
thinkingOf = oneTo(1000);
```

# JAVASCRIPT FUNCTIONS

- Function call syntax

```
thinkingOf = oneTo(1000);
```

Function call is an expression, can be used on right-hand side of assignments, as expression statement, etc.

# JAVASCRIPT FUNCTIONS

- Function call syntax

```
thinkingOf = oneTo(1000);
```

Function name

# JAVASCRIPT FUNCTIONS

- Function call syntax

```
thinkingOf = oneTo(1000);
```

Argument list

# JAVASCRIPT FUNCTIONS

- Function call semantics:

```
function oneTo(high) {  
    return Math.ceil(Math.random()*high);  
}  
thinkingOf = oneTo(1000);
```

# JAVASCRIPT FUNCTIONS

- Function call semantics:

```
function oneTo(high) {  
    return Math.ceil(Math.random()*high);  
}  
thinkingOf = oneTo(1000);
```

Argument value(s)  
associated with corresponding  
formal parameters

# JAVASCRIPT FUNCTIONS

- Function call semantics:

```
function oneTo(high) {  
    return Math.ceil(Math.random()*high);  
}  
thinkingOf = oneTo(1000);
```

Expression(s) in body  
evaluated as if formal  
parameters are variables  
initialized by argument  
values

# JAVASCRIPT FUNCTIONS

- Function call semantics:

```
function oneTo(high) {  
  return Math.ceil(Math.random()*high);  
}
```

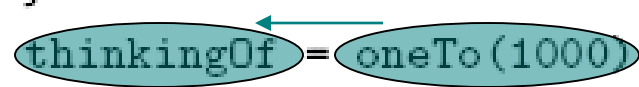
```
thinkingOf = oneTo(1000);
```

If final statement executed  
is return-value, then value of  
its expression becomes value  
of the function call

# JAVASCRIPT FUNCTIONS

- Function call semantics:

```
function oneTo(high) {  
    return Math.ceil(Math.random()*high);  
}  
thinkingOf = oneTo(1000);
```

A diagram illustrating function call semantics. It shows a function definition for 'oneTo' and a call to it. The variable 'thinkingOf' is assigned the result of 'oneTo(1000)'. Both 'thinkingOf' and 'oneTo(1000)' are enclosed in light blue ovals. A light blue arrow points from the 'oneTo(1000)' oval to the 'thinkingOf' oval, indicating that the value of the function call is being used to assign a value to the variable.

Value of function call is then used  
in larger expression containing  
function call.

# JAVASCRIPT FUNCTIONS

- Function call semantics details:
  - Arguments:
    - May be expressions:
    - Object's effectively passed by reference `oneTo(999+1)`,
  - Formal parameters:
    - May be assigned values, argument is not affected
  - Return value:
    - If last statement executed is not return-value, then returned value is of type Undefined

# JAVASCRIPT FUNCTIONS

- **Number mismatch** between argument list and formal parameter list:
  - **More arguments**: excess ignored
  - **Fewer arguments**: remaining parameters are Undefined

# JAVASCRIPT FUNCTIONS

- Local vs. global variables

Global variable: declared outside any function

```
var j=6; // global variable declaration and initialization
function test()
{
    var j; // local variable declaration
    j=7;    // Which variable(s) does this change?
    return;
}
test();
window.alert(j);
```

# JAVASCRIPT FUNCTIONS

- Local vs. global variables

Local  
variable  
declared  
within  
a function

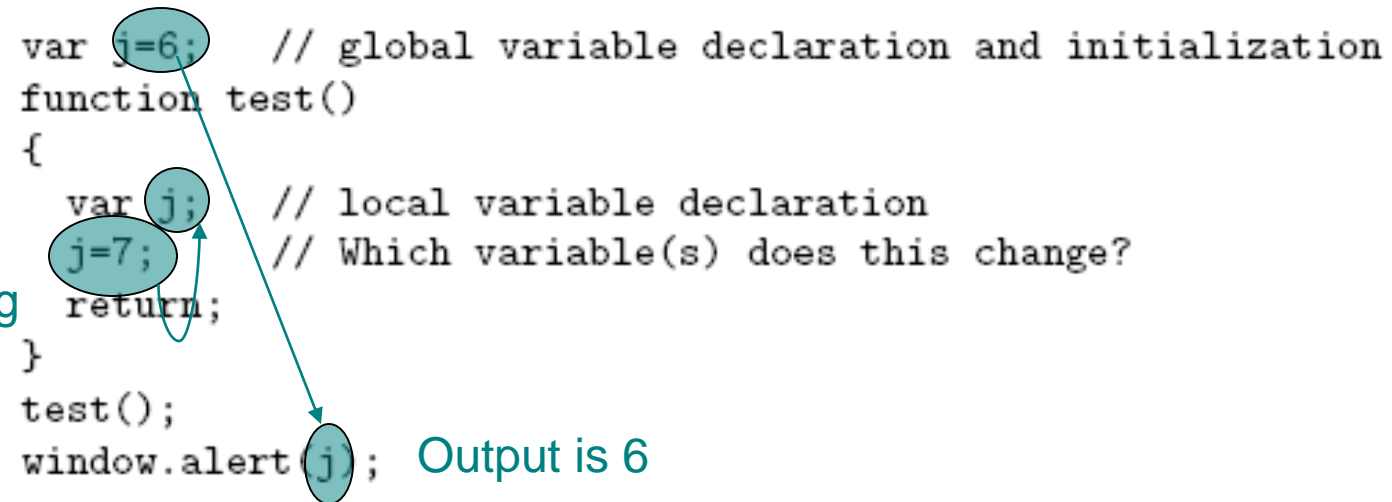
```
var j=6;    // global variable declaration and initialization
function test()
{
  var j;    // local variable declaration
  j=7;      // Which variable(s) does this change?
  return;
}
test();
window.alert(j);
```

# JAVASCRIPT FUNCTIONS

- Local vs. global variables

Local  
declaration  
shadows  
corresponding  
global  
declaration

