



НАЦІОНАЛЬНИЙ
УНІВЕРСИТЕТ
КОРАБЛЕБУДУВАННЯ
ІМЕНІ АДМІРАЛА МАКАРОВА

Object-Oriented Programming in the Java language

Part 2. Control Flow + TDD



Yevhen Berkunskyi, NUoS
eugeny.berkunsky@gmail.com
<http://www.berkut.mk.ua>



Overview of Control Flow Statements

- Selection statements: **if**, **if-else**, and **switch**.
- Iteration statements: **while**, **do-while**, *basic for*, and *enhanced for*.
- Transfer statements: **break**, **continue**, **return**, **try-catch-finally**, **throw**, and **assert**.

Selection Statements

- simple **if** statement
- **if-else** statement
- **switch** statement

The Simple if Statement

- The simple if statement has the following syntax:

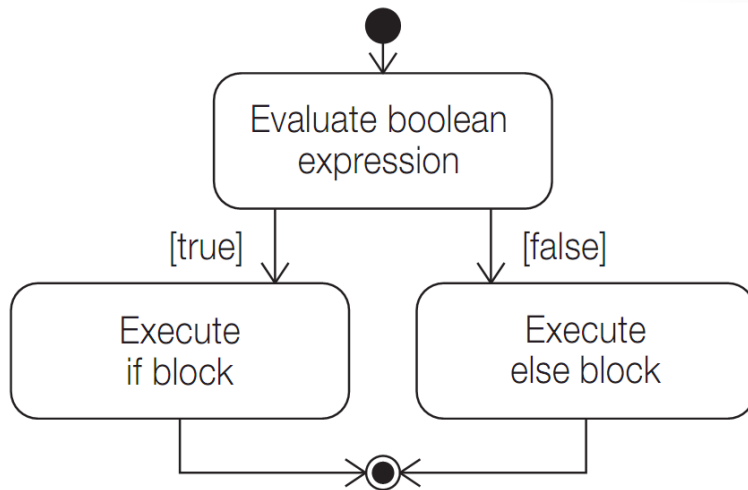
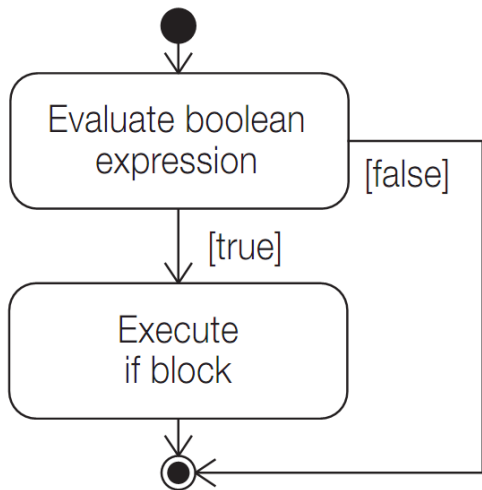
```
if (<conditional expression>) <statement>
```

Examples:

```
// emergency is a boolean variable  
if (emergency) operate();
```

```
if (temperature > critical)  
    soundAlarm();
```

Activity Diagram for if Statements



The Simple if Statement

Note that <statement> can be a block, and the block notation is necessary if more than one statement is to be executed when the <conditional expression> is true.

```
if (catIsAway()) {           // Block
    getFishingRod();
    goFishing();
}
```

The Simple if Statement

Note that the if block can be any valid statement. In particular, it can be the empty statement (;) or the empty block ({}). A common programming error is an inadvertent use of the empty statement.

```
if (emergency) ;  
// Empty if block operate() ;  
// Executed regardless of whether  
// it was an emergency or not.
```

The if-else Statement

The if-else statement is used to decide between two actions, based on a condition. It has the following syntax:

```
if (<conditional expression>
    <statement1>
else
    <statement2>
```


The if-else Statement examples

```
if (emergency)
    operate();
else
    joinQueue();
if
    (temperature > critical)
```

```
if (catIsAway()) {
    getFishingRod();
    goFishing();
} else
    playWithCat();
```

```
    soundAlarm();
```

```
else
```

if statements can be nested

```
if (temperature >= upperLimit) {  
  // (1) if (danger)  
  // (2) Simple if.  
    soundAlarm();  
  if (critical) // (3)  
    evacuate();  
  else // Goes with if at (3).  
    turnHeaterOff();  
} else // Goes with if at (1).  
  turnHeaterOn();
```

