ΕΡΓΑΣΤΗΡΙΟ 6
JUnit

Using JUnit in Eclipse

- Creating a JUnit Test Case in Eclipse
- Writing Tests
- Running Your Test Case

Τα τεστ λογισμικού βοηθούν τους προγραμματιστές να βεβαιωθούν ότι η λογική του κώδικα είναι σωστή. Υπάρχουν πολλά εργαλεία που μας επιτρέπουν να κάνουμε κάτι τέτοιο, με πιο δημοφιλείς τα JUnit και TestNG. Για το μάθημα ΕΠΛ133 θα χρησιμοποιήσουμε το Junit.

Το JUnit είναι μια βιβλιοθήκη της Java που χρησιμοποιείται για τη δημιουργία ελέγχων unit testing. Το unit testing είναι η διαδικασία έλεγχου μιας μικρής «μονάδας» λογισμικού (συνήθως μιας κλάσης) για να βεβαιωθούμε ότι ανταποκρίνεται στις προσδοκίες ή στις προδιαγραφές του.

Ένα unit test είναι μια κλάση που ελέγχει μια άλλη κλάση (η κλάση υπό δοκιμή). Για παράδειγμα, η κλάση ArrayIntListTest ελέγχει την κλάση ArrayIntList που είναι η κλάση υπό δοκιμή. Μια κλάση unit test αποτελείται από διάφορες μεθόδους δοκιμών που η κάθε μια αλληλεπιδρά με την κλάση υπό δοκιμή με κάποιο συγκεκριμένο τρόπο, έτσι ώστε να βεβαιωθούμε ότι λειτουργεί όπως αναμένεται.

Το JUnit δεν αποτελεί μέρος της βιβλιοθήκης έτοιμων κλάσεων της Java, αλλά περιλαμβάνεται στο Eclipse. Αν δεν χρησιμοποιείτε Eclipse, το JUnit μπορείτε να το κατεβάσετε δωρεάν από την ιστοσελίδα http://junit.org. Το JUnit διανέμεται ως "JAR", το οποίο είναι ένα συμπιεσμένο αρχείο που περιέχει αρχεία .class.

Writing Tests

Κάθε μέθοδος δοκιμής (unit test method) στο αρχείο δοκιμής JUnit, πρέπει να τεστάρει μια συγκεκριμένη μικρή πτυχή της συμπεριφοράς της "κλάσης υπό τεστ." Για παράδειγμα, μια κλάση ArrayIntListTest θα μπορούσε να έχει μία μέθοδο δοκιμής για να δούμε εάν τα στοιχεία μπορούν να εισάγονται στη λίστα και στη συνέχεια να ανακτώνται. Ένα άλλο τεστ μπορεί να ελέγξει για να βεβαιωθείτε ότι το μέγεθος της λίστας είναι σωστό μετά από διάφορους χειρισμούς. Και ούτω καθεξής. Κάθε μέθοδος δοκιμών θα πρέπει να είναι σύντομη και θα πρέπει να δοκιμάσει μόνο μια συγκεκριμένη πτυχή της κλάσης υπό δοκιμή.

Οι μέθοδοι δοκιμών JUnit χρησιμοποιούν *assertions*, οι οποίες είναι δηλώσεις που ελέγχουν κατά πόσον μια δεδομένη συνθήκη είναι αληθείς ή ψευδείς. Εάν η συνθήκη είναι ψευδής, η μέθοδος δοκιμής αποτυγχάνει. Εάν όλα τα *assertions' conditions* στη μέθοδο δοκιμής είναι αληθές, τότε η μέθοδος δοκιμής επιτυγχάνει. Μπορείτε να χρησιμοποιήσετε *assertions* για καταστάσεις που περιμένετε πάντα να είναι αληθείς, όπως `assertEquals(3, list.size());` εάν περιμένετε ότι η λίστα περιέχει ακριβώς 3 στοιχεία σε εκείνο το σημείο στον κώδικα. Το JUnit παρέχει τις ακόλουθες μεθόδους *assertion*:

method name / parameters	description
`assertTrue(`*`test`*`)` `assertTrue("`*`message`*`", `*`test`*`)`	Causes this test method to fail if the given `boolean` test is not `true`.
`assertFalse(`*`test`*`)` `assertFalse("`*`message`*`", `*`test`*`)`	Causes this test method to fail if the given `boolean` test is not `false`.
`assertEquals(`*`expectedValue`*`, `*`value`*`)` `assertEquals("`*`message`*`", `*`expectedValue`*`, `*`value`*`)`	Causes this test method to fail if the given two values are not equal to each other. (For objects, it uses the `equals` method to compare them.) The first of the two values is considered to be the result that you expect; the second is the actual result produced by the class under test.
`assertNotEquals(`*`value1`*`, `*`value2`*`)` `assertNotEquals("`*`message`*`", `*`value1`*`, `*`value2`*`)`	Causes this test method to fail if the given two values *are* equal to each other. (For objects, it uses the `equals` method to compare them.)
`assertNull(`*`value`*`)` `assertNull("`*`message`*`", `*`value`*`)`	Causes this test method to fail if the given value is not `null`.
`assertNotNull(`*`value`*`)` `assertNotNull("`*`message`*`", `*`value`*`)`	Causes this test method to fail if the given value *is* `null`.

`assertSame(`*`expectedValue`*`, `*`value`*`)`
`assertSame("`*`message`*`", `*`expectedValue`*`, `*`value`*`)`
`assertNotSame(`*`value1`*`, `*`value2`*`)`
`assertNotSame("`*`message`*`", `*`value1`*`, `*`value2`*`)`
	Identical to `assertEquals` and `assertNotEquals` respectively, except that for objects, it uses the == operator rather than the `equals` method to compare them. (The difference is that two objects that have the same state might be `equals` to each other, but not == to each other. An object is only == to itself.)

`fail()` `fail("`*`message`*`")`	Causes this test method to fail.

`void assertArrayEquals([String message], expectedArray, resultArray)`	Asserts that the array expected and the resulted array are equal. The type of Array might be int, long, short, char, byte or java.lang.Object.

Πιο κάτω δείτε ένα γρήγορο παράδειγμα που χρησιμοποιεί πολλές από αυτές τις μεθόδους assertion.

```
ArrayIntList list = new ArrayIntList();
list.add(42);
list.add(-3);
list.add(17);
list.add(99);

assertEquals(4, list.size());
assertEquals(17, list.get(2));
assertTrue(list.contains(-3));
assertFalse(list.isEmpty());
```

Σημειώστε ότι κατά τη χρήση συγκρίσεων όπως `assertEquals`, οι αναμενόμενες τιμές γράφονται ως η αριστερή (πρώτη) παράμετρο, και οι πραγματικές κλήσεις στη λίστα πρέπει να είναι γραμμένες στα δεξιά (δεύτερη παράμετρο). Αυτό γίνεται έτσι ώστε αν μια δοκιμή αποτύχει, το JUnit να δώσει το σωστό μήνυμα λάθους όπως "`expected 4 but found 0`".