```
var j=6; // global variable declaration and initialization
function test()
{
  var j; // local variable declaration
  j=7; // Which variable(s) does this change?
  return;
}
test();
window.alert(j); Output is 6
```



# JAVASCRIPT FUNCTIONS

- Local vs. global variables

```
var j=6;    // global variable declaration and initialization
function test()
{
    var j;   // local variable declaration
    window.j = 7; // Which variable(s) does this change?
    return;
}
test();
window.alert(j);
```

Output is 7

In browsers, global variables (and functions) are stored as properties of the window built-in object.

# JAVASCRIPT FUNCTIONS

- Recursive functions
  - **Recursion** (function calling itself, either directly or indirectly) is supported
  - C++ **static variables** are not supported
  - Order of declaration of mutually recursive functions is unimportant (no need for prototypes as in C++)

# JAVASCRIPT FUNCTIONS

- Explicit **type conversion** supplied by built-in functions
  - Boolean(), String(), Number()
  - Each takes a single argument, returns value representing argument converted according to type-conversion rules given earlier

# OBJECT INTRODUCTION

- An **object** is a set of **properties**
- A **property** consists of a unique (within an object) **name** with an associated **value**
- The type of a property depends on the type of its value and can vary dynamically

<code>o.prop = true;</code>	prop is Boolean
<code>o.prop = "true";</code>	prop is now String
<code>o.prop = 1;</code>	prop is now Number

# OBJECT INTRODUCTION

- There are **no classes** in JavaScript
- Instead, properties can be created and deleted dynamically

```
var o1 = new Object();  
o1.testing = "This is a test";  
delete o1.testing;
```

Create an object o1

Create property testing

Delete testing property

# OBJECT CREATION

- Objects are created using **new** expression
- A **constructor** is a function *new Object()* *Constructor and argument list*
  - When called via **new** expression, a new empty Object is created and passed to the constructor along with the argument values
  - Constructor performs initialization on object
    - Can add **properties** and **methods** to object
    - Can add object to an **inheritance hierarchy**

# OBJECT CREATION

- The `Object()` built-in constructor
  - Does not add any properties or methods directly to the object
  - Adds object to hierarchy that defines default `toString()` and `valueOf()` methods (used for conversions to `String` and `Number`, resp.)

# PROPERTY CREATION

- **Assignment** to a non-existent (even if inherited) property name creates the property:
- **Object initializer** notation can be used to create an object (using `Object()` constructor), and one or more properties in a single statement:  

```
o1.testing = "This is a test";
```

```
var o2 = { p1:5+9, p2:null, testing:"This is a test" };
```

# ENUMERATING PROPERTIES

- Special form of **for** statement used to **iterate through all properties** of an object:

```
var hash = new Object();  
hash.kim = "85";  
hash.sam = "92";  
hash.lynn = "78";  
for (var aName in hash) {  
    window.alert(aName + " is a property of hash.");  
}
```

Produces three alert boxes;  
order of names is implementation-dependent.

# ACCESSING PROPERTY VALUES

- The JavaScript object dot notation is actually shorthand for a more general **associative array** notation in which Strings are array indices:
- Expressions can supply property names:

`hash.kim`  $\longrightarrow$  `hash["kim"]`

`window.alert(aName + " scored " + hash[aName]);`

Converted to String  
if necessary

# OBJECT VALUES

- Value of Object is reference to object:

```
var o1 = new Object();  
o1.data = "Hello";  
var o2 = o1;  
o2.data += " World!";  
window.alert(o1.data);
```

# OBJECT VALUES

- Value of Object is reference to object:

o2 is another  
name for o1

```
var o1 = new Object();  
o1.data = "Hello";  
var o2 = o1;  
o2.data += " World!";  
window.alert(o1.data);
```

# OBJECT VALUES

- Value of Object is reference to object:

```
var o1 = new Object();  
o1.data = "Hello";  
var o2 = o1;  
o2.data += " World!";  
window.alert(o1.data);
```

o1 is  
changed

# OBJECT VALUES

- Value of Object is reference to object:

```
var o1 = new Object();  
o1.data = "Hello";  
var o2 = o1;  
o2.data += " World!";  
window.alert(o1.data);
```

Output is Hello World!

# OBJECT VALUES

- Object argument values are references

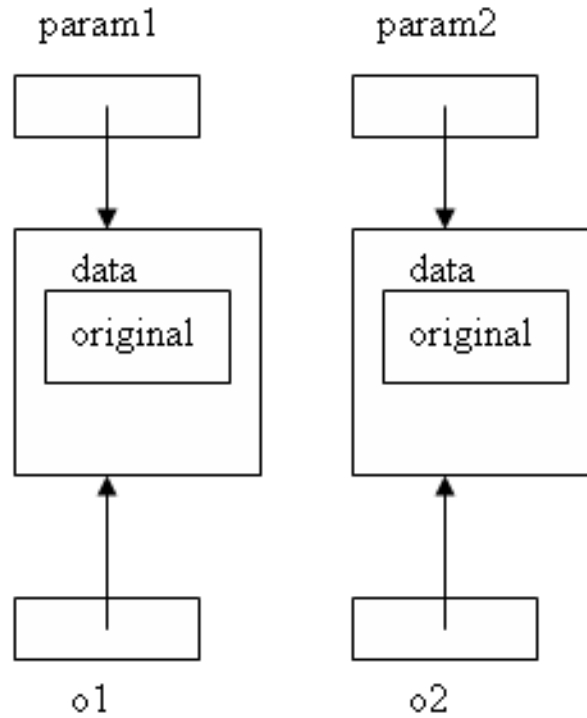
```
// Create two different objects with identical data
var o1 = new Object();
o1.data = "original";
var o2 = new Object();
o2.data = "original";