Use of block notation {}

```
// (A): Block notation
if (temperature > upperLimit) { // (1)
    if (danger) soundAlarm(); // (2)
} else // Goes with if at (1).
    turnHeaterOn();

// (B): Without block notation
if (temperature > upperLimit) // (1)
    if (danger) soundAlarm(); // (2)
else turnHeaterOn(); // Goes with if at (2).

// (C):
if (temperature > upperLimit) // (1)
    if (danger) // (2)
        soundAlarm();
else // Goes with if at (2).
    turnHeaterOn();
```

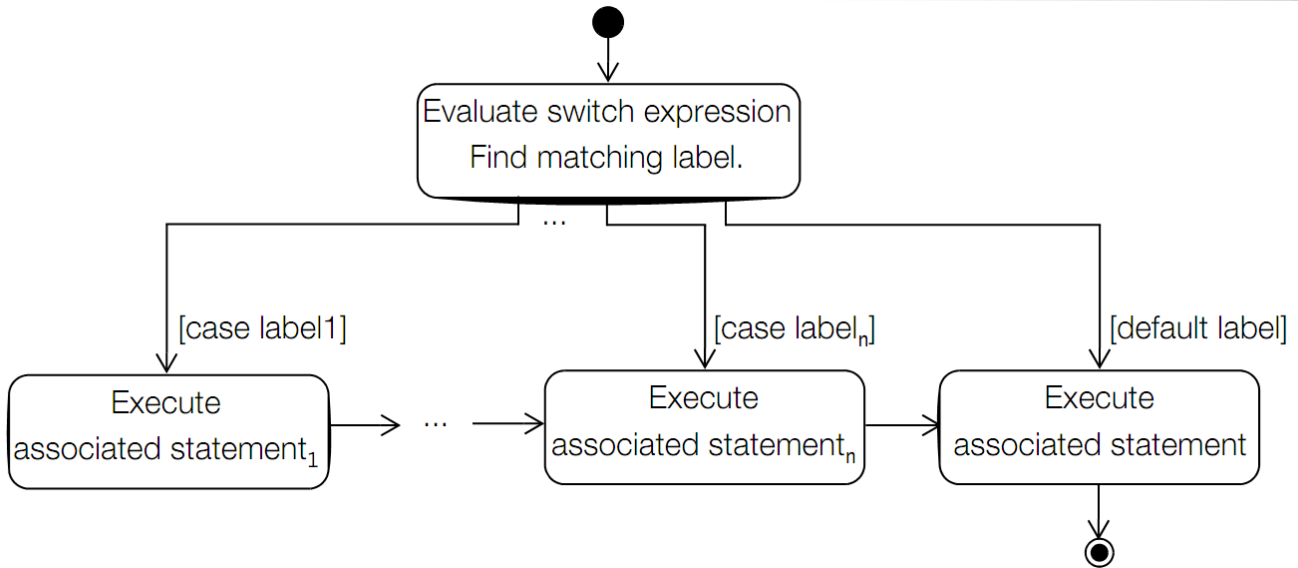
Cascading if-else statements

```
if (temperature >= upperLimit) { // (1)
    soundAlarm();
    turnHeaterOff();
} else if (temperature < lowerLimit) { // (2)
    soundAlarm();
    turnHeaterOn();
} else if (temperature==(upperLimit+lowerLimit)/2) { // (3)
    doingFine();
} else // (4)
    noCauseToWorry();
```

The switch Statement

```
switch (<switch expression>) {  
    case label1: <statement1>  
    case label2: <statement2>  
    ...  
    case labeln: <statementn>  
    default: <statement>  
} // end switch
```

Activity Diagram for a switch Statement



Fall Through in a switch Statement

```
public class Advice {  
    public final static int LITTLE_ADVICE    = 0;  
    public final static int MORE_ADVICE     = 1;  
    public final static int LOTS_OF_ADVICE  = 2;  
    public static void main(String[] args) {  
        dispenseAdvice(LOTS_OF_ADVICE);  
    }  
    .....// in the next slide  
}
```

