

Verifying and Documenting ADTs: Javadoc, Java Assertions and JUnits



Documenting Java Code

- Regular Java comments: `/* ... */`
 - ◆ for programmers who must *read or modify* your code
- “One Liners”: `// ...`
 - ◆ for programmers who must *read or modify* your code
- Javadoc comments: `/** ... */`
 - ◆ for those who must **use** your code

Comments and Documentation

- Already mentioned distinction between comments and documentation
 - ◆ **Comments** embedded in the code
 - for the benefit of you and other programmers
 - ◆ **Documentation** separate from the code
 - for the benefit of anyone who wants to use your code
- Both are essential in software of any size
 - ◆ traditionally completely separate
 - ◆ but Java provides a tool to help bridge this gap...

JavaDoc

- A tool that reads comments and produces documentation from them
 - ◆ Comments must be written in a *particular style*
 - ◆ Documentation generated is limited to a particular style
 - not complete on its own
 - still need user manuals etc...

General Form of a JavaDoc Comment (1/2)

- **The first line** is indented to line up with the code below the comment, and starts with the begin-comment symbol (`/**`) followed by a return
- **Subsequent lines** start with an asterisk (`*`). They are indented an additional space so the asterisks line up. A space separates the asterisk from the descriptive text or tag that follows it
- Insert a **blank comment line** between the description and the list of tags, as shown
- Insert additional blank lines to create "**blocks**" of related tags

```
/**  
 * This is the description part of a doc comment.  
 *  
 * @tag Comment for the tag  
 */
```

General Form of a JavaDoc Comment (2/2)

- **The last line** begins with the end-comment symbol (`*/`) indented so the asterisks line up and followed by a return
 - ◆ Note that the end-comment symbol contains *only a single asterisk* (*)
- Break any doc-comment lines exceeding 80 characters in length, if possible
 - ◆ If you have more than one paragraph in the doc comment, separate the paragraphs with a `<p>` paragraph tag
- Comments can contain **HTML tags** (including links)

First Sentence

- Should be a **summary sentence**, containing a concise but complete description of the API item
- The Javadoc tool copies this first sentence to the appropriate member, class/interface or package summary
- This sentence ends at the first period that is followed by a blank, tab, or line terminator, or at the first tag (as defined below). For example, this first sentence ends at "Prof.":

```
/**  
 * This is a simulation of Prof. Knuth's MIX computer.  
 */
```

- you can work around this by typing an HTML meta-character such as "&" or "<" immediately after the period, such as:

```
/**  
 * This is a simulation of Prof.&nbsp; Knuth's MIX computer.  
 */
```

Tagged Paragraphs

- Tags that can be inserted in comments depend on what is commented:
 - ◆ **class**
 - general description, cross-references, author, version
 - ◆ **field**
 - general description, cross-references
 - ◆ **method**
 - general description, cross-references, parameters, return value
- Tagged Paragraphs begin with **@followed by keyword**
 - ◆ End at the next tag or end of the comment

JavaDoc Comment Syntax

- Required syntax:

```
/**  
 * general comment  
 * @tag1 value for tag1  
 * @tag2 value for tag2  
 * ...  
 * @tagn value for tagn  
 */
```

- Anything that doesn't conform to this syntax is ignored!

Main Tags:

- | | |
|------------------------|----------|
| ◆ author: | @author |
| ◆ version: | @version |
| ◆ cross-reference: | @see |
| ◆ method parameter: | @param |
| ◆ method return value: | @return |

@see (Cross-References)

- **Classes**

ClassName e.g. @**see** Amoeba

- **Fields**

#LocalFieldName e.g. @**see** #number_of_feet

ClassName#FieldName e.g. @**see** Mammal#number_of_feet

- **Methods**

#LocalMethodName e.g. @**see** #sayHello

ClassName#MethodName e.g. @**see** Student#sayHello

ClassName#MethodName(params)
e.g. @**see** Person#sayHello(String)

- **Other docs**

@**see** Java Spec

@author

- May be used for class and interface declarations

```
@author Nikos Armenatzoglou
```

- A documentation comment may contain more than one @author tag

```
@author Nikos Armenatzoglou  
@author Vassilis Christophides
```

@version

- May be used for class and interface declarations

```
@version 493.0.1beta
```

- A documentation comment may contain at most one @version tag

@param

- May be used in comments for method and constructor declarations

@param i The begin index

- Parameters should be documented in the order in which they are declared

@return

- Used to provide a short description of the return value

@return The object at the top of the stack

- May be used in documentation comments for declarations of methods whose result type is not void
- Cannot be used in constructors (constructors do not return anything; only create!!!)

@exception (or @throws)

- May be used in documentation comments for method and constructor declarations

```
@throws IndexOutOfBoundsException The matrix is  
too large
```

- The name of the class followed by a short description of when it is thrown

@deprecated

- Adds a comment indicating that this API should no longer be used (even though it may continue to work).

@deprecated *deprecated-text*

- Javadoc moves the **deprecated-text** ahead of the description, placing it in *italics* and preceding it with a bold warning: "**Deprecated**".
- The first sentence of *deprecated-text* should at least tell the user when the API was deprecated and what to use as a replacement.