Μια καλογραμμένη μέθοδο δοκιμής επιλέγει τη μέθοδο `assert` που είναι η πλέον κατάλληλη για το συγκεκριμένο έλεγχο. Χρησιμοποιώντας την πιο κατάλληλη μέθοδο `assert` βοηθά το JUnit να παρέχει καλύτερα μηνύματα λάθους όταν μια δοκιμή αποτύχει. Δείτε το παράδειγμα που ακλουθεί:

```
// This code uses bad style.
assertTrue(list.size() == 4);        // bad; use assertEquals
assertTrue(list.get(2) == 17);       // bad; use assertEquals
if (!list.contains(-3)) {
    fail();                          // bad; use assertTrue
}
assertTrue(!list.isEmpty());         // bad; use assertFalse and delete the !
```

Καλές μέθοδοι δοκιμής θεωρούνται αυτές που είναι σύντομες και δοκιμάζουν μόνο μια συγκεκριμένη πτυχή της κλάσης υπό δοκιμή. Το παραπάνω παράδειγμα κώδικα είναι με αυτή την έννοια ένα κακό παράδειγμα: δεν πρέπει να τεστάρουμε το `size`, `get`, `contains`, and `isEmpty`, όλα σε μία μέθοδο. Μια καλύτερη (ελλιπή) σειρά δοκιμών θα μπορούσε να είναι σαν το παρακάτω:

```
@Test
public void testAddAndGet1() {
    ArrayIntList list = new ArrayIntList();
    list.add(42);
    list.add(-3);
    list.add(17);
    list.add(99);
    assertEquals(42, list.get(0));
```

```
    assertEquals(-3, list.get(1));
    assertEquals(17, list.get(2));
    assertEquals(99, list.get(3));

    assertEquals("second attempt", 42, list.get(0));   // make sure I can get
them a second time
    assertEquals("second attempt", 99, list.get(3));
}

@Test
public void testSize1() {
    ArrayIntList list = new ArrayIntList();
    assertEquals(0, list.size());
    list.add(42);
    assertEquals(1, list.size());
    list.add(-3);
    assertEquals(2, list.size());
    list.add(17);
    assertEquals(3, list.size());
    list.add(99);
    assertEquals(4, list.size());
    assertEquals("second attempt", 4, list.size());   // make sure I can get
it a second time
}

@Test
public void testIsEmpty1() {
    ArrayIntList list = new ArrayIntList();
    assertTrue(list.isEmpty());
    list.add(42);
    assertFalse("should have one element", list.isEmpty());
    list.add(-3);
    assertFalse("should have two elements", list.isEmpty());
}

@Test
public void testIsEmpty2() {
    ArrayIntList list = new ArrayIntList();
    list.add(42);
    list.add(-3);
    assertFalse("should have two elements", list.isEmpty());
    list.remove(1);
    list.remove(0);
    assertTrue("after removing all elements", list.isEmpty());
    list.add(42);
    assertFalse("should have one element", list.isEmpty());
}

...
```

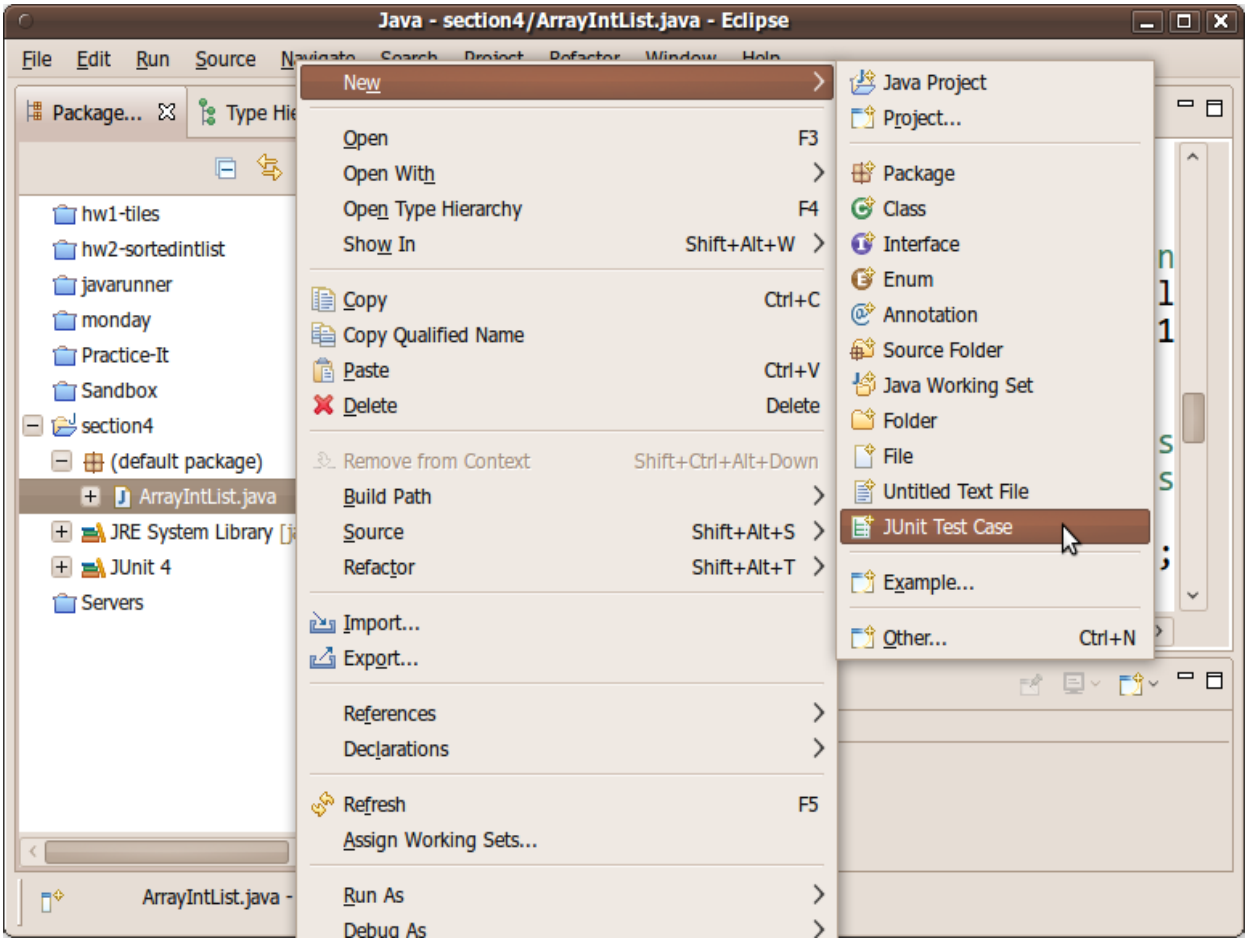
Υπάρχουν πολύ περισσότερα που θα μπορούσε να πει κανείς για το γράψιμο αποτελεσματικών μονάδων τεστ, αλλά αυτό είναι έξω από το πεδίο αυτού του εργαστηρίου. Εάν είστε περίεργοι, μπορείτε να μάθετε περισσότερα διαβάζοντας σελίδες όπως αυτό ή αυτό ή αυτό.