Activity Diagram for a switch Statement

```
public static void dispenseAdvice(int howMuchAdvice) {  
    switch(howMuchAdvice) { // (1)  
        case LOTS_OF_ADVICE:  
            System.out.println("See no evil."); // (2)  
        case MORE_ADVICE:  
            System.out.println("Speak no evil."); // (3)  
        case LITTLE_ADVICE:  
            System.out.println("Hear no evil."); // (4)  
            break; // (5)  
        default:  
            System.out.println("No advice."); // (6)  
    }  
}
```


Using break in a switch Statement

```
public static String digitToString(char dig) {  
    String str = "";  
    switch(dig) {  
        case '1': str = "one";    break;  
        case '2': str = "two";    break;  
        case '3': str = "three";  break;  
        case '4': str = "four";   break;  
        case '5': str = "five";   break;  
        case '6': str = "six";    break;  
        case '7': str = "seven";  break;  
        case '8': str = "eight";  break;  
        case '9': str = "nine";   break;  
        case '0': str = "zero";   break;  
        default:  
            System.out.println(dig+" is not a digit!");  
    }  
    return str;  
}
```

Iteration Statements

Java provides four language constructs for loop construction:

- the **while** statement
- the **do-while** statement
- the **basic for** statement
- the **enhanced for** statement

The while Statement

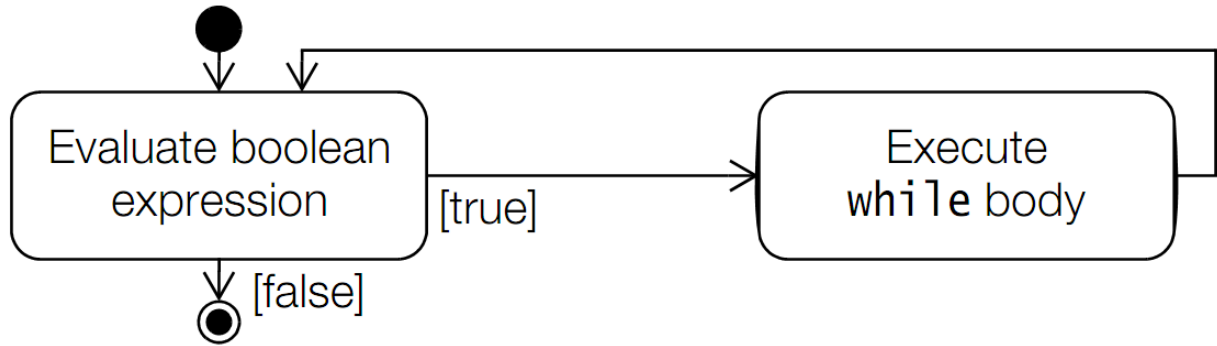
The syntax of the while loop is

```
while (<loop condition>)  
    <loop body>
```

The <loop condition> is evaluated before executing the <loop body>. The while statement executes the <loop body> as long as the <loop condition> is true.

When the <loop condition> becomes false, the loop is terminated and execution continues with the statement immediately following the loop.

Activity Diagram for the `while` Statement



The `while` statement is normally used when the number of iterations is not known.

```
while (noSignOfLife ())  
    keepLooking ();
```

The while Statement (warning)

Since the <loop body> can be any valid statement, inadvertently terminating each line with the empty statement (;) can give unintended results.

Always using a block statement, { ... }, as the <loop body> helps to avoid such problems.

```
//Empty statement as loop body!  
while (noSignOfLife());  
    keepLooking();  
// Statement not in the loop body.
```