JavaDoc Example (1/3)

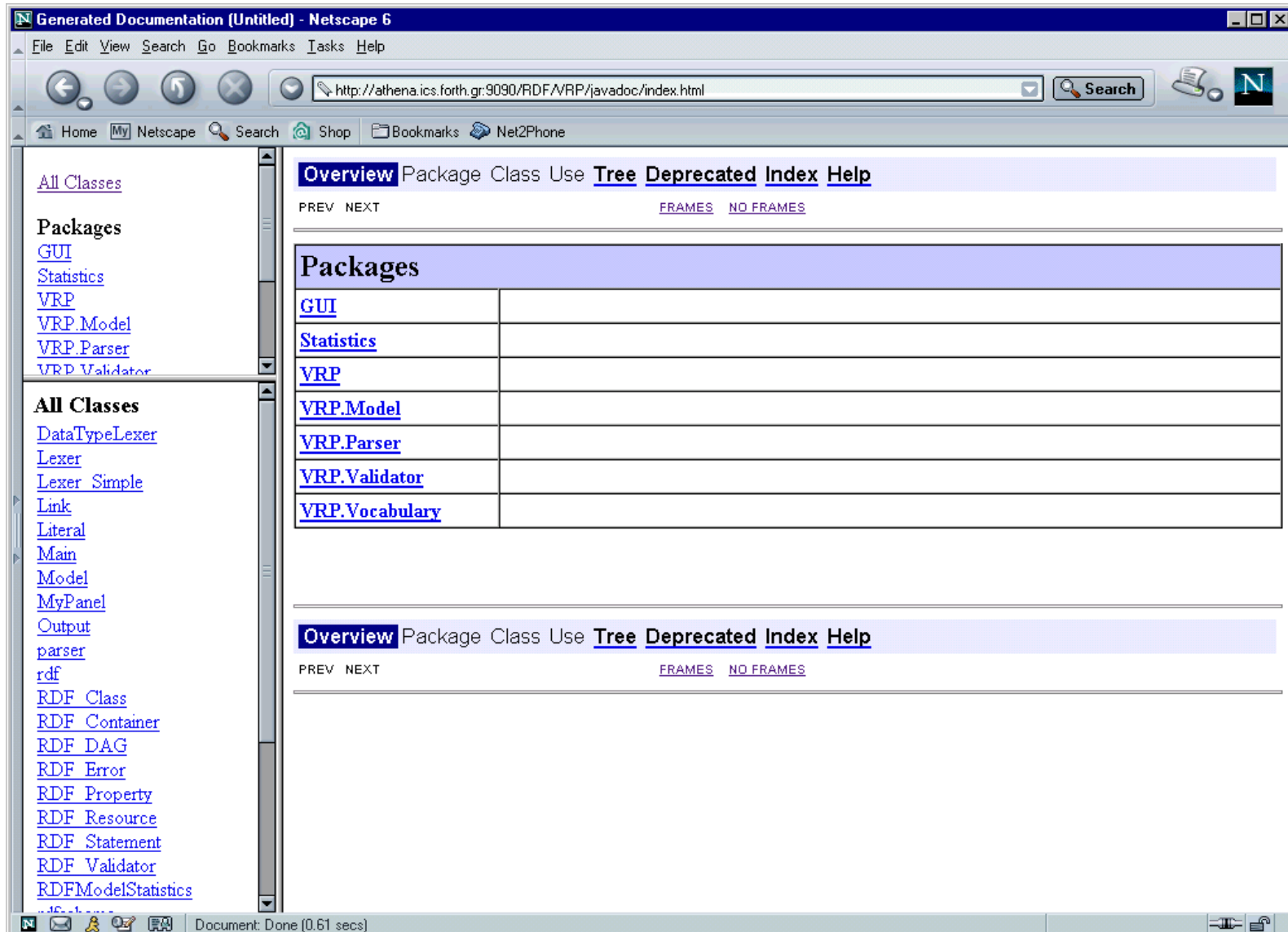
```
/**  
 * The root of the Class hierarchy. Every Class in the  
 * system has Object as its ultimate parent. Every  
 * variable and method defined here is available in every  
 * object.  
 *  
 * @see Class  
 * @version 1.37, 6 Nov 2007  
 */  
  
public class Object {  
    ...  
}
```

JavaDoc Example (2/3)

```
/**
 * Compares two Objects for equality.
 * Returns a boolean that indicates whether this Object
 * is equivalent to the specified Object. This method is
 * used when an Object is stored in a hashtable.
 *
 * @param   obj   The Object to compare with
 * @return  True if these Objects are equal;
 *          false otherwise
 * @see    java.util.Hashtable
 */

public boolean equals (Object obj) {
    return (this == obj);
}
```

JavaDoc Example (3/3)



Generated Documentation (Untitled) - Netscape 6

File Edit View Search Go Bookmarks Tasks Help

http://athena.ics.forth.gr:9090/RDF/VRP/javadoc/index.html

Home My Netscape Search Shop Bookmarks Net2Phone

[All Classes](#)

Packages

[GUI](#)

[Statistics](#)

[VRP](#)

[VRP.Model](#)

[VRP.Parser](#)

[VRP.Validator](#)

All Classes

[DataTypeLexer](#)

[Lexer](#)

[Lexer Simple](#)

[Link](#)

[Literal](#)

[Main](#)

[Model](#)

[MyPanel](#)

[Output](#)

[parser](#)

[rdf](#)

[RDF Class](#)

[RDF Container](#)

[RDF DAG](#)

[RDF Error](#)

[RDF Property](#)

[RDF Resource](#)

[RDF Statement](#)

[RDF Validator](#)