// Call the function on these objects and display the results
objArgs(o1, o2);

function objArgs(param1, param2) { ...}
```

# OBJECT VALUES

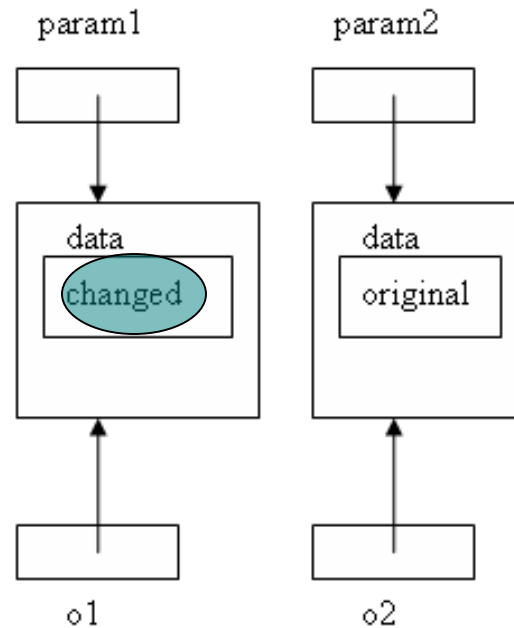
- Object argument values are references



# OBJECT VALUES

- Object argument values are references

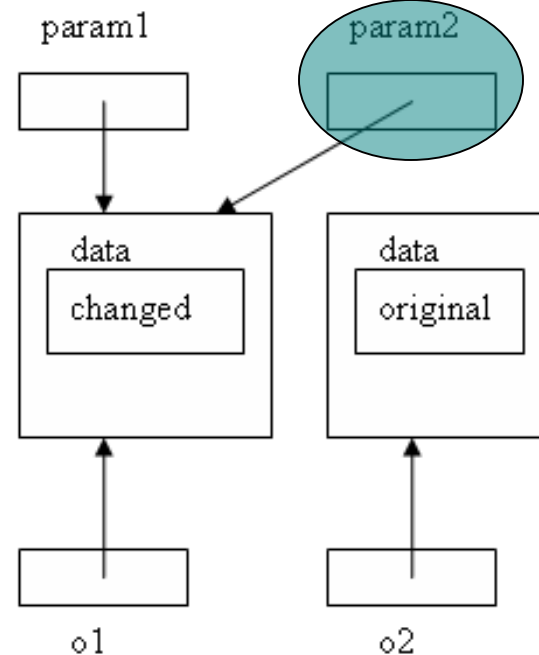
```
function objArgs(param1, param2) {  
  // Change the data in param1 and its argument  
  param1.data = "changed";  
}
```



# OBJECT VALUES

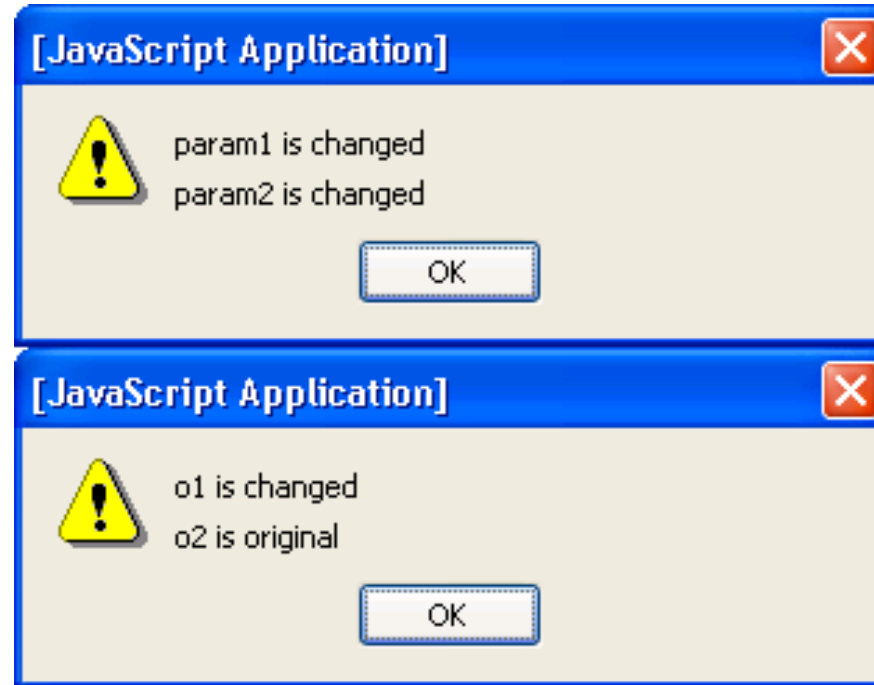
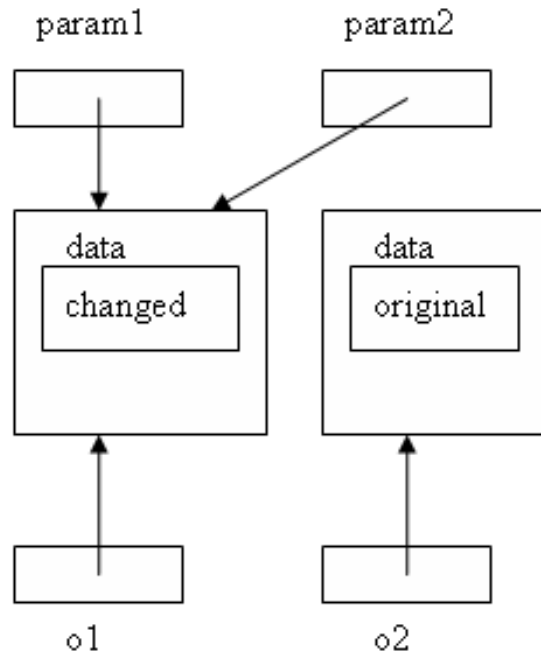
- Object argument values are references

```
function objArgs(param1, param2) {  
  // Change the data in param1 and its argument  
  param1.data = "changed";  
  // Change the object referenced by param2, but not its argument  
  param2 = param1;  
}
```



# OBJECT VALUES

- Object argument values are references



# OBJECT METHODS

- JavaScript **functions are stored as values** of type Object
- A function declaration creates a function value and stores it in a variable (property of `window`) having the same name as the function
- A **method** is an object property for which the value is a function

# OBJECT METHODS