The do-while Statement

The syntax of the do-while loop is

do

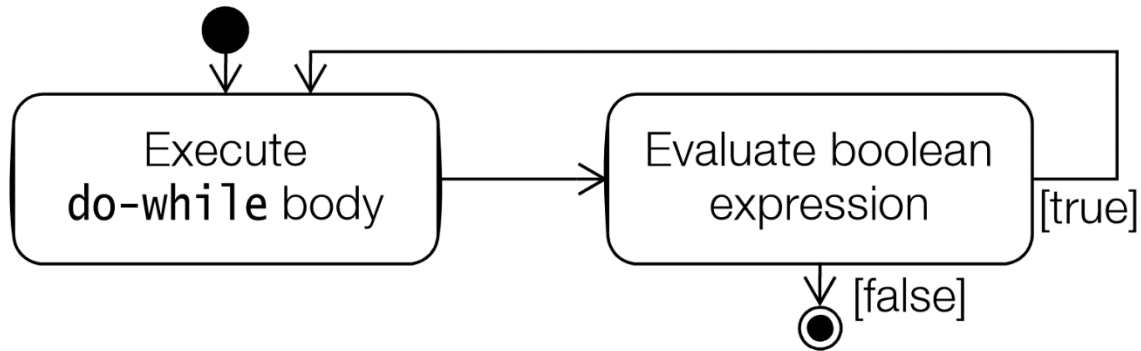
```
<loop body>
```

```
while (<loop condition>);
```

The `<loop condition>` is evaluated after executing the `<loop body>`. The value of the `<loop condition>` is subjected to unboxing if it is of the type `Boolean`. The do-while statement executes the `<loop body>` until the `<loop condition>` becomes false.

When the `<loop condition>` becomes false, the

Activity Diagram for the do-while Statement



while and do-while

The <loop body> in a do-while loop is invariably a statement block. It is instructive to compare the while and the do-while loops.

In the examples below, the mice might never get to play if the cat is not away, as in the loop at (1). The mice do get to play at least once (at the peril of losing their life) in the loop at (2).

```
while (cat.isAway()) {           // (1)
    mice.play();
}
```

```
do {                               // (2)
    mice.play();
} while (cat.isAway());
```


The `for (; ;)` Statement

The `for(;;)` loop is the most general of all the loops. It is mostly used for counter-controlled loops, i.e., when the number of iterations is known beforehand.

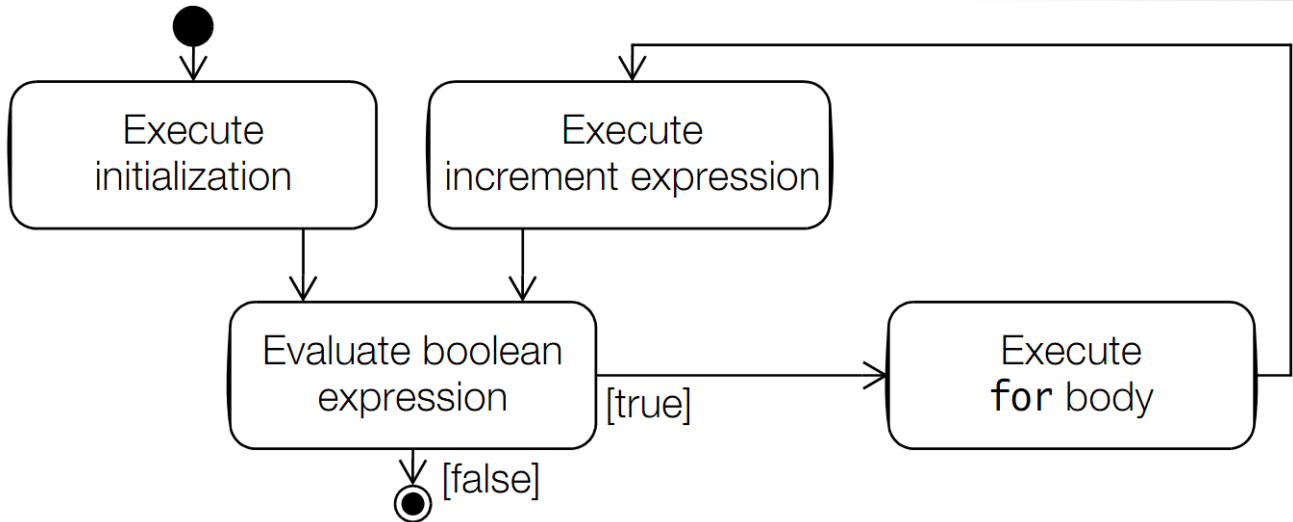
The syntax of the loop is as follows:

```
for (<initialization>;  
      <loop condition>;  
      <increment expression>)  
    <loop body>
```

The semantics of the `for (; ;)` loop

```
<initialization>  
while (<loop condition>) {  
    <loop body>  
    <increment expression>  
}
```