[RDFModelStatistics](#)

Overview [Package](#) [Class](#) [Use](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV](#) [NEXT](#) [FRAMES](#) [NO FRAMES](#)

Packages	
GUI	
Statistics	
VRP	
VRP.Model	
VRP.Parser	
VRP.Validator	
VRP.Vocabulary	

Overview [Package](#) [Class](#) [Use](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV](#) [NEXT](#) [FRAMES](#) [NO FRAMES](#)

Document: Done (0.61 secs)

Running Javadoc

- Synopsis:

```
javadoc [ options ] [ packagenames ] [ sourcefiles ]  
[ @files ]
```

- Example:

- ◆ Directory where the .html output of the javadoc will be stored: api/
- ◆ Directories with source code: src/ and src/folder/

```
javadoc -d api/ src/*.java src/folder/*.java
```

- For further information type “*javadoc*” at your command line...

Style Guidelines

- Be brief
 - ◆ don't repeat things that are obvious from names and types
- Avoid repeating subject
 - ◆ e.g. Gets the name. Not This method gets the name
 - ◆ e.g. The age. Not This field contains the age
- Use 3rd person for methods
 - ◆ e.g. Gets the label. Not Get the label

Adding Preconditions and Postconditions in Javadoc

- In JDK 1.4, we can add preconditions and postconditions as simple Javadoc comments.

```
/**
 * Creates a substring of this string.
 *
 * Precondition:  $0 \leq i < j \leq \text{length}()$ .
 * Postcondition: Returns the substring of this string
 * consisting of the characters whose indexes are  $i, \dots, j-1$ 
 * @param i The begin index
 * @param j The end index
 * @return The substring of this string consisting of the
 * characters whose indexes are  $i, \dots, j-1$ 
 */
public String substring (int i, int j);
```



Assertions

Verifying your code with assertions

- An *assertion* is a statement that enables you to test your assumptions about your program
- SDK 1.4 introduces a new keyword *assert*, which allows the insertion of assertions in a Java program.
- Each assertion contains a boolean expression that you believe will be true when the assertion is executed
 - ◆ If it is not true, the system will throw an error *AssertionError* (subclass of `java.lang.Error`)
- Assertions can be activated during the test phase of the development and deactivated in the production

Syntax

- ***assert assertion;***

where *assertion* is a boolean expression

- ***assert assertion : expression;***

- ◆ *assertion*: a boolean expression
- ◆ *expression*: is an expression that has a **value** (It cannot be an invocation of a method that is declared void)
 - passes the **value** of *expression* to the appropriate **AssertionError** constructor, which uses the string representation of the value as the error's detail message.

Why use assertions?

- By verifying that the boolean expression is indeed true, the assertion confirms your assumptions about the behavior of your program, increasing your confidence that the program is free of errors
- Experience has shown that writing assertions while programming is one of the quickest and most effective ways to detect and correct bugs

Where to use assertions

- **Internal Invariants**

- ◆ assert that an internal invariant has actual *the value that it's expected to have*

- **Control-Flow Invariants**

- ◆ place an assertion at any location you assume *will not be reached*

- **Preconditions, Postconditions, and Class Invariants**

- ◆ it can help support an informal design-by-contract style of programming

Internal Invariants (1/2)

- Programmers are used to write comments to indicate their assumptions concerning a program's behavior, ex.

```
if (i % 3 == 0) {  
    ...  
} else if (i % 3 == 1) {  
    ...  
} else {  
    // we know (i % 3 == 2)  
    ...  
}
```

```
if (i % 3 == 0) {  
    ...  
} else if (i % 3 == 1) {  
    ...  
} else {  
    assert i % 3 == 2 : i;  
    ...  
}
```

Use an assertion whenever you would have written a comment that asserts an invariant

Internal Invariants (2/2)

- Another good candidate for an *assertion* is a **switch statement** with no **default case**.

```
switch(suit) {  
    case Suit.CLUBS:  
        ...  
        break;  
    case Suit.DIAMONDS:  
        ...  
        break;  
    case Suit.HEARTS:  
        ...  
        break;  
    case Suit.SPADES:  
        ...  
}
```

The absence of a default case typically indicates that a programmer believes that one of the cases will always be executed

```
default:  
    assert false: suit;
```

Control-Flow Invariants

- Place an assertion at any location you assume will not be reached
- For example, suppose you have a method that looks like this:

```
void foo() {  
    for (...) {  
        If (...)  
            return;  
    }  
    // Execution should never  
    // reach this point!!!  
}
```

```
void foo() {  
    for (...) {  
        If (...)  
            return;  
    }  
    assert false;  
}
```

Use this technique with discretion

If a statement is unreachable as defined in the Java Language Specification, you will get a compile time error, if you try to assert that it is not reached.

Preconditions

Assert what must be true when a method is invoked. **BUT...**

- **Do not use** assertions to check the **parameters of a public method...**
 - ◆ Preconditions on public methods are enforced by explicit checks that throw particular, specified exceptions
 - ◆ It must check its arguments whether or not assertions are enabled
 - ◆ The assert construct does not throw an exception of the specified type. It can throw only an AssertionError

- **Use** an assertion to test a **non-public method's precondition** that you believe it will be true no matter what a client does with the class.