```
function leaf() {  
    return this.left == null && this.right == null;  
}  
  
function makeBTNode(value) {  
    var node = new Object();  
    node.left = node.right = null;  
    node.value = value;  
    node.isLeaf = leaf;  
    return node;  
}
```

# OBJECT METHODS

Creates global variable named `leaf` with function value

```
function leaf() {  
    return this.left == null && this.right == null;  
}  
  
function makeBTNode(value) {  
    var node = new Object();  
    node.left = node.right = null;  
    node.value = value;  
    node.isLeaf = leaf;  
    return node;  
}
```

# OBJECT METHODS

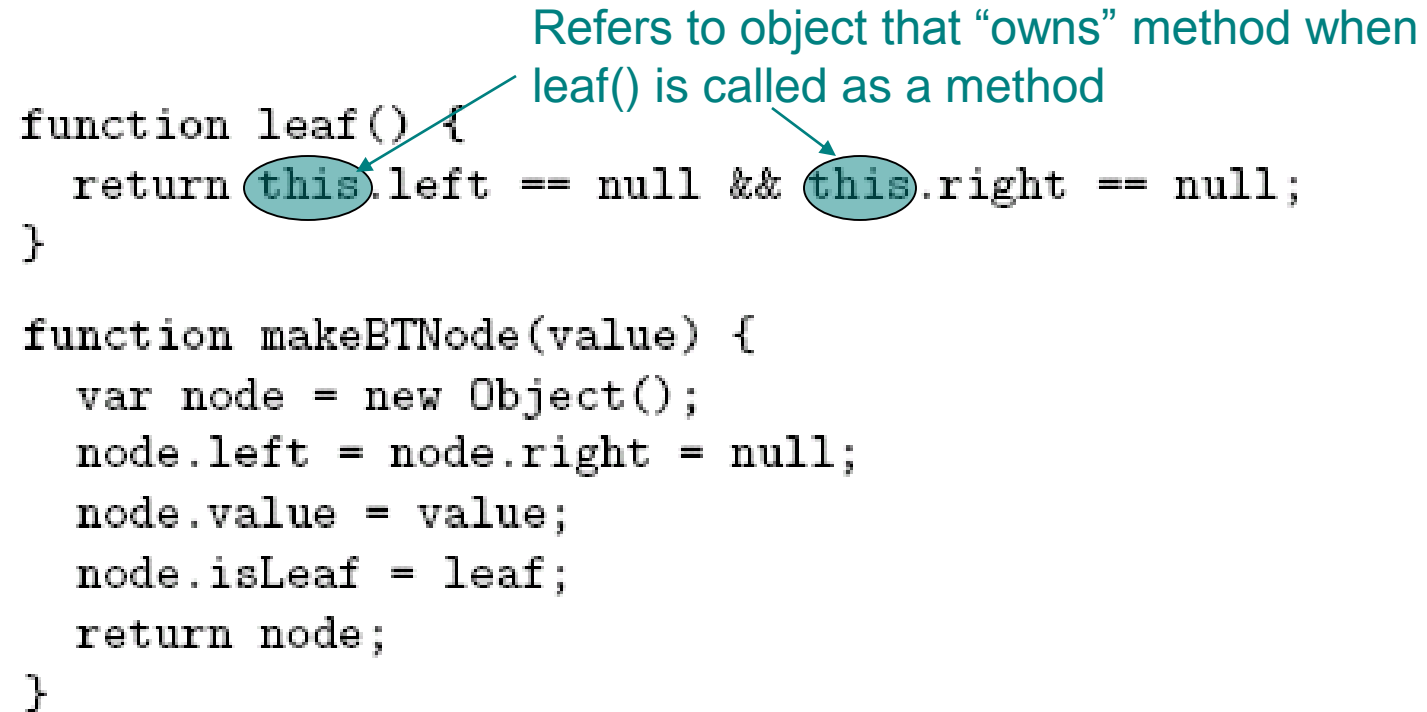
```
function leaf() {  
    return this.left == null && this.right == null;  
}  
  
function makeBTNode(value) {  
    var node = new Object();  
    node.left = node.right = null;  
    node.value = value;  
    node.isLeaf = leaf;  
    return node;  
}
```

Creates isLeaf() method that is  
defined by leaf() function

# OBJECT METHODS

Refers to object that “owns” method when  
leaf() is called as a method

```
function leaf() {  
    return this.left == null && this.right == null;  
}  
  
function makeBTNode(value) {  
    var node = new Object();  
    node.left = node.right = null;  
    node.value = value;  
    node.isLeaf = leaf;  
    return node;  
}
```



# OBJECT METHODS

```
var node1 = makeBTNode(3);  
var node2 = makeBTNode(7);  
node1.right = node2;  
  
// Output the value of isLeaf() on each node  
window.alert("node1 is a leaf: " + node1.isLeaf());  
window.alert("node2 is a leaf: " + node2.isLeaf());
```

# OBJECT METHODS

```
var node1 = makeBTNode(3);  
var node2 = makeBTNode(7);  
node1.right = node2;
```

Creates two objects each with  
method `isLeaf()`

```
// Output the value of isLeaf() on each node  
window.alert("node1 is a leaf: " + node1.isLeaf());  
window.alert("node2 is a leaf: " + node2.isLeaf());
```

# OBJECT METHODS

```
var node1 = makeBTNode(3);  
var node2 = makeBTNode(7);  
node1.right = node2;  
  
// Output the value of isLeaf() on each node  
window.alert("node1 is a leaf: " + node1.isLeaf());  
window.alert("node2 is a leaf: " + node2.isLeaf());
```

Calls to isLeaf() method

# OBJECT METHODS

- Original version: `leaf()` can be called as function, but we only want a method

```
function leaf() {  
    return this.left == null && this.right == null;  
}  
  
function makeBTNode(value) {  
    var node = new Object();  
    node.left = node.right = null;  
    node.value = value;  
    node.isLeaf = leaf;  
    return node;  
}
```

# OBJECT METHODS

- Alternative:

```
function leaf() {  
    return this.left == null && this.right == null;  
}
```