Ίσως να σκεφτείτε ότι το γράψιμο μονάδων δοκιμών δεν είναι χρήσιμο και ότι μπορείτε να εξετάσετε μόνο τον κώδικα των μεθόδων, όπως η μέθοδος add() ή η isEmpty() και να δείτε αν δουλεύουν. Αλλά είναι εύκολο να υπάρχουν σφάλματα (πάντα υπάρχουν), και το JUnit θα τα βρει καλύτερα από τα δικά σας μάτια.

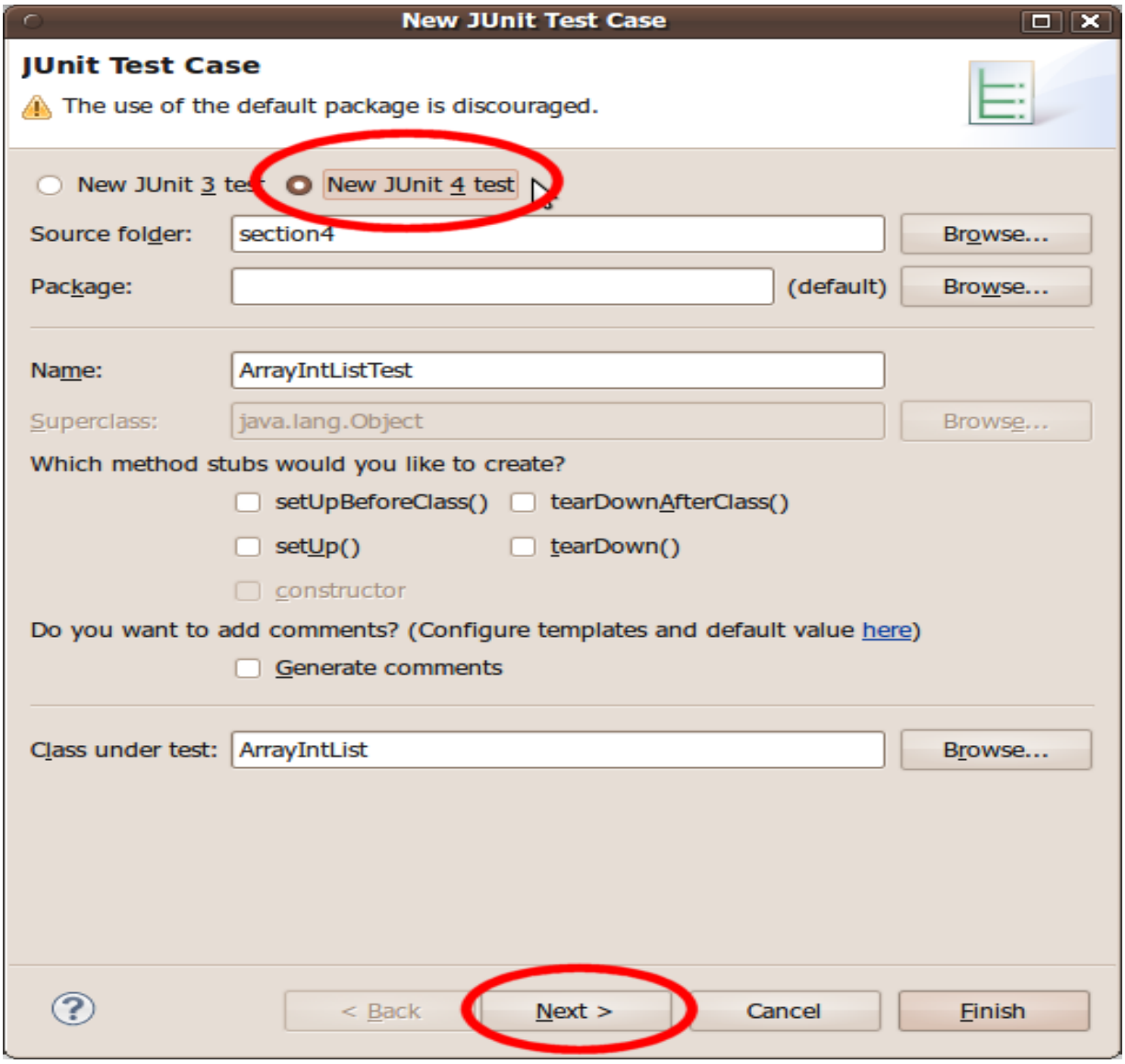
Ακόμα και αν γνωρίζουμε ήδη ότι ο κώδικας λειτουργεί, τα unit test μπορούν ακόμα να αποδεδειχθούν χρήσιμα. Μερικές φορές εισάγουμε σφάλματα κατά την προσθήκη νέων χαρακτηριστικών ή αλλαγών στο υφιστάμενο κώδικα, οπότε κάτι που λειτουργούσε σωστά, τώρα έχει πρόβλημα. Αυτό ονομάζεται παλινδρόμηση (*regression*). Αν όμως έχουμε μονάδες δοκιμών (unit test) JUnit για το παλιό κώδικα, μπορούμε να διασφαλίσουμε ότι εξακολουθούν να λειτουργούν σωστά (να περνούν τα τεστ) και να αποφευχθούν οι δαπανηρές παλινδρομήσεις.