Postconditions

You **can** test postcondition with assertions in both **public** and **non-public** methods

```
/** Returns a BigInteger whose value is (this-1 mod m).
 *
 * @param m the modulus.
 * @return this-1 mod m.
 * @throws ArithmeticException m <= 0, or this BigInteger
 * has no multiplicative inverse mod m (that is, this BigInteger
 * is not relatively prime to m).
 */
public BigInteger modInverse (BigInteger m) {
    if (m.signum <= 0)
        throw new ArithmeticException("Modulus not positive: " + m);
    ... // Do the computation
    assert ( ((this-1)*result) % m ) == 0 : this;
    return result;
}
```


Class Invariants

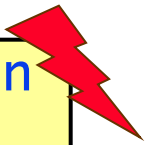
- A class invariant is a type of **internal invariant** that applies to every instance of a class at all times, except when an instance is in transition from one consistent state to another.
- That is, a class invariant should be true **before and after any method completes**
- ex. implement a balanced tree data structure
check that *insertObject()* method, after the end of an insertion and before the *return;* statement, keeps the tree in fact balanced.

```
// Returns true if this tree is properly balanced
private boolean balanced() {
    ...
}
assert balanced();
```

Where not to use assertions

- **DO NOT** use assertions for:
 - ◆ **Argument checking in public methods**
 - Correct arguments is part of the published specifications(or **contract**) of a method and must be obeyed whether assertions are enabled or disabled
 - Erroneous arguments should result in an appropriate **runtime exception**
 - ◆ **Doing any work that your application requires for correct operation**
 - Assertions may be disabled, thus expression contained in an assertion may not be evaluated

// Broken! - Action is contained in assertion
assert names.remove(null);



// Fixed - Action precedes assertion
boolean nullsRemoved = names.remove(null);
assert nullsRemoved;
//Runs whether or not asserts are enabled



Compiling Files That Use Assertions

- For the javac compiler to accept code containing assertions, you must use the “**-source 1.4**” command-line option:

```
%> javac -source 1.4 MyClass.java
```

Enabling and Disabling Assertions

- By **default**, assertions are disabled at runtime
- **Enable** assertions at runtime:
 - ◆ *-enableassertions* or *-ea*
- **Disable** assertions at runtime:
 - ◆ *-disableassertions* or *-da*

Granularity

- You can specify the granularity:
 - ◆ ***no arguments***
 - enables or disables assertions in all classes except system classes
 - ◆ ***packageName***
 - enables or disables assertions in the named package and any subpackages
 - ◆ ***className***
 - enables or disables assertions in the named class
 - ◆ ***...***
 - enables or disables assertions in the unnamed package in the current working directory

```
java -ea:com.wombat.fruitbat...  
-da:com.wombat.fruitbat.Brickbat BatTutor
```

The ADT Stack

- A **stack** is a LIFO queue of objects
- The elements of the stack have consecutive **indices** starting from 0 in the topmost element
- Assuming application requirements. It must be possible:
 - ① To make an empty stack
 - ② To add an element to the top of a stack
 - ③ To remove the topmost element from a stack
 - ④ To test whether a stack is empty
 - ⑤ To access the element at the top of the stack without removing it
 - ⑥ To find the position of an object in the stack

ADT Stack Operations

- The operations defined in a possible contract of ADT Stack are:

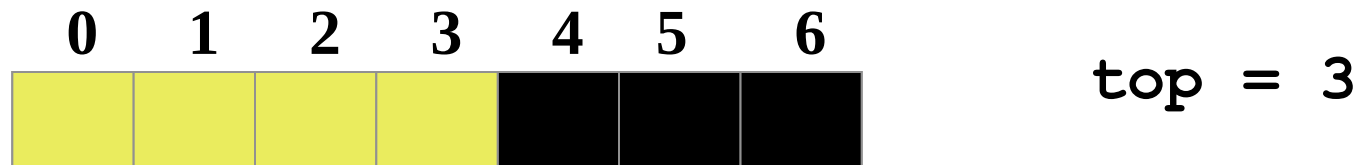
①	stack():	-> Stack
②	isEmpty():	Stack -> Boolean
③	peek():	Stack -> Object
④	search(elem):	Stack x Object -> Int
⑤	clear():	Stack -> Stack (void)
⑥	push(elem):	Stack x Object -> Stack (void)
⑦	pop():	Stack -> Stack (void)

Stack Interface

```
/**
 * Every Class that implements a Stack ADT should
 * implement this interface.
 *
 * @version 1.0
 */
public interface Stack {
    // Method headers and comments go here
    void push (Object item) throws IllegalArgumentException;
    Object pop () throws IllegalStateException;
    Object peek () throws IllegalStateException;
    boolean isEmpty ();
    void clear ();
    int search (Object item);
}
```


ArrayStack

- We implement the stack itself as an Array of Objects
- The top of the stack is an integer index to the last position of the array



- Make the decision that the size of the array can change dynamically
(Assumption: All stacks start with size 100)

ADT Stack Code

```
/**
 * A stack implementation using Array.
 * The capacity of the Array is set to default but it can
 * change dynamically if needed.
 *
 * @version 1.0
 * @author Nikos Armenatzoglou
 */
public class ArrayStack implements Stack {
    // The default capacity of the Array
    public final static int DEFAULT_CAPACITY = 100;

    private Object stack[];           // The stack itself
    private int top;                  // index of the top of the
                                    // stack.
```