```
function makeBTNode(value) {  
    var node = new Object();  
    node.left = node.right = null;  
    node.value = value;  
    node.isLeaf = leaf;  
    return node;  
}
```

*Function expression syntactically the same as function declaration but does not produce a global variable.*

# OBJECT METHODS

- Alternative

```
function makeBTNode(value) {  
    var node = new Object();  
    node.left = node.right = null;  
    node.value = value;  
  
    node.isLeaf =  
        function leaf() {  
            return this.left == null && this.right == null;  
        };  
  
    return node;  
}
```

# OBJECT CONSTRUCTORS

- User-defined constructor is just a function called using **new** expression:
- Object created by constructor is known as an **instance** of the class

```
var node1 = new BTreeNode(3);  
var node2 = new BTreeNode(7);
```

Constructor

# OBJECT CONSTRUCTORS

Original  
function

```
function makeBTNode(value) {  
    var node = new Object();  
    node.left = node.right = null;  
    node.value = value;  
    node.isLeaf =  
        function leaf() {  
            return this.left == null && this.right == null;  
        };  
    return node;  
}
```

Function  
intended  
to be used  
as constructor

```
function BTNode(value) {  
    this.left = this.right = null;  
    this.value = value;  
    this.isLeaf =  
        function leaf() {  
            return this.left == null && this.right == null;  
        };  
}
```

# OBJECT CONSTRUCTORS

```
function makeBTNode(value) {  
    var node = new Object();  
    node.left = node.right = null;  
    node.value = value;  
    node.isLeaf =  
        function leaf() {  
            return this.left == null && this.right == null;  
        };  
    return node;  
}  
  
function BTNode(value) {  
    this.left = this.right = null;  
    this.value = value;  
    this.isLeaf =  
        function leaf() {  
            return this.left == null && this.right == null;  
        };  
}
```

Object is  
constructed  
automatically  
by new  
expression

# OBJECT CONSTRUCTORS

```
function makeBTNode(value) {  
    var node = new Object();  
    node.left = node.right = null;  
    node.value = value;  
    node.isLeaf =  
        function leaf() {  
            return this.left == null && this.right == null;  
        };  
    return node;  
}
```

Object  
referenced  
using this  
keyword

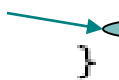
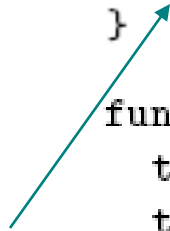
```
function BTNode(value) {  
    this.left = this.right = null;  
    this.value = value;  
    this.isLeaf =  
        function leaf() {  
            return this.left == null && this.right == null;  
        };  
}
```

# OBJECT CONSTRUCTORS

```
function makeBTNode(value) {  
    var node = new Object();  
    node.left = node.right = null;  
    node.value = value;  
    node.isLeaf =  
        function leaf() {  
            return this.left == null && this.right == null;  
        };  
    return node;  
}
```

```
function BTNode(value) {  
    this.left = this.right = null;  
    this.value = value;  
    this.isLeaf =  
        function leaf() {  
            return this.left == null && this.right == null;  
        };  
}
```

No need  
to return  
initialized  
object



# OBJECT CONSTRUCTORS

- Object created using a constructor is known as an **instance** of the constructor

```
var node1 = new BTNode(3);
```

```
var node2 = new BTNode(7);
```

- **instanceof** operator can be used to test this relationship:

Instances of BTNode

```
window.alert("node1 is instance of BTNode: " +  
              (node1 instanceof BTNode));
```

Evaluates to true

# JAVASCRIPT ARRAYS

- The **Array** built-in object can be used to construct objects with special properties and that inherit various methods

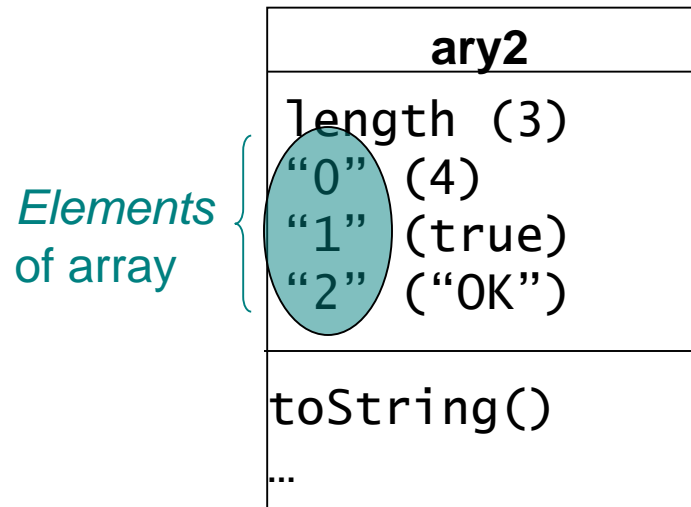
```
var ary1 = new Array();
```

ary1	
length (0)	Properties
toString() sort() shift() ...	Inherited methods

# JAVASCRIPT ARRAYS

- The **Array** built-in object can be used to construct objects with special properties and that inherit various methods

```
var ary2 = new Array(4, true, "OK");
```



Accessing array elements:

✓ ary2[1]

✓ ary2["1"]

✗ ary2.1

Must follow identifier  
syntax rules

# JAVASCRIPT ARRAYS

- The Array constructor is indirectly called if an **array initializer** is used

- Array initializers can be used to create **multidimensional arrays**

```
var ary2 = new Array(4, true, "OK");
```



```
var ary3 = [4, true, "OK"];
```

```
var ttt = [ [ "X", "0", "0" ],  
            [ "0", "X", "0" ],  
            [ "0", "X", "X" ] ];
```

ttt[1][2]

# JAVASCRIPT ARRAYS

- Changing the number of elements:

```
var ary2 = new Array(4, true, "OK");
```

```
ary2[3] = -12.6;
```

Creates a new element dynamically,  
increases value of length

ary2	
length	(4)
"0"	(4)
"1"	(true)
"2"	("OK")
"3"	(-12.6)
toString()	
...	

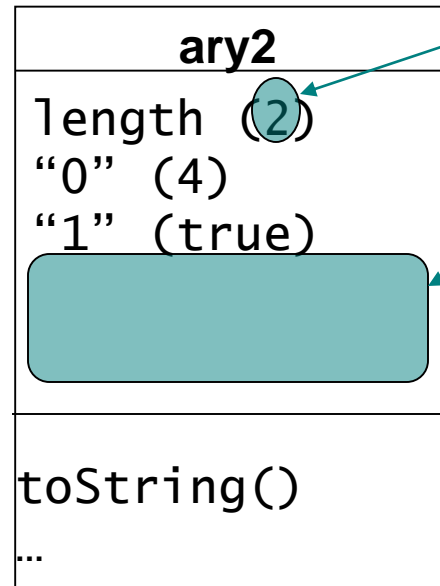
# JAVASCRIPT ARRAYS

- Changing the number of elements:

```
var ary2 = new Array(4, true, "OK");  
ary2[3] = -12.6;
```

```
ary2.length = 2;
```

Decreasing length can delete elements



# JAVASCRIPT ARRAYS

- Value of `length` is not necessarily the same as the actual number of elements

`var ary4 = new Array(200);` Calling constructor with single argument sets `length`, does not create elements

ary4
length (200)
toString() sort() shift() ...

# JAVASCRIPT ARRAYS

TABLE 4.7: Methods inherited by array objects. Unless otherwise specified, methods return a reference to the array on which they are called.

Method	Description
<code>toString()</code>	Return a String value representing this array as a comma-separated list.
<code>sort(Object)</code>	Modify this array by sorting it, treating the Object argument as a function that specifies sort order (see below).
<code>splice(Number, 0, any type)</code>	Modify this array by adding the third argument as an element at the index given by the first argument, “shifting” elements up one index to make room for the new element.
<code>splice(Number, Number)</code>	Modify this array by removing a number of elements specified by the second argument (a positive integer), starting with the index specified by the first element, “shifting” elements down to take the place of those elements removed. Returns an array of the elements removed.
<code>push(any type)</code>	Modify this array by appending an element having the given argument value. Returns <b>length</b> value for modified array.
<code>pop()</code>	Modify this array by removing its last element (the element at index <b>length</b> – 1). Returns the value of the element removed.
<code>shift()</code>	Modify this array by removing its first element (the element at index 0) and “shifting” all remaining elements down one index. Returns the value of the element removed.

# JAVASCRIPT ARRAYS

```
// ArrayMethods.js
var numArray = [1,3,8,4,9,7,6,2,5];

// Sort in ascending order
numArray.sort(
  function compare (first, second) {
    return first - second;
  }
);
// numArray.toString(): 1,2,3,4,5,6,7,8,9

numArray.splice(2, 0, 2.5);
// numArray.toString(): 1,2,2.5,3,4,5,6,7,8,9

// output of following: 5,6,7
window.alert(numArray.splice(5,3).toString());
// numArray.toString(): 1,2,2.5,3,4,8,9
window.alert(numArray.toString());
```

# JAVASCRIPT ARRAYS

Argument to sort  
is a function

```
// ArrayMethods.js
var numArray = [1,3,8,4,9,7,6,2,5];

// Sort in ascending order
numArray.sort(
  { function compare (first, second) {
    return first - second;
  }
});
// numArray.toString(): 1,2,3,4,5,6,7,8,9

numArray.splice(2, 0, 2.5);
// numArray.toString(): 1,2,2.5,3,4,5,6,7,8,9

// output of following: 5,6,7
window.alert(numArray.splice(5,3).toString());
// numArray.toString(): 1,2,2.5,3,4,8,9
window.alert(numArray.toString());
```

# JAVASCRIPT ARRAYS

```
// ArrayMethods.js
var numArray = [1,3,8,4,9,7,6,2,5];

// Sort in ascending order
numArray.sort(
  function compare (first, second) {
    return first - second;
  }
);
// numArray.toString(): 1,2,3,4,5,6,7,8,9

numArray.splice(2, 0, 2.5);
// numArray.toString(): 1,2,2.5,3,4,5,6,7,8,9

// output of following: 5,6,7
window.alert(numArray.splice(5,3).toString());
// numArray.toString(): 1,2,2.5,3,4,8,9
window.alert(numArray.toString());
```

Return negative if first value should come before second after sorting

# JAVASCRIPT ARRAYS

```
// ArrayMethods.js
var numArray = [1,3,8,4,9,7,6,2,5];

// Sort in ascending order
numArray.sort(
  function compare (first, second) {
    return first - second;
  }
);
// numArray.toString(): 1,2,3,4,5,6,7,8,9
```

```
numArray.splice(2, 0, 2.5);
```

Add element with value 2.5 at index 2, shift existing elements

```
// numArray.toString(): 1,2,2.5,3,4,5,6,7,8,9
```

```
// output of following: 5,6,7
window.alert(numArray.splice(5,3).toString());
// numArray.toString(): 1,2,2.5,3,4,8,9
window.alert(numArray.toString());
```

# JAVASCRIPT ARRAYS

```
// ArrayMethods.js
var numArray = [1,3,8,4,9,7,6,2,5];

// Sort in ascending order
numArray.sort(
  function compare (first, second) {
    return first - second;
  }
);
// numArray.toString(): 1,2,3,4,5,6,7,8,9
```

```
numArray.splice(2, 0, 2.5);
// numArray.toString(): 1,2,2.5,3,4,5,6,7,8,9
```

```
// output of following: 5,6,7
window.alert(numArray.splice(5,3).toString());
// numArray.toString(): 1,2,2.5,3,4,8,9
window.alert(numArray.toString());
```

Remove 3 elements starting at index 5

# JAVASCRIPT ARRAYS

```
// stack.js
var stack = new Array();
stack.push('H');
stack.push('i');
stack.push('!');

var c3 = stack.pop(); // pops '!'
var c2 = stack.pop(); // pops 'i'
var c1 = stack.pop(); // pops 'H'
window.alert(c1 + c2 + c3); // displays "Hi!"
```

# JAVASCRIPT ARRAYS