Creating a JUnit Test Case in Eclipse

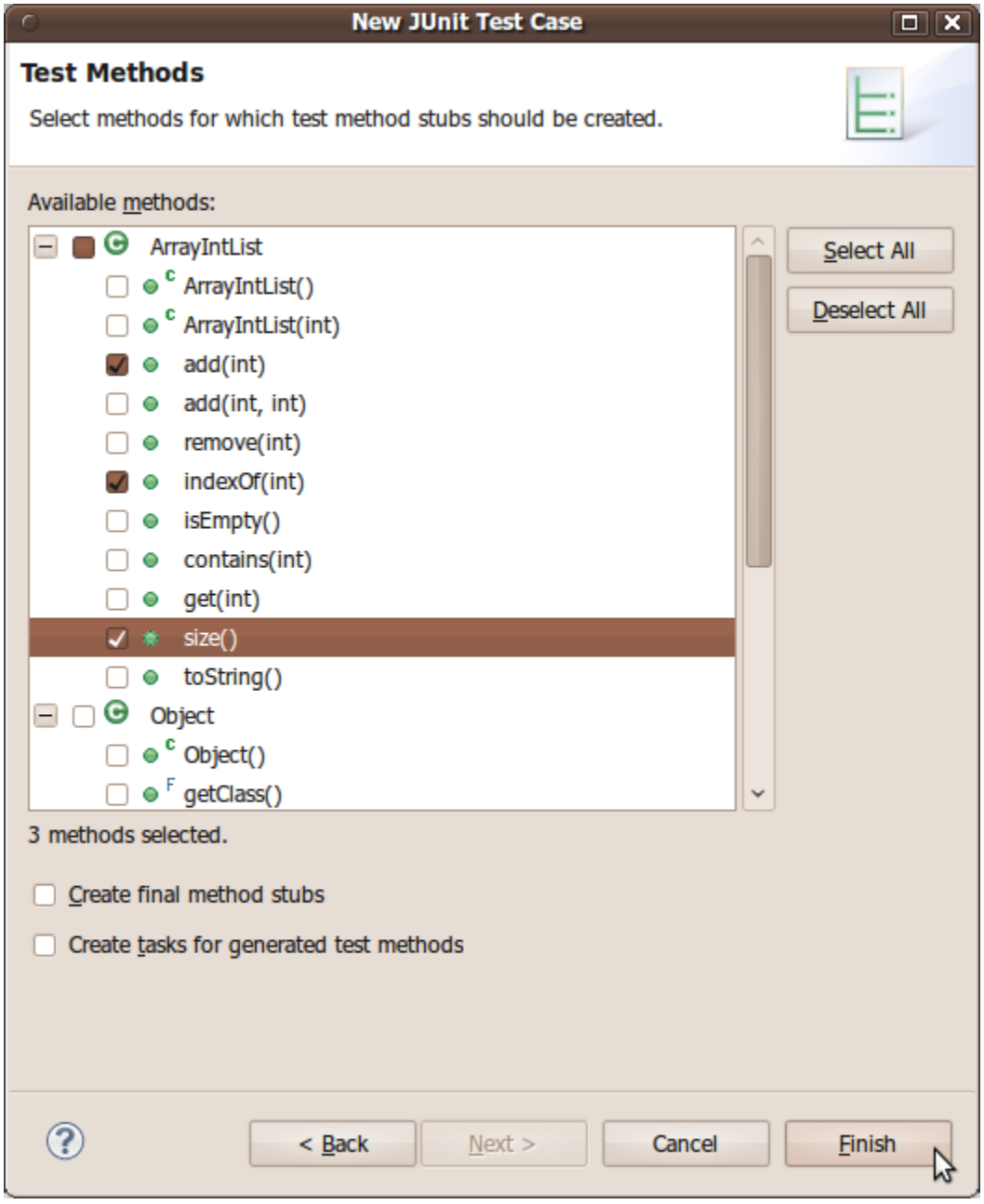
Για να χρησιμοποιήσετε το JUnit πρέπει να δημιουργήσετε ένα ξεχωριστό αρχείο .java στο δικό σας project. Σε αυτό το αρχείο θα γίνουν τα τεστ μιας κλάσης από το σύνολο των κλάσεων του προγράμματος σας. Στο Package Explorer αριστερά, κάντε δεξί κλικ στην κλάση που θέλετε να τεστάρετε και μετά New → JUnit Test Case.



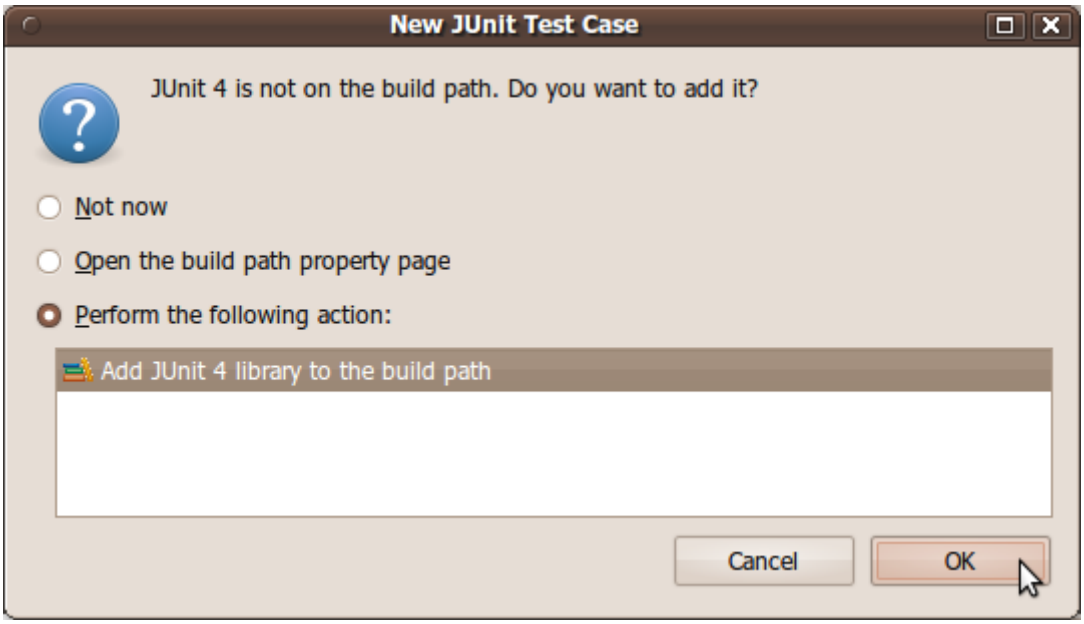
Ένα dialog box θα ανοίξει και θα σας οδηγήσει στη δημιουργία του test case. Βεβαιωθείτε ότι η επιλογή στην κορυφή αυτού του dialog box είναι Junit4 και όχι Junit3. Μετά πατήστε Next.



Θα δείτε μια σειρά από checkboxes που δείχνουν τις μεθόδους που θέλετε να ελέγξετε. Η Eclipse θα σας βοηθήσει με τη δημιουργία «stub» μεθόδων δοκιμής τις οποίες μπορείτε να συμπληρώσετε με κώδικα. (Μπορείτε πάντα να προσθέσετε περισσότερα αργότερα με το χέρι.) Επιλέξτε τις μεθόδους για τη δοκιμή και κάντε κλικ στο Finish.