ADT Stack Code

```
/**
 * Create a stack with Default Capacity
 */
public ArrayStack() {
    stack = new Object[DEFAULT_CAPACITY];
    top = -1; //No items:top can't point to anything useful
    assert stack != null;           //postcondition
}

/**
 * Tests if this stack is empty.
 *
 * @returns true if and only if this stack contains no
 * items; false otherwise.
 */
public boolean isEmpty() { return top == -1; }
```

ADT Stack Code

```
/**
 * Looks at the object at the top of this stack without
 * removing it from the stack.
 *
 * @return the object at the top of this stack
 * @throws IllegalStateException If this stack is empty.
 */

public Object peek() throws IllegalStateException {
    if (isEmpty())
        throw new IllegalStateException ("Stack is empty");
    assert top != -1: top;
    assert stack[top] != null;           // Invariant
    return stack[top]; // return a reference to the item
                        // on top of the stack
}
```

ADT Stack Code

```
/**
 * Adds an item to the top of the stack.
 *
 * @param item the Object to be pushed onto the Stack
 * @return void
 * @throws IllegalArgumentException If item is null
 */
public void push(Object item) throws IllegalArgumentException{
    if (item == null)
        throw new IllegalArgumentException("item cant be null");
    if (isFull())
        resizeStack();           // resizeStack would be a private
                                // arrayStack method
    top++;                       // Increment top index
    stack[top] = item;           // Add item to new top position
    assert stack[top] == item;   // postcondition
}
```

ADT Stack Code

```
/**
 * Removes the object at the top of this stack and
 * returns that object as the value of this function.
 *
 * @return The object at the top of this stack
 * @throws IllegalStateException - If this stack is
 * empty.
 */
public Object pop() throws IllegalStateException {
    if (isEmpty()) {
        throw new IllegalStateException("Stack is empty");
    }
    Object topItem = stack[top];
    stack[top] = null; //Remove ref to item on stack
    top--;             // increment top
    return topItem;    // return the item that was popped
}
```

ADT Stack Code

```
/**
 * Searches for the specified object in the stack and
 * returns it's position.
 *
 * @param obj The object to find in the stack
 * @return The object's position in this stack, -1 if
 * object wasn't found
 * @throws IllegalStateException If this stack is empty.
 */
public int search(Object obj) throws IllegalStateException {
    if (isEmpty())
        throw new IllegalStateException ("Stack is empty");
    for (int i = 0; i < top; i++) {
        if (stack[i] == obj)
            return i;
    }
    assert i == top : i;
    return -1;
}
```

ADT Stack Code

```
/**
 * clears this stack from all elements.
 */
public void clear() {
    while (top != -1) {
        stack[top--] = null;
    }
    assert top == -1 : top;           // postcondition
    return;
}

} // end of class ArrayStack
```


JUnit

A tool for test-driven development

History

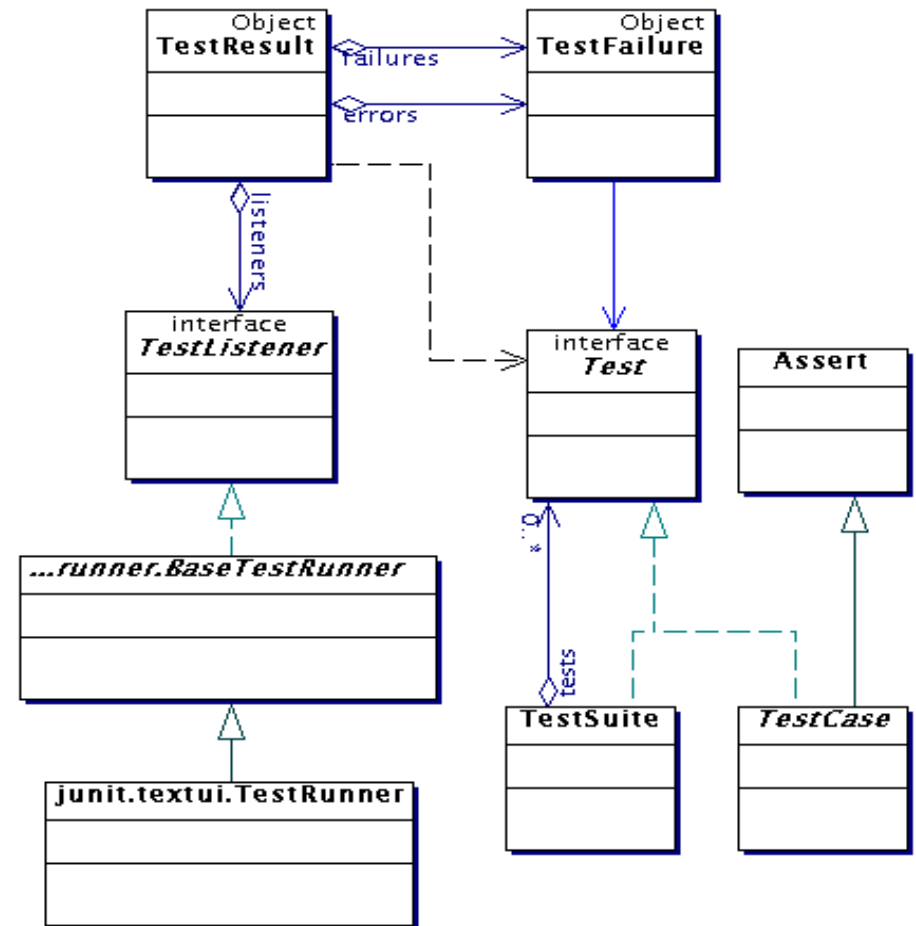
- Kent Beck developed the first xUnit automated test tool for Smalltalk in mid-90's
- Beck and Gamma (of design patterns Gang of Four) developed JUnit on a flight from Zurich to Washington, D.C.
- Martin Fowler: "Never in the field of software development was so much owed by so many to so few lines of code."
- Junit has become the standard tool for Test-Driven Development in Java (see JUnit.org)
- Junit test generators now part of many Java IDEs (Eclipse, BlueJ, Jbuilder, DrJava)
- Xunit tools have since been developed for many other languages (Perl, C++, Python, Visual Basic, C#, ...)