```
// stack.js
var stack = new Array();
stack.push('H');  push() adds an element to the end of the
stack.push('i');  array
stack.push('!');

var c3 = stack.pop(); // pops '!'
var c2 = stack.pop(); // pops 'i'
var c1 = stack.pop(); // pops 'H'
window.alert(c1 + c2 + c3); // displays "Hi!"
```

# JAVASCRIPT ARRAYS

```
// stack.js
var stack = new Array();
stack.push('H');
stack.push('i');
stack.push('!');  pop() deletes and returns last
                  element of the array

var c3 = stack.pop(); // pops '!'
var c2 = stack.pop(); // pops 'i'
var c1 = stack.pop(); // pops 'H'
window.alert(c1 + c2 + c3); // displays "Hi!"
```

# JAVASCRIPT ARRAYS

```
// stack.js  
var stack = new Array();  
stack.push('H');  
stack.push('i');  
stack.push('!');
```

Use `shift()` instead to implement queue

```
var c3 = stack.pop(); // pops '!'  
var c2 = stack.pop(); // pops 'i'  
var c1 = stack.pop(); // pops 'H'  
window.alert(c1 + c2 + c3); // displays "Hi!"
```

# BUILT-IN OBJECTS

- The `global object`
  - Named `window` in browsers
  - Has properties representing all global variables
  - Other built-in objects are also properties of the global object
    - Ex: initial value of `window.Array` is `Array` object
  - Has some other useful properties
    - Ex: `window.Infinity` represents `Number` value

# BUILT-IN OBJECTS

- The global object and variable resolution:

`i` = 42;

What does `i` refer to?

1. Search for local variable or formal parameter  
named `i`

2. If none found, see if global object (window)  
has property named `i`

- This is why we can refer to built-in objects (Object, Array, etc.)  
without prefixing with `window`.

# BUILT-IN OBJECTS

- `String()`, `Boolean()`, and `Number()` built-in functions can be called as constructors, created “wrapped” Objects:
- Instances inherit `valueOf()` method that returns wrapped value of specified type:

```
var wrappedNumber = new Number(5.625);
```

```
window.alert(typeof wrappedNumber.valueOf());
```

Output is “number”

# BUILT-IN OBJECTS

- Other methods inherited by `Number` instances:

<code>var wrappedNumber = new Number(5.625);</code>	<u>Outputs</u>
<code>window.alert(wrappedNumber.toFixed(2));</code>	5.63
<code>window.alert(wrappedNumber.toExponential(2));</code>	5.63e+0
<code>window.alert(wrappedNumber.toString(2));</code>	101.101
Base 2	

# BUILT-IN OBJECTS

- Properties provided by `Number` built-in object:
  - `Number.MIN_VALUE`: smallest (absolute value) possible JavaScript Number value
  - `Number.MAX_VALUE`: largest possible JavaScript Number value

# BUILT-IN OBJECTS

TABLE 4.8: Some of the methods inherited by `String` instances.

Method	Description
<code>charAt(Number)</code>	Return string consisting of single character at position (0-based) <code>Number</code> within this string.
<code>concat(String)</code>	Return concatenation of this string to <code>String</code> argument.
<code>indexOf(String, Number)</code>	Return location of leftmost occurrence of <code>String</code> within this string at or after character <code>Number</code> , or -1 if no occurrence exists.
<code>replace(String, String)</code>	Return string obtained by replacing first occurrence of first <code>String</code> in this string with second <code>String</code> .
<code>slice(Number, Number)</code>	Return substring of this string starting at location given by first <code>Number</code> and ending one character before location given by second <code>Number</code> .
<code>toLowerCase()</code>	Return this string with each character having a Unicode Standard lowercase equivalent replaced by that character.
<code>toUpperCase()</code>	Return this string with each character having a Unicode Standard uppercase equivalent replaced by that character.

# BUILT-IN OBJECTS

- Instances of `String` have a `length` property (number of characters)
- JavaScript automatically wraps a primitive value of type `Number` or `String` if the value is used as an object:

```
window.alert("a String value".slice(2,5));
```

Output is “Str”

# BUILT-IN OBJECTS

- The `Date()` built-in constructor can be used to create `Date` instances that represent the current date and time
- Often used to display local date and/or time in Web pages

```
var now = new Date();
```

- Other methods: `toLocaleDateString()` , `toLocaleTimeString()`, *etc.*

```
window.alert("Current date and time: "  
              + now.toLocaleString());
```

# BUILT-IN OBJECTS

- `valueOf()` method inherited by `Date` instances returns integer representing number of milliseconds since midnight 1/1/1970
- Automatic type conversion allows `Date` instances to be treated as Numbers:

```
var startTime = new Date();  
// Perform some processing  
...  
var endTime = new Date();  
window.alert("Processing required " +  
              (endTime - startTime)/1000 +  
              " seconds.");
```

# BUILT-IN OBJECTS

- `Math` object has methods for performing standard mathematical calculations:

```
Math.sqrt(15.3)
```

- Also has properties with approximate values for standard mathematical quantities, *e.g.*,  $e$  (`Math.E`) and  $\pi$  (`Math.PI`)

# BUILT-IN OBJECTS

TABLE 4.9: Methods of the `Math` built-in object.

Method	Return Value
<code>abs(Number)</code>	Absolute value of Number.
<code>acos(Number)</code>	Arc cosine of Number (treated as radians).
<code>asin(Number)</code>	Arc sine of Number.
<code>atan(Number)</code>	Arc tangent of Number (range $-\text{Math.PI}/2$ to $\text{Math.PI}/2$ ).
<code>atan2(Number, Number)</code>	Arc tangent of first Number divided by second (range $-\text{Math.PI}$ to $\text{Math.PI}$ ).
<code>ceil(Number)</code>	Smallest integer no greater than Number.
<code>cos(Number)</code>	Cosine of Number (in radians).
<code>exp(Number)</code>	$\text{Math.E}$ raised to power Number.
<code>floor(Number)</code>	Largest integer no less than Number.
<code>log(Number)</code>	Natural logarithm of Number.
<code>max(Number, Number, ...)</code>	Maximum of given values.
<code>min(Number, Number, ...)</code>	Minimum of given values.
<code>pow(Number, Number)</code>	First Number raised to power of second Number.
<code>random()</code>	Pseudo-random floating-point number in range 0 to 1.
<code>round(Number)</code>	Nearest integer value to Number.
<code>sin(Number)</code>	Sine of Number.
<code>sqrt(Number)</code>	Square root of Number.
<code>tan(Number)</code>	Tangent of Number.

# JAVASCRIPT REGULAR EXPRESSIONS

- A **regular expression** is a particular representation of a set of strings
  - Ex: JavaScript regular expression representing the set of syntactically-valid US telephone **area codes** (three-digit numbers):
    - `\d` represents the set { “0”, “1”, ..., “9” }
    - Concatenated regular expressions represent the “concatenation” (Cartesian product) of their sets



`\d\d\d`

# JAVASCRIPT REGULAR EXPRESSIONS

- Using regular expressions in JavaScript