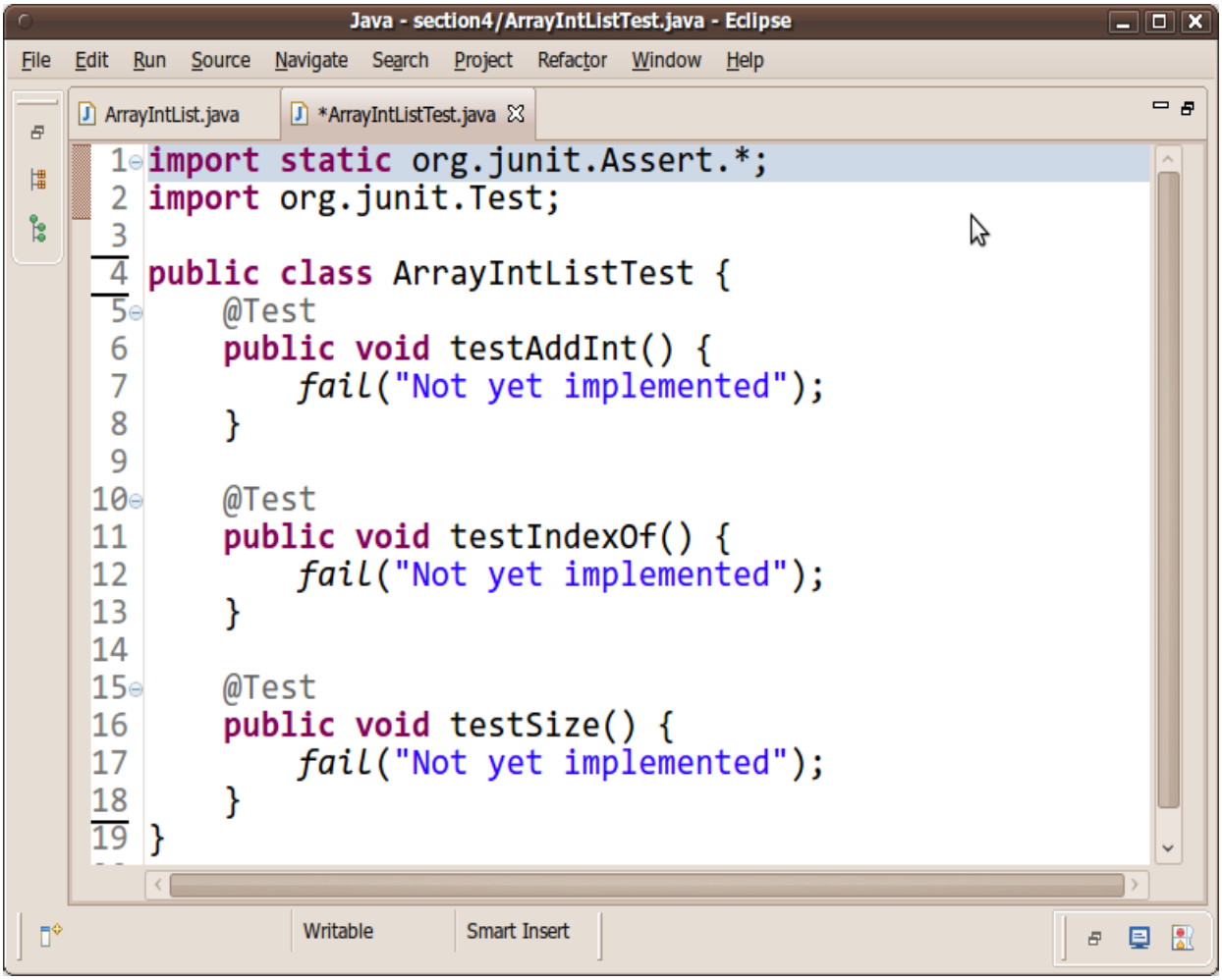
Σε αυτό το σημείο το Eclipse θα σας ρωτήσει αν θέλετε να επισυνάψετε αυτόματα τη βιβλιοθήκη JUnit στο project σας. Ναι, πρέπει να το κάνουμε. Επιλέξτε " Perform the following action: Add JUnit 4 library to the build path " και πατήστε OK.



(Αν ξεχάσατε να προσθέσετε το JUnit στο project σας, μπορείτε να το προσθέσετε αργότερα κάνοντας κλικ στο μενού Project, στη συνέχεια, Properties, στη συνέχεια, Java Build Path, στη συνέχεια, κάντε κλικ στο κουμπί Add Library... και επιλέξτε JUnit 4 από τη λίστα.)