Why Create a Test Suite?

- Obviously you have to test your code—right?
 - ◆ You can do *ad hoc* testing (running whatever tests occur to you at the moment), or
 - ◆ You can build a test suite (a thorough set of tests that can be run at any time)
- Disadvantages of a test suite
 - ◆ It's a lot of extra programming
 - True, but use of a good test framework can help quite a bit
 - ◆ You don't have time to do all that extra work
 - *False!* Experiments repeatedly show that test suites reduce debugging time more than the amount spent building the test suite
- Advantages of a test suite
 - ◆ Reduces total number of bugs in delivered code
 - ◆ Makes code much more maintainable and refactorable

Architectural Overview

- JUnit test framework is a package of classes that lets you write tests for each method, then easily run those tests
- TestRunner** runs tests and reports **TestResults**
- You test your class by extend abstract class **TestCase**
- To write test cases, you need know and understand the **Assert** class



Writing a TestCase

- To start using JUnit, create a subclass of *TestCase*, to which you add test methods
- Here's a skeletal test class:

```
import junit.framework.TestCase;
public class TestBow1 extends TestCase {
} //Test my class Bow1
```

- Name of class is important – should be of the form ***Test***MyClass or MyClass***Test***
- This naming convention lets TestRunner automatically find your test classes

Writing methods in TestCase

- Pattern follows *programming by contract* paradigm:

- ◆ Set up **preconditions**
- ◆ Exercise functionality being tested
- ◆ Check **postconditions**

- Example:

```
public void testEmptyList() {  
    Bowl emptyBowl = new Bowl();  
    assertEquals("Size of an empty list should be zero.",  
        0, emptyList.size());  
    assertTrue("An empty bowl should report empty.",  
        emptyBowl.isEmpty());  
}
```

- Things to notice:

- ◆ Specific method signature – public void *test*whatever()
 - Allows them to be found and collected automatically by JUnit
- ◆ Coding follows pattern
- ◆ Notice the assert-type calls...

Assert methods

- Each assert method has parameters like these:
message, expected-value, actual-value
- Assert methods dealing with floating point numbers get an additional argument, a tolerance
- Each assert method has an equivalent version that does not take a message – however, this use is not recommended because:
 - ◆ messages helps documents the tests
 - ◆ messages provide additional information when reading failure logs

Assert methods

- `assertTrue(String message, Boolean test)`
- `assertFalse(String message, Boolean test)`
- `assertNull(String message, Object object)`
- `assertNotNull(String message, Object object)`
- `assertEquals(String message, Object expected, Object actual)`
 - ◆ (uses `equals` method)
- `assertSame(String message, Object expected, Object actual)`
 - ◆ (uses `==` operator)
- `assertNotSame(String message, Object expected, Object actual)`

More stuff in test classes

- Suppose you want to test a class `Counter`
- `public class CounterTest`
`extends junit.framework.TestCase {`
 - ◆ This is the unit test for the `Counter` class
- `public CounterTest() { } //Default constructor`
- `protected void setUp()`
 - ◆ Test *fixture* creates and initializes instance variables, etc.
- `protected void tearDown()`
 - ◆ Releases any system resources used by the test fixture
- `public void testIncrement(), public void testDecrement()`
 - ◆ These methods contain tests for the `Counter` methods `increment()`, `decrement()`, etc.
 - ◆ Note capitalization convention

JUnit tests for Counter

```
public class CounterTest extends junit.framework.TestCase {
    Counter counter1;
    public CounterTest() { } // default constructor
    protected void setUp() { // creates a (simple) test fixture
        counter1 = new Counter();
    }
    public void testIncrement() {
        assertTrue(counter1.increment()==1);
        assertTrue(counter1.increment()==2);
    }

    public void testDecrement() {
        assertTrue(counter1.decrement()==-1);
    }
}
```

Note that each test begins with a *brand new* counter

This means you don't have to worry about the order in which the tests are run

TestSuites

- TestSuites collect a selection of tests to run them as a unit
- Collections automatically use TestSuites, however to specify the order in which tests are run, write your own:

```
public static Test suite() {  
    suite.addTest(new TestBow1("testBow1"));  
    suite.addTest(new TestBow1("testAdding"));  
    return suite;  
}
```

- Should seldom have to write your own TestSuites as each method in your TestCase should be independent of all others
- Can create TestSuites that test a whole package:

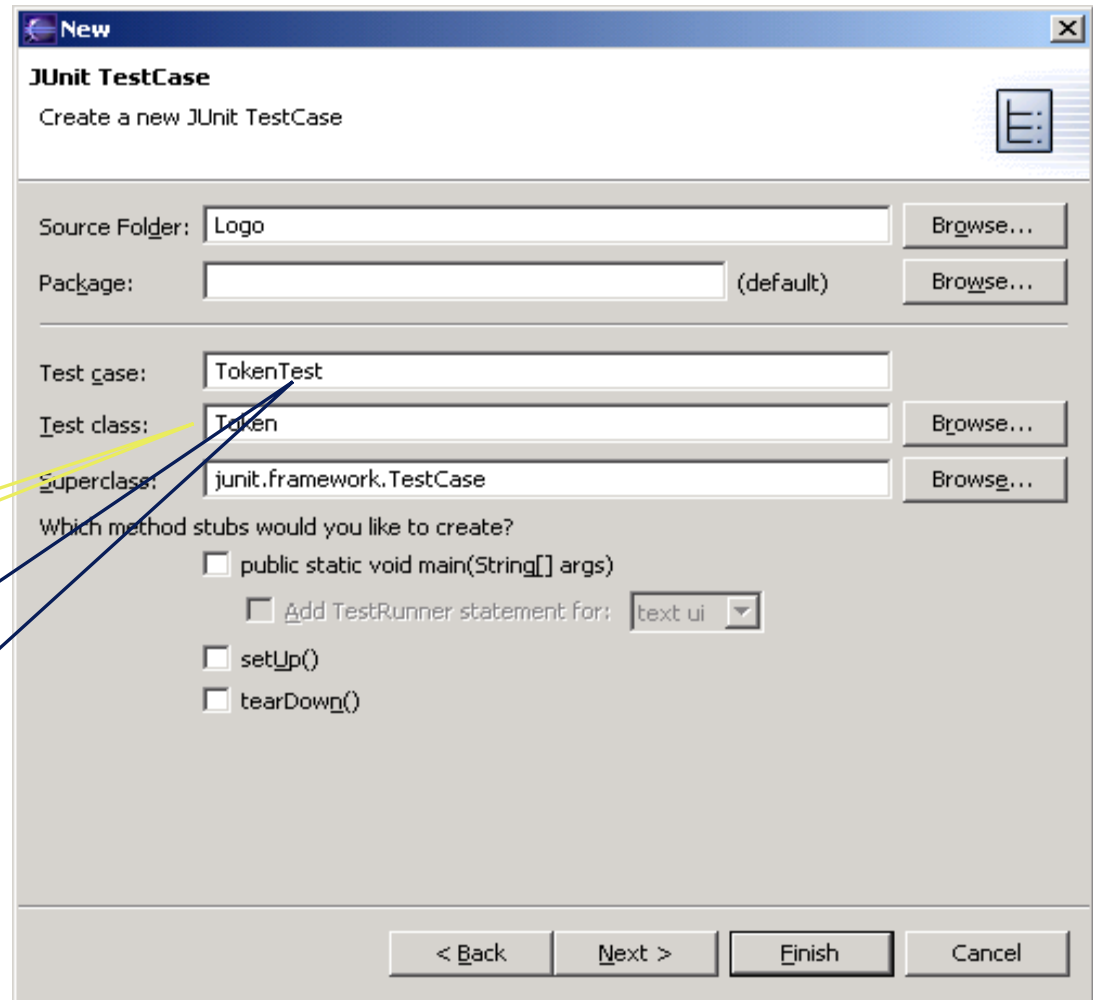
```
public static Test suite() {  
    TestSuite suite = new TestSuite();  
    suite.addTestSuite(TestBow1.class);  
    suite.addTestSuite(TestFruit.class);  
    return suite;  
}
```

JUnit in Eclipse

- To create a test class, select File→ New→ Other... → Java, JUnit, TestCase and enter the name of the *class* you will test

Fill this in

**This will be filled in
*automatically***



New
JUnit TestCase
Create a new JUnit TestCase

Source Folder: Logo Browse...

Package: (default) Browse...

Test case: TokenTest

Test class: Token Browse...

Superclass: junit.framework.TestCase Browse...

Which method stubs would you like to create?

☐ public static void main(String[] args)

☐ Add TestRunner statement for: text ui

☐ setUp()

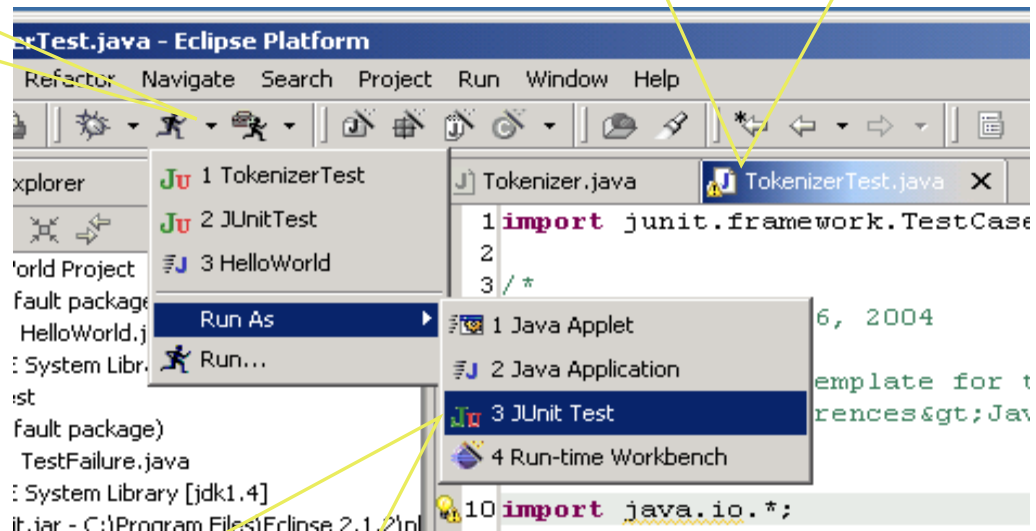
☐ tearDown()