Activity Diagram for the for Statement



for statement examples

```
int sum = 0;
int[] array = {12, 23, 5, 7, 19};
for (int index = 0; index < array.length; index++) // (1)
    sum += array[index];

for (int i = 0, j = 1, k = 2; ... ; ...) ...; // (2)

for (int i = 0, String str = "@"; ... ; ...) ...; // (3)
//Compile time error.

int i, j, k; // Variable declaration
for (i = 0, j = 1, k = 2; ... ; ...) ...; // (4)
//Only initialization
```

for statement examples

```
// (5) Not legal and ugly:  
for (int i = 0, System.out.println("not legal!");  
      flag; i++) { //Error!  
    // loop body  
}
```

```
// (6) Legal, but still ugly:  
int i;          // declaration factored out.  
for (i = 0, System.out.println("legal!");  
      flag; i++) { // OK.  
    // loop body  
}
```

for statement examples

The <increment expression> can also be a comma-separated list of expression statements. The following code specifies a for(;;) loop that has a comma-separated list of three variables in the <initialization> section, and a comma-separated list of two expressions in the <increment expression> section:

for statement examples

```
// Legal usage but not recommended.  
int[][] sqMatrix = { {3, 4, 6}, {5, 7, 4}, {5, 8, 9} };  
for (int i = 0,  
      j = sqMatrix[0].length - 1,  
      asymDiagonal = 0;           // initialization  
      i < sqMatrix.length;       // loop condition  
      i++, j--) // increment expression  
  asymDiagonal += sqMatrix[i][j]; // loop body
```

for (; ;) statement

All sections in the **for** (; ;) header are optional. Any or all of them can be left empty, but the two semicolons are mandatory. In particular, leaving out the <loop condition> signifies that the loop condition is true.

The “**crab**”, (; ;), is commonly used to construct an infinite loop, where termination is presumably achieved through code in the loop body (see next section on transfer statements):

```
for ( ; ; ) Java.programming ( ) ;  
// Infinite loop
```


The for(:) Statement

The element variable is local to the loop block and is not accessible after the loop terminates.

Also, changing the value of the current variable does not change any value in the array.

The loop body, which can be a simple statement or a statement block, is executed for each element in the array and there is no danger of any out-of-bounds errors.

Transfer Statements

Java provides six language constructs for transferring control in a program:

- break
- continue
- return
- try-catch-finally
- throw
- assert

Labeled Statements

A statement may have a label.

`<label> : <statement>`

A label is any valid identifier and it always immediately precedes the statement.

Label names exist in their own name space, so that they do not conflict with names of packages, classes, interfaces, methods, fields, and local variables.

Labeled Statements

A statement can have multiple labels:

```
LabelA: LabelB:  
System.out.println(  
    "Multiple labels. Use  
judiciously.");
```

A declaration statement cannot have a label:

```
L0: int i = 0; // Compile time error.
```

A labeled statement is executed as if it was unlabeled, unless it is the break or continue statement.

The break Statement

The break statement comes in two forms:
the unlabeled and the labeled form.

```
break;           // the unlabeled form
```

```
break <label>; // the labeled form
```

Unlabeled break

The unlabeled break statement terminates loops (for(;;), for(:), while, do-while) and switch statements, and transfers control out of the current context (i.e., the closest enclosing block).

The rest of the statement body is skipped, and execution continues after the enclosing statement.

Labeled break

A labeled break statement can be used to terminate any labeled statement that contains the break statement.

Control is then transferred to the statement following the enclosing labeled statement.

In the case of a labeled block, the rest of the block is skipped and execution continues with the statement following the block:



Labeled break

```
out:
{ // (1) Labeled block
  // ...
  if (j == 10) break out;
  // (2) Terminate block. Control to (3).
  System.out.println(j);
  // Rest of the block not executed if j == 10.
  // ...
}
// (3) Continue here.
```

The continue Statement

Like the break statement, the continue statement also comes in two forms: the unlabeled and the labeled form.

```
continue; // the unlabeled form
```

```
continue <label>; // the labeled form
```

The continue Statement

The continue statement can only be used in a **for (; ;)**, **for (:)**, **while**, or **do-while** loop to prematurely stop the current iteration of the loop body and proceed with the next iteration, if possible.