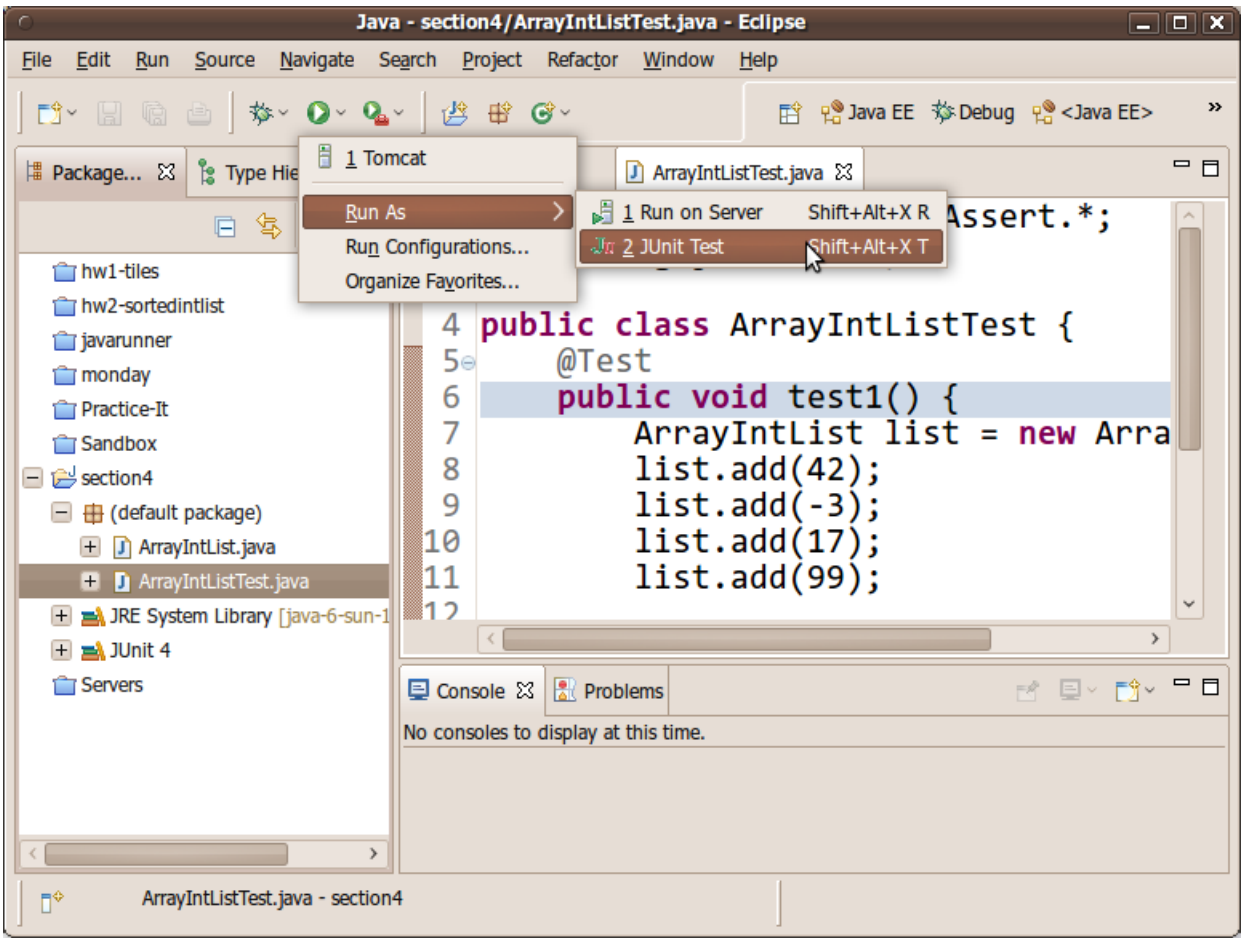
Όταν έχετε τελειώσει με τα παραπάνω, πρέπει να έχετε ένα ωραίο νέο αρχείο δοκιμής JUnit. Προτείνω να αλλάξετε τη δεύτερη δήλωση ενσωμάτωσης στην κορυφή αυτού του αρχείου ως εξής:

```
import org.junit.*;   // instead of  import org.junit.Test;
```