```
var acTest = new RegExp("^\\d\\d\\d$");  
if (!acTest.test(areaCode)) {  
    window.alert(areaCode + " is not a valid area code.");  
}
```

# JAVASCRIPT REGULAR EXPRESSIONS

- Using regular expressions in JavaScript

```
var acTest = new RegExp("^\\d\\d\\d$");  
if (!acTest.test(areaCode)) {  
    window.alert(areaCode + " is not a valid area code.");  
}
```

Variable containing string to be tested

# JAVASCRIPT REGULAR EXPRESSIONS

- Using regular expressions in JavaScript

Regular expression as String (must escape \)

```
var acTest = new RegExp("\\d\\d\\d$");  
if (!acTest.test(areaCode)) {  
    window.alert(areaCode + " is not a valid area code.");  
}
```

# JAVASCRIPT REGULAR EXPRESSIONS

- Using regular expressions in JavaScript

Built-in constructor

```
var acTest = new RegExp("^\\d\\d\\d$");  
if (!acTest.test(areaCode)) {  
    window.alert(areaCode + " is not a valid area code.");  
}
```

# JAVASCRIPT REGULAR EXPRESSIONS

- Using regular expressions in JavaScript

```
var acTest = new RegExp("^\\d\\d\\d$");  
if (!acTest.test(areaCode)) {  
    window.alert(areaCode + " is not a valid area code.");  
}
```

Method inherited by RegExp instances:  
returns true if the argument *contains* a  
substring in the set of strings represented by  
the regular expression

# JAVASCRIPT REGULAR EXPRESSIONS

- Using regular expressions in JavaScript

Represents beginning of string

Represents end of string

```
var acTest = new RegExp("^\\d\\d\\d$");  
if (!acTest.test(areaCode)) {  
    window.alert(areaCode + " is not a valid area code.");  
}
```

This expression matches only strings with exactly three digits (no other characters, even white space)

# JAVASCRIPT REGULAR EXPRESSIONS

- Using regular expressions in JavaScript

```
var acTest = new RegExp("\d\d\d");
```

- Alternate syntax:

Represents all strings that *begin*  
with three digits

```
var acTest = /\d\d\d/;
```

*Regular expression literal.*  
Do *not* escape \.

# JAVASCRIPT REGULAR EXPRESSIONS

- Simplest regular expression is any character that is not a **special character**:

`^ $ \ . * + ? ( ) [ ] { } |`

- Ex: `_` is a regular expression representing {“\_”}
- Backslash-escaped special character is also a regular expression
  - Ex: `\$` represents {“\$”}

# JAVASCRIPT REGULAR EXPRESSIONS

- Special character `.` (dot) represents any character except a line terminator
- Several **escape codes** are regular expressions representing sets of chars:

TABLE 4.10: JavaScript multi-character escape codes.

Escape Code	Characters Represented
<code>\d</code>	digit: 0 through 9.
<code>\D</code>	Any character except those matched by <code>\d</code> .
<code>\s</code>	space: any JavaScript white space or line terminator (space, tab, line feed, etc.).
<code>\S</code>	Any character except those matched by <code>\s</code> .
<code>\w</code>	“word” character: any letter (a through z and A through Z), digit (0 through 9), or underscore (_)
<code>\W</code>	Any character except those matched by <code>\w</code> .

# JAVASCRIPT REGULAR EXPRESSIONS

- Three types of operations can be used to combine simple regular expressions into more complex expressions:
  - Concatenation
  - Union (|)
  - Kleene star (\*)
- XML DTD content specification syntax based in part on regular expressions

# JAVASCRIPT REGULAR EXPRESSIONS

- Concatenation

- Example:

String consisting entirely of four characters:

- Digit followed by `^\d\.` `\w$`
    - A `.` followed by
    - A single space followed by
    - Any “word” character

- **Quantifier** shorthand syntax for concatenation:

`\d{3}`  $\longleftrightarrow$  `\d\d\d`

# JAVASCRIPT REGULAR EXPRESSIONS

- Union

- Ex:

Union of set of strings represented by regular expressions

- Set of single-character `\d|\s` s that are either a digit or a space character

- **Character class**: shorthand for union of one or more ranges of characters

- Ex:           set of lower case letters

- Ex:           the `\w` escape code class

`[a-z]`

`[a-zA-Z0-9] | _`

# JAVASCRIPT REGULAR EXPRESSIONS

- Unions of concatenations

- Note that concatenation has higher precedence than union

- **Optional**  $\text{reg\_expression}^{\{3,6\}}$   $\longleftrightarrow$   $\text{reg\_expression}^{\{3,6\}}$

$(+|-)?\backslash d \longleftrightarrow (+|-)\{0,1\}\backslash d$

# JAVASCRIPT REGULAR EXPRESSIONS

- Kleene star

- Ex: any number of digits (including none)
- Ex: `\d*`
  - Strings consisting of only “word” characters
  - String must contain both a digit and a letter (in either order)

`\w*(\d\w*[a-zA-Z] | [a-zA-Z] \w*\d) \w*`

## 5) CODE PRACTICE