The continue Statement

- In the case of the while and do-while loops, the rest of the loop body is skipped, that is, stopping the current iteration, with execution continuing with the `<loop condition>`.
- In the case of the `for(;;)` loop, the rest of the loop body is skipped, with execution continuing with the `<increment expression>`.

The return Statement

The return statement is used to stop execution of a method and transfer control back to the calling code (also called the caller).

The usage of the two forms of the return statement is dictated by whether it is used in a void or a non-void method



The return Statement

Form of return Statement	In void Method	In Non-void Method
return;	optional	not allowed
return <expression>;	not allowed	mandatory, if the method is not terminated explicitly



Modern technology development in Java

- TDD
- Maven
- Gradle
- ...



History

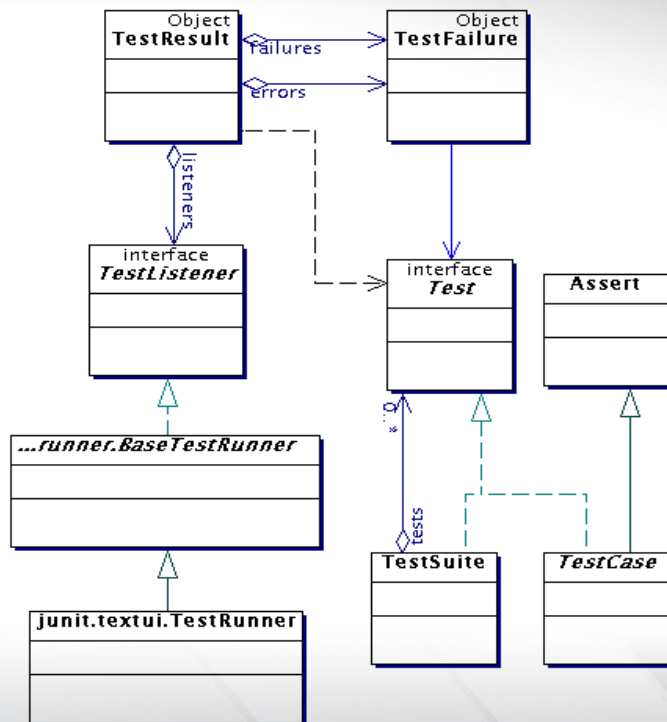
- Kent Beck developed the first xUnit automated test tool for Smalltalk in mid-90's
- Beck and Gamma (of design patterns Gang of Four) developed JUnit on a flight from Zurich to Washington, D.C.
- Martin Fowler: "Never in the field of software development was so much owed by so many to so few lines of 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 (IntelliJ IDEA, NetBeans, Eclipse, BlueJ, ...)

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 extending abstract class **TestCase** (optional)
- To write test cases, you need to know and understand the **Assert** class



Writing a TestCase

- To start using JUnit, create a subclass of *TestCase*, (optional in JUnit 4 and 5) to which you add test methods
- Name of class is important – should be of the form **MyClassTest**
- This naming convention lets TestRunner automatically find your test classes

```
import org.junit.jupiter.api.BeforeEach;
```

```
import static org.junit.jupiter.api.Assertions.*;
```

```
class MainTest {  
    @BeforeEach  
    void setUp() {
```

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 **testWhatever()**
 - 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 {**
 - 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 {
    Counter counter1;
    @BeforeEach
    protected void setUp() { // creates a test fixture
        counter1 = new Counter();
    }
    @Test
    public void testIncrement() {
        assertTrue(counter1.increment() == 1);
        assertTrue(counter1.increment() == 2);
    }
    @Test
    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 TestBowl("testBowl"));  
    suite.addTest(new TestBowl("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(TestBowl.class);  
    suite.addTestSuite(TestFruit.class);  
    return suite;  
}
```

JUnit Demo (for online)



Questions?





НАЦІОНАЛЬНИЙ
УНІВЕРСИТЕТ
КОРАБЛЕБУДУВАННЯ
ІМЕНІ АДМІРАЛА МАКАРОВА

Object-Oriented Programming in the Java language

Part 2. Control Flow + TDD



Yevhen Berkunskyi, NUoS
eugeny.berkunsky@gmail.com
<http://www.berkut.mk.ua>