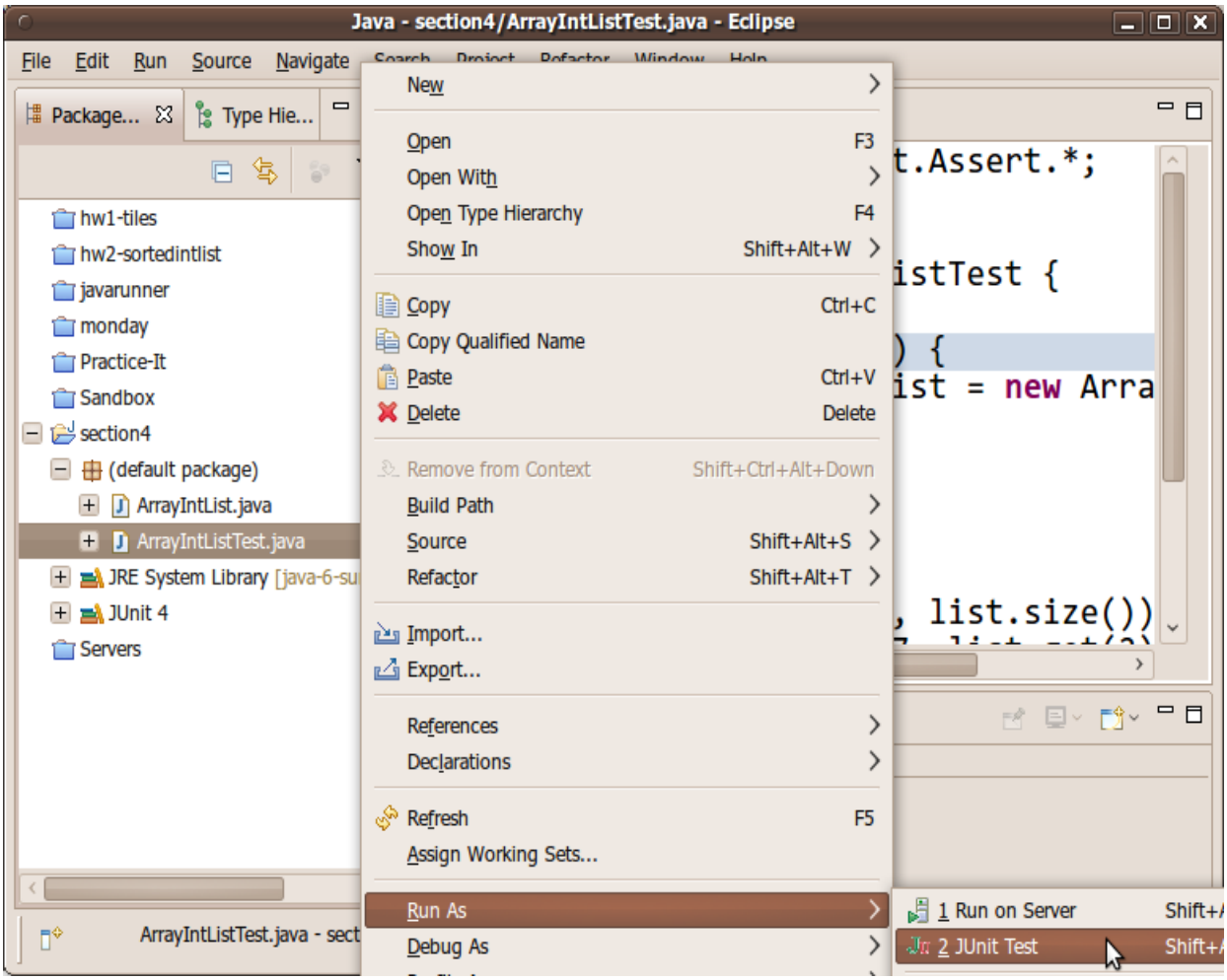


Running Your Test Case

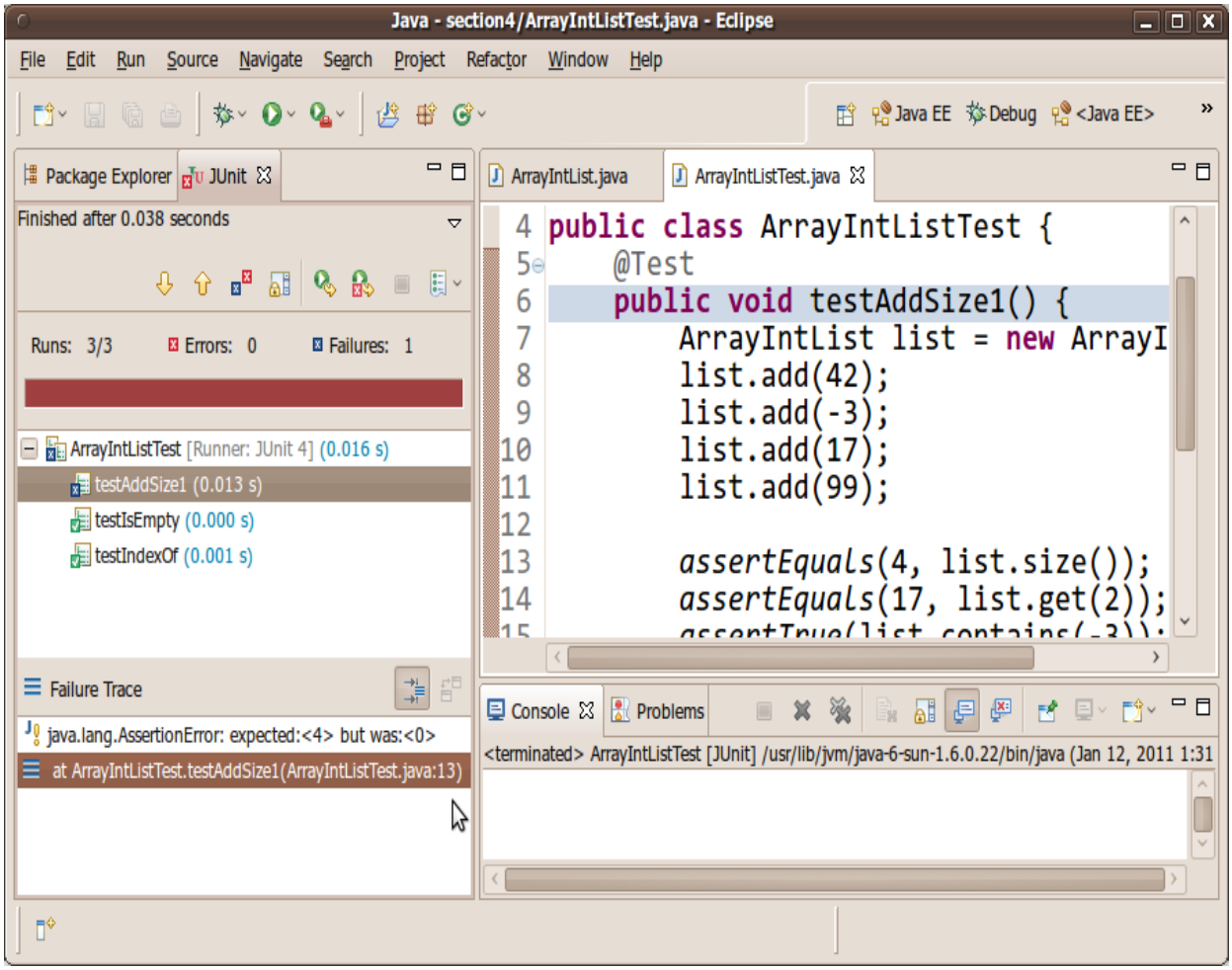
Όταν έχετε γράψει μια ή δύο μεθόδους δοκιμών, τρέξε την κλάση δοκιμή JUnit σας. Υπάρχουν δύο τρόποι για να γίνει αυτό. Ένας τρόπος είναι να κάνετε κλικ στο κουμπί Run στο toolbar (μοιάζει με ένα πράσινο σύμβολο "Play"). Ένα μενού θα εμφανιστεί προς τα κάτω, επιλέξετε να εκτελέσετε την κλάση ως JUnit Test.



Ο άλλος τρόπος είναι να κάνετε δεξί κλικ στη κλάση JUnit test case και επιλέξτε Run As → JUnit Test.



Ένα νέο παράθυρο θα εμφανιστεί δείχνοντας τα αποτελέσματα των δοκιμών για κάθε μέθοδο. Θα δείτε μια πράσινη γραμμή, εάν όλες οι δοκιμές έχουν επιτύχει ή μια κόκκινη γραμμή, αν κάποια από τις δοκιμές απέτυχε. Αν κάποιες δοκιμές αποτύχουν, μπορείτε να δείτε τις λεπτομέρειες σχετικά με την αποτυχία, κάνοντας κλικ στο αποτυχημένη δοκιμή όνομα/εικονίδιο και κοιτάζοντας τις λεπτομέρειες στο παρακάτω παράθυρο.



Οι περισσότεροι πιστεύουν ότι έχοντας πάρει μια κόκκινη γραμμή η αποτυχία είναι κάτι κακό. Δεν είναι! Είναι καλό, και αυτό σημαίνει ότι έχετε βρει ένα πιθανό σφάλμα που πρεπει να διορθωθεί. Η εύρεση και η διόρθωση σφαλμάτων είναι ένα καλό πράγμα. Κάνοντας μια κόκκινη γραμμή να γίνει μια πράσινη γραμμή (με τη διόρθωση του κώδικα και στη συνέχεια εκ νέου την εκτέλεση του προγράμματος δοκιμών) μπορεί να είναι πολύ θετικο.

JUnit annotations

Στην ενότητα αυτή θα αναφέρουμε τα βασικά annotations που υποστηρίζει το JUnit 4. Ο παρακάτω πίνακας παρουσιάζει μια σύνοψη αυτών των annotations:

Πίνακας 1

Annotation	Description
`@Test` `public void method()`	The `Test` annotation indicates that the public void method to which it is attached can be run as a test case.
`@Before` `public void method()`	The `Before` annotation indicates that this method must be executed before each test in the class, so as to execute some preconditions necessary for the test.
`@BeforeClass` `public static void method()`	The `BeforeClass` annotation indicates that the static method to which is attached must be executed once and before all tests in the class. That happens when the test methods share computationally expensive setup (e.g. connect to database).
`@After` `public void method()`	The `After` annotation indicates that this method gets executed after execution of each test (e.g. reset some variables after execution of every test, delete temporary variables etc)
`@AfterClass` `public static void method()`	The `AfterClass` annotation can be used when a method needs to be executed after executing all the tests in a JUnit Test Case class so as to clean-up the expensive set-up (e.g disconnect from a database). Attention: The method attached with this annotation (similar to `BeforeClass`) must be defined as static.
`@Ignore` `public static void method()`	The `Ignore` annotation can be used when you want temporarily disable the execution of a specific test. Every method that is annotated with `@Ignore` won't be executed.