< Back Next > Finish Cancel

Running JUnit

Second, use this pulldown menu

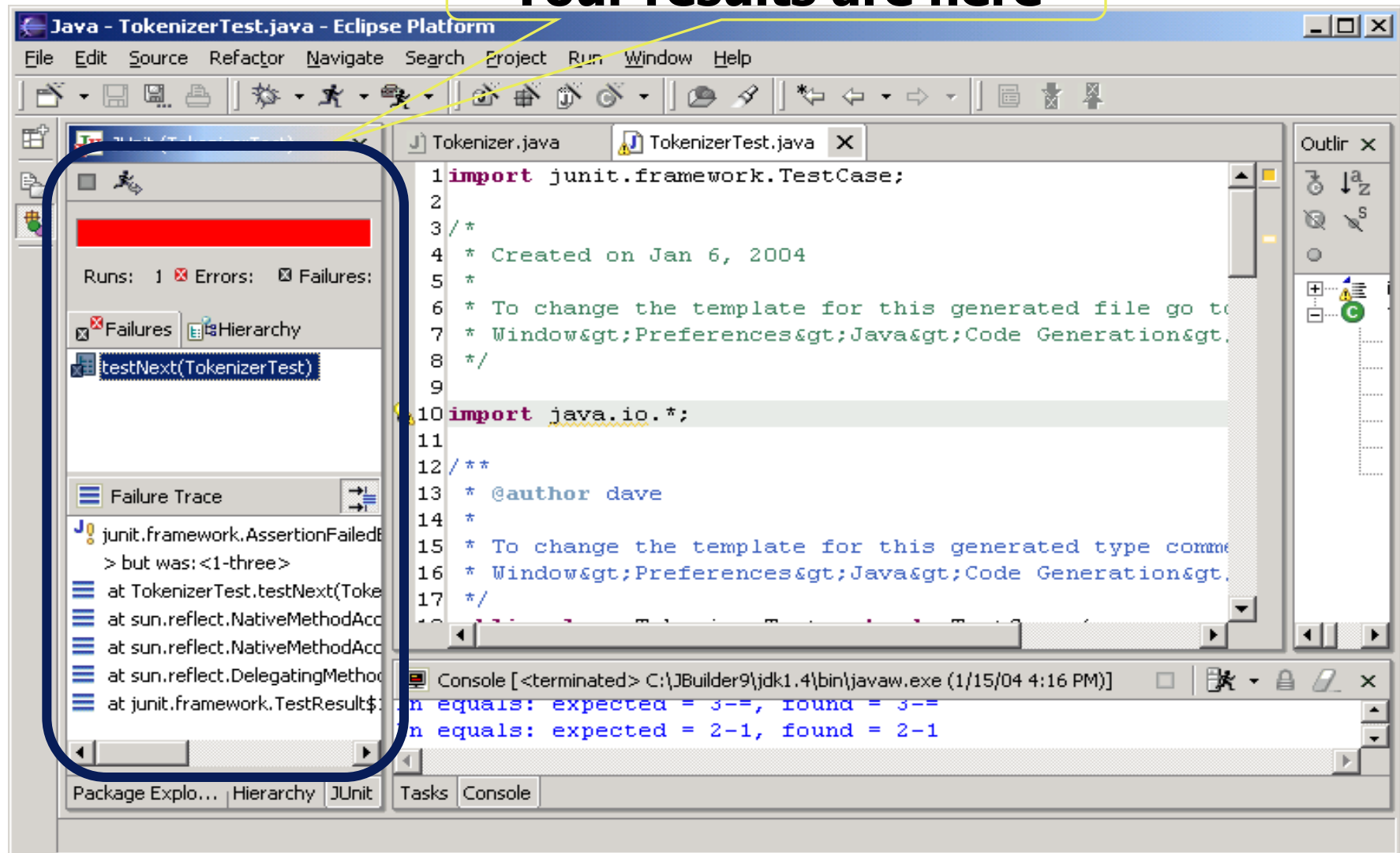
First, select a *Test* class



Third, Run As → JUnit Test

Results

Your results are here



The screenshot shows the Eclipse IDE interface. The main editor displays the `TokenizerTest.java` file, which is a JUnit test class. The test class includes a `testNext` method. The left sidebar shows the `JUnit` view, which indicates that the test `testNext(TokenizerTest)` has failed. The failure trace shows the following details:

- Failure Trace:
 - junit.framework.AssertionFailedError: expected = 3--, round = 3--
 - at TokenizerTest.testNext(TokenizerTest.java:10)
 - at sun.reflect.NativeMethodAccessorImpl.invoke0(NativeMethodAccessorImpl.java:62)
 - at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
 - at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
 - at junit.framework.TestRunner\$1.run(TestRunner.java:39)

The bottom console window shows the output of the test run, indicating that the test failed due to an assertion error. The console output is:

```
assertEquals: expected = 3--, round = 3--
assertEquals: expected = 2-1, found = 2-1
```

More Information

- <http://www.junit.org>
 - ◆ Download of JUnit
 - ◆ Lots of information on using JUnit
- <http://www.thecoadletter.com>
 - ◆ Information on Test-Driven Development