Ας δούμε ένα παράδειγμα μίας κλάσης δοκιμή με μερικά από τα annotations που αναφέρονται στο προηγούμενο πίνακα.

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

public class AnnotationsTest {

	private ArrayList testList;

	@BeforeClass
	public static void onceExecutedBeforeAll() {
		System.out.println("@BeforeClass: onceExecutedBeforeAll");
	}

	@Before
	public void executedBeforeEach() {
		testList = new ArrayList();
		System.out.println("@Before: executedBeforeEach");
	}

	@AfterClass
	public static void onceExecutedAfterAll() {
		System.out.println("@AfterClass: onceExecutedAfterAll");
	}

	@After
	public void executedAfterEach() {
		testList.clear();
		System.out.println("@After: executedAfterEach");
	}

	@Test
	public void EmptyCollection() {
		assertTrue(testList.isEmpty());
		System.out.println("@Test: EmptyArrayList");

	}

	@Test
	public void OneItemCollection() {
		testList.add("oneItem");
		assertEquals(1, testList.size());
		System.out.println("@Test: OneItemArrayList");
	}

	@Ignore
	public void executionIgnored() {

		System.out.println("@Ignore: This execution is ignored");
	}

}
```

Εάν τρέξουμε το παραπάνω τεστ, η έξοδος του προγράμματος θα είναι η ακόλουθη:

```
@BeforeClass: onceExecutedBeforeAll
@Before: executedBeforeEach
@Test: EmptyArrayList
@After: executedAfterEach
@Before: executedBeforeEach
@Test: OneItemArrayList
@After: executedAfterEach
@AfterClass: onceExecutedAfterAll
```

Ασκήσεις.

1. Σας δίνεται ο παρακάτω κώδικας:

```
public class Calculation {

  public static int findMax(int arr[]){
            int max=0;
        for(int i=1;i<arr.length;i++){
                if(max<arr[i])
                    max=arr[i];
      }
      return max;
        }
}
```

   Γράψτε τα Junit test για αυτήν την κλάση. Διορθώστε τον κώδικα σε περίπτωση που υπάρχει λάθος.

2. Προσθέστε στην προηγούμενη κλάση τις ακόλουθες μεθόδους:

```
//method that returns cube of the given number
public static int cube(int n) {
      return n * n * n;
}

// method that returns reverse words
public static String reverseWord(String str) {

      StringBuilder result = new StringBuilder();
      StringTokenizer tokenizer = new StringTokenizer(str, " ");

      while (tokenizer.hasMoreTokens()) {
            StringBuilder sb = new StringBuilder();
            sb.append(tokenizer.nextToken());
            sb.reverse();

            result.append(sb);
            result.append(" ");
      }
      return result.toString();
}
```

   Δημιουργήστε ένα ολοκληρωμένο Junit test που να περιλαμβάνει όλα τα annotations του Πίνακα 1.

3. Γράψτε τα Junit tests για την κλάση ArrayIntList που σας δίνεται.

```
// An ArrayIntList object stores an ordered list of integers using
// an unfilled array.

import java.util.*; // for Arrays

public class ArrayIntList {
	private static final int INITIAL_CAPACITY = 10;

	// fields - the data inside each ArrayIntList object
	private int size;
	private int[] elementData;

	// Initializes a new empty list with initial capacity of 10 integers.
	public ArrayIntList() {
		this(INITIAL_CAPACITY); // call the (int) constructor
	}

	// Initializes a new empty list with the given initial capacity.
	// Precondition: capacity > 0
	public ArrayIntList(int capacity) {
		if (capacity <= 0) {
			throw new IllegalArgumentException("capacity must be positive: "
					+ capacity);
		}
		size = 0;
		elementData = new int[capacity];
	}

	// Adds the given value to the end of the list.
	// If necessary, resizes the array to fit the value.
	public void add(int value) {
		// just call the other add method (to remove redundancy)
		add(size, value);
	}

	// Inserts the given value into the list at the given index.
	// If necessary, resizes the array to fit the value.
	// Precondition: 0 <= index <= size
	public void add(int index, int value) {
		checkIndex(index, 0, size); // okay to add at index == size (end of
		                                                    // list)
		checkResize();

		// slide elements to the right to make room
		for (int i = size; i > index; i--) {
			elementData[i] = elementData[i - 1];
		}

		// insert the value in the hole we just made
		elementData[index] = value;
		size++;
	}
```

```
// Returns the value in the list at the given index.
// Precondition: 0 <= index < size
public int get(int index) {
	checkIndex(index, 0, size - 1);
	return elementData[index];
}

// Sets the given index to store the given value.
// Precondition: 0 <= index < size
public void set(int index, int value) {
	checkIndex(index, 0, size - 1);
	elementData[index] = value;
}

// Returns the number of elements in the list.
public int size() {
	return size;
}

// Returns true if the list does not contain any elements.
public boolean isEmpty() {
	return size == 0; // "boolean zen"
}

// Removes the value from the given index, shifting following elements left
// by 1 slot to cover the hole.
// Precondition: 0 <= index < size
public void remove(int index) {
	checkIndex(index, 0, size - 1);
	for (int i = index; i <= size - 1; i++) {
		elementData[i] = elementData[i + 1];
	}
	size--;
}

// Returns a String representation of the elements in the list, such as
// "[42, -3, 17, 99]", or "[]" for an empty list.
public String toString() {
	if (size > 0) {
		String result = "[" + elementData[0];
		for (int i = 1; i < size; i++) {
			result = result + ", " + elementData[i];
		}
		result += "]";
		return result;
	} else {
		return "[]"; // empty list
	}
}

// Returns the index of the first occurrence of the given value in the list,
// or -1 if the value is not found in the list.
public int indexOf(int value) {
	for (int i = 0; i < size; i++) {
		if (elementData[i] == value) {
```

```
                        return i;
                    }
                }
                return -1; // not found
        }

        // Returns true if the given value is found in this list.
        public boolean contains(int value) {
                return indexOf(value) >= 0;
        }

        // A "private helper method" to resize the array if necessary.
        // Checks whether the list's array is full, and if so,
        // doubles its size so that more elements can be added.
        private void checkResize() {
                if (size == elementData.length) {
                        // resize the array
                        elementData = Arrays.copyOf(elementData, 2 * size);
                }
        }

        // A helper that throws an exception unless the given index is between the
        // given minimum / maximum values, inclusive.
        private void checkIndex(int index, int min, int max) {
                if (index < min || index > max) {
                        throw new ArrayIndexOutOfBoundsException();
                }
        }
}